

武汉大学国家网络安全学院 本科生实验报告

操作系统设计与实践期末综合实验



专业名称：_____信息 安 全_____

课程名称：_____操作系统设计与实践_____

指导老师：_____严 飞 教 授_____

学生学号: _____20223302181145_____

学生姓名：_____周 业 营_____

2024 年 12 月 9 日

摘要

本实验报告主要介绍在操作系统设计与实践期末综合实验中，本人所负责题目的原理分析及其实现过程，也记录了小组其他成员负责任务的原理及实现过程。在最后的这次大实验中，本人主要负责的部分包括：Part A OS 综合装配-任务一（集成 OS）以及 Part B 安全性分析与可信防御实验-任务二：自我 OS 防护 - 2. 扩展 OS 的数据防泄露能力的技术路线一（对磁盘文件进行加密保护）。实验报告中主要将对这两部分的内容、原理和实现过程进行详细介绍与分析。

关键词： MiniOS; shell 指令; 系统日志; 数据加密; 多级反馈队列调度; 内存分配与释放

Contents

1 实验目的和内容	4
1.1 实验目的	4
1.2 实验内容	4
2 实验设计	6
2.1 概述	6
2.2 实验环境	6
2.2.1 工具简介	6
2.3 实验原理分析与实验步骤	8
2.3.1 Part A 任务一：集成你的 OS	8
2.3.2 Part A 任务二：扩展 shell 功能	28
2.3.3 Part A 任务三：扩展系统日志能力	29
2.3.4 Part B 任务一：自我 OS 安全分析	30
2.3.5 Part B 任务二：自我 OS 防护：针对发现的攻击，进行防护	31
2.3.6 Part B 任务二：自我 OS 防护：扩展 OS 的数据防泄露能力：磁盘文件 加密保护	33
2.3.7 Part B 任务二：自我 OS 防护：扩展 OS 的数据防泄露能力：文件资源 访问保护	46
2.4 实验结果与分析	47
2.4.1 实验中遇到的问题	47
2.4.2 Part A 任务一：集成你的 OS	47
2.4.3 Part A 任务二：拓展 shell	55
2.4.4 Part A 任务三：系统日志框架	58
2.4.5 Part B 任务一：自我 OS 分析	60
2.4.6 Part B 任务二：自我 OS 防护：针对发现的攻击，进行防护	61
2.4.7 Part B 任务二：自我 OS 防护 2. 扩展 OS 的数据防泄露能力	62
2.4.8 Part B 任务二：自我 OS 防护：扩展 OS 的数据防泄露能力：文件资源 访问保护	66
3 实验总结与心得	67
3.1 实验总结	67
3.2 个人心得体会	68

1 实验目的和内容

1.1 实验目的

1. **综合装配实验**，组装自己的简易操作系统：理解并掌握第 9 至 11 章的相关代码结构与 OrangeS 操作系统的功能。在已有实验代码基础上，综合前 1 至 8 章（可包含第 9 章）完成一个简易操作系统的设计与实现。
 - 多进程管理与调度
 - 内存分配与释放
 - 目录树结构管理
 - 拓展 shell 功能
 - 系统日志框架
2. **安全性分析与可信防御实验**：结合操作系统设计与软件安全知识，分析 OS 设计中的潜在安全问题。理解并掌握操作系统安全的基本思想与实现手段。通过分析与实现，增强 OS 的安全性和可信性。
 - 堆栈溢出
 - 磁盘文件加密保护
 - 进程文件访问权限

1.2 实验内容

Part A 部分

1. 集成 OS

在已有实验代码基础上，将 1-8 章节（可以包含第 9 章）进行功能综合，形成自己的一个简易 OS。

- 使用硬盘启动该 mini-OS
- 实现内存分配与释放
- 实现多进程管理与调度
- 用目录树结构管理代码，添加完整的 makefile 编译

2. 扩展 shell 功能

- 利用当前 OrangeS 所提供的系统调用和 API，扩展 Shell 命令
- 支持进程管理
 - 列出当前运行的进程
 - 终止指定的进程
- 支持文件管理
 - 列出当前目录的文件，以及文件的相关属性信息
 - 创建新文件
 - 打开或编辑指定文件（如果是可执行文件，则运行，如果是文本文件，则打开可编辑）

- 删除指定文件
- 支持在同一个 TTY 上，可并发运行多个 shell 任务

3. 扩展系统日志能力

- 增加系统日志的框架
 - 使用参数开关相应的系统日志
 - 对进程的运行过程，可以进行日志记录
 - 对文件的访问过程，可以进行日志记录
 - 对系统调用过程，可以进行日志记录
 - 对设备访问过程，可以进行日志记录

Part B 部分

1. 自我 OS 安全分析

- 分析提示：
可执行文件的篡改、内核关键、数据破坏、内存破坏漏洞、权限绕过等
- POC 实现：
编写若干你发现的漏洞 Demo 程序，对该 OS 实施攻击测试

2. 自我 OS 防护

针对你发现的攻击，进行防护-提示：

- 可执行文件篡改，可采用静态装入度量
- 运行时关键内核数据破坏，可采用运行时关键模块检查
- 缓冲区溢出，可采用添加类似堆栈保护等方式

3. 扩展 OS 的数据防泄露能力

- 对磁盘文件进行加密保护，文件打开时，读入内存的为明文，如果进行编辑，写回磁盘的为密文。自行设计密码算法、密钥管理方法。
- 技术路线二：白名单进程允许访问指定文件资源，非白名单进程不允许访问受保护文件资源。

2 实验设计

2.1 概述

本实验报告主要介绍在操作系统设计与实践期末综合实验中，本人所负责题目的原理分析及其实现过程，也记录了小组其他成员负责任务的原理及实现过程。在最后的这次大实验中，本人主要负责的部分包括：

1. Part A OS 综合装配-任务一（集成 OS）
2. Part B 安全性分析与可信防御实验-任务二：自我 OS 防护 - 2. 扩展 OS 的数据防泄露能力的技术路线一（对磁盘文件进行加密保护）

实验报告中主要将对这两部分的内容、原理和实现过程进行详细介绍与分析。最后 MiniOS 上实现了：

1. 使用硬盘启动 MiniOS
2. 实现内存分配与释放
3. 使用多级反馈队列调度算法进行多进程调度
4. 对磁盘文件进行加密保护

2.2 实验环境

- Windows Subsystem for Linux 2 (WSL 2)
- Ubuntu 20.04
- NASM 2.14.02
- Bochs 2.7
- Visual Studio Code (VSCode)
- GNU gdb (Ubuntu 9.2-0ubuntu1 20.04.2) 9.2

2.2.1 工具简介

1. WSL 2

- WSL2 (Windows Subsystem for Linux 2) 是微软为 Windows 系统提供的一个功能，使用户能够直接在 Windows 上运行完整的 Linux 操作系统（包括命令行工具和应用程序），无需安装双系统或使用虚拟机。

2. VSCode

- Visual Studio Code (VSCode) 是由微软推出的一款免费、开源、跨平台的代码编辑器。它支持多种编程语言，具有丰富的扩展生态系统和高度可定制的特性，可以特质化搭建自己的编程环境。

- VSCode 针对 OS 实验和其他学习开发环境的一大亮点是它具有对远程开发的强大支持，可以直接通过 Remote - WSL 插件连接 WSL。相比于 VMware 不友好的虚拟机体验，VSCode 远程连接的使用体验简直超模得没边。

3. Ubuntu 20.04

- 这是一个不得不提的点，我觉得相较于为了跑比较老旧的 OrangeOS 代码而故意去搭建一个 32 位的虚拟机环境，不如想办法在自己的 64 位环境上跑起来 OrangeOS，这应该是一个向下兼容的过程，而不是特质化回溯到旧的环境上去主动适应旧事物的过程。
- 而且这个向下兼容的过程也并不复杂，只要在 Makefile 中的编译参数中加上少许的参数就可以了，所以我觉得这是一个很划算的过程。具体 64 实验环境的搭建方式在集成 OS 部分我有比较详细的说明。
- 在这里也感谢好舍友 & 好队长黄东威同学，是他启蒙和引导了我的 VSCode+WSL Ubuntu 开发环境的搭建，他在学习和使用工具提高生产力的方面超越了我太多，教了我太多，也帮了我太多。

4. Bochs 2.7

- Bochs 是一个开源的、跨平台的、功能强大的 x86 PC 模拟器，主要用于虚拟化和模拟硬件环境，以支持开发和测试操作系统或底层程序。它可以用于学习计算机体系结构、调试操作系统内核或运行旧版本的操作系统。
- 在这次实验任务中，我使用了两种不同插件的 Bochs，第一个是支持 Bochs 调试的 Bochs 版本，第二个是支持远程 GDB 调试的 Bochs 版本。配置两个不同插件版本的 Bochs 是因为这两个版本互斥，而我又分别需要这两个版本的不同功能。使用支持远程 GDB 调试的 Bochs 版本是为了更好地调试代码，否则通过意念调试或者嵌入汇编后使用 magic_break 调试这样的一个微内核结构的 OS 实在是痛苦至极。使用支持 Bochs 调试的 Bochs 版本是为了完成集成 OS 中的内存分配与释放模块，在这一模块中，我需要在 loader.asm 中实现内存分配与释放功能，并且需要使用 magic_break 来打断点，呈现使用效果，而使用 GDB 调试的话这段装载内核的过程会透明化，我无法在这期间打断点调试和演示效果，所以在重新编译安装能够连接使用 GDB 调试的 Bochs 之后我又单独在用户目录下安装了一个支持 Bochs 调试的版本。
- 将支持远程 GDB 调试的 Bochs 版本装入根目录，默认使用这个版本的 Bochs 启动模拟器，然后将支持 Bochs 调试的版本放在用户目录下，通过使用直接路径的方式访问和使用指定的 Bochs 版本。

5. GDB

- 这是一个我觉得做晚了的事情，使用 GDB 进行代码调试这种比较普通、十分强大的调试方法应该早一些应用到实验的过程中，否则在前面的一些动手做代码编写时遇到 bug 就十分难调试，只能用 jmp \$ 或 magic_break 来调试还是很不方便。GDB 调试在完成实验的过程中起到的作用，用黄东威同学的形容来说就是“进入了工业革命时代”。
- 简述一下这部分实验环境的搭建过程。

- (a) 仍然是使用 Bochs 源码下载安装，然后可以使用下列指令来进行安装。使用`—enable-debugger`参数，Bochs 使用内置的反汇编器进行调试；使用`—enable-gdb-stub`参数，Bochs 开放端口供 GDB 远程调试；使用`—enable-readline`参数，可以让我们在使用 Bochs 内置调试器时使用 readline 库提供的自动补全和历史命令功能。

```
./configure —enable-debugger —enable-readline
make -j8
sudo make install
./configure —enable-gdb-stub
make -j8
sudo make install
```

- (b) bochs 中设置 gdb 通道与端口。

```
gdbstub: enabled = 1, port = 1234
```

- (c) 在 Makefile 时加上调试参数`-g`，去掉`-s`参数，`-s`参数的作用是从可执行文件中删除所有符号表和重定位信息。

```
ASMKFLAGS= -g -I include/ -I include/sys/ -f elf32
CFLAGS = -g -I include/ -I include/sys/ -c -fno-builtin -Wall -
LDFLAGS = -g -m elf_i386 -Ttext $(ENTRYPOINT) -Map krnl.map
```

- (d) 这时候生成的 kernel.bin 文件带有调试信息，文件太大，Bochs 无法直接启动，会出现“Too large”的提示，因为 Bochs 只能引导小于 100K 的内核文件。

- (e) 这就需要我们将带有调试信息的二进制文件单独存储下来，然后用 Bochs 启动不带调试信息的较小的内核文件，调试信息的内核文件在 GDB 连接前将调试信息导入到 GDB 中。可以直接修改 Makefile，用一组带调试信息参数的编译参数多生成一个 kernel.dbg 文件，或者在创建出一个 kernel.dbg 文件之后，将带调试信息的 kernel.bin 文件复制到.dbg 文件中。

```
> objcopy kernel.bin kernel.dbg && objcopy --strip-debug kernel.bin
```

- (f) 最后就是启动 bochs，然后新开一个中断，用 sudo 权限运行 gdb，加载符号表文件，远程连接 1234 端口。

```
(gdb) add-symbol-file kernel.dbg
```

```
(gdb) target remote :1234
```

2.3 实验原理分析与实验步骤

2.3.1 Part A 任务一：集成你的 OS

问题介绍

这部分任务是集成一个自己的 OS，核心目标是在现有的实验代码基础上，将课程前 1-8 章（可包含第 9 章）涉及的功能进行综合整合，构建一个可启动的简易操作系统（Mini-OS）。主要实现的技术要点是以下几个：

- 实现使用硬盘启动 Mini-OS
- 实现内存分配与释放功能
- 实现多级反馈队列的多进程调度

解决方案

这部分内容由本人完成，对于使用硬盘启动 Mini-OS，我在研读教材及项目代码的基础上，修改了一些项目代码中原有的 bug，使用 chapter11/c 目录代码作为基础集成 Mini-OS。对于内存分配与释放，我在王俊杰同学实现的内存分配与释放功能的基础上添加了测试函数，并调用分配释放页功能。对于实现复杂的多进程调度算法，综合调研得到的多种进程调度算法的特点、当前 MiniOS 的功能实现、进程调度算法的实现难度和之前小实验对多级反馈队列调度算法的讨论，最后选择了实现多级反馈队列调度算法。这部分工作将原本针对模拟的五个进程的多级反馈队列修改拓展完善为对所有进程进行调度，在当前 MiniOS 中存在多种不同系统级别的进程、多种不同运行状态的进程、父子进程，这部分任务的工作量并不小。

具体思路及实现

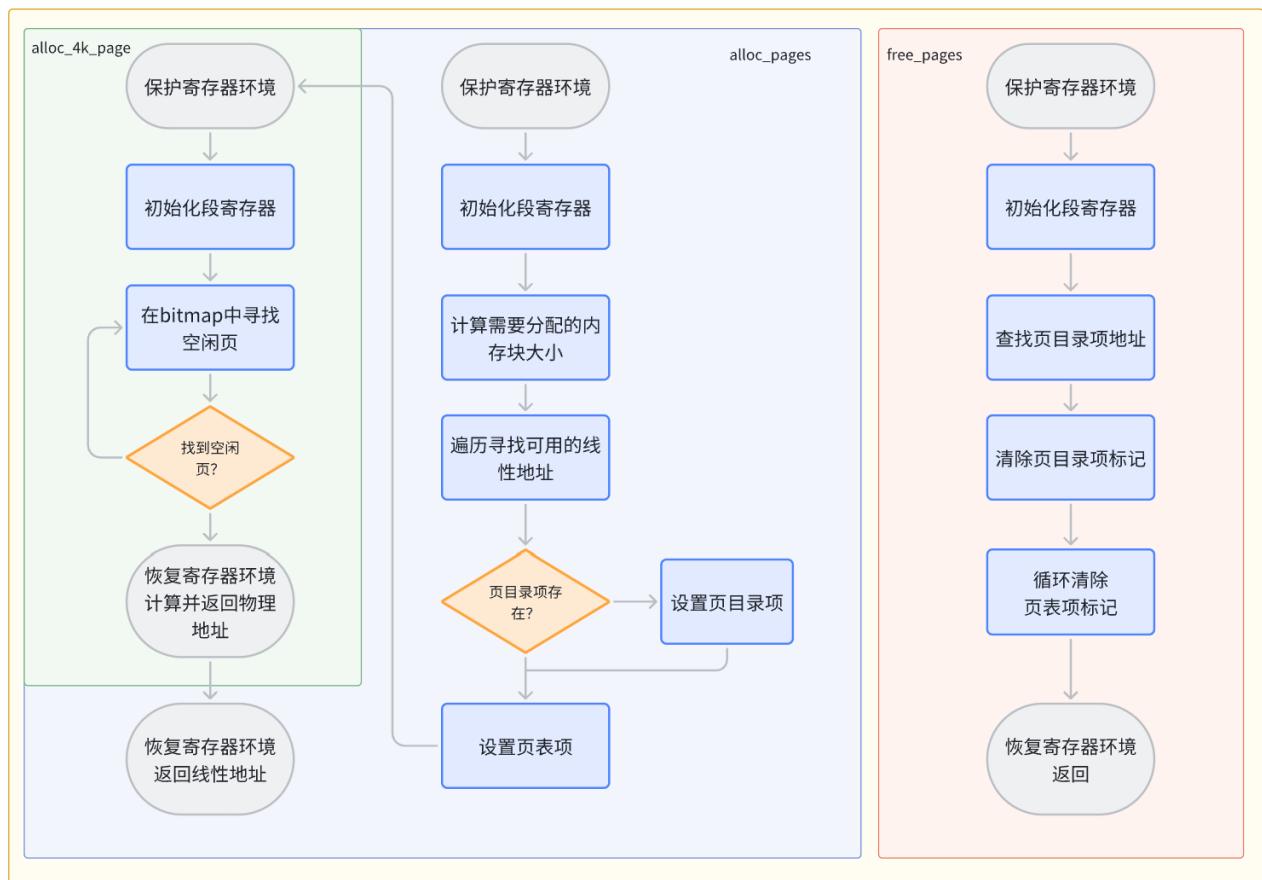


Figure 1: 内存分配与释放流程图

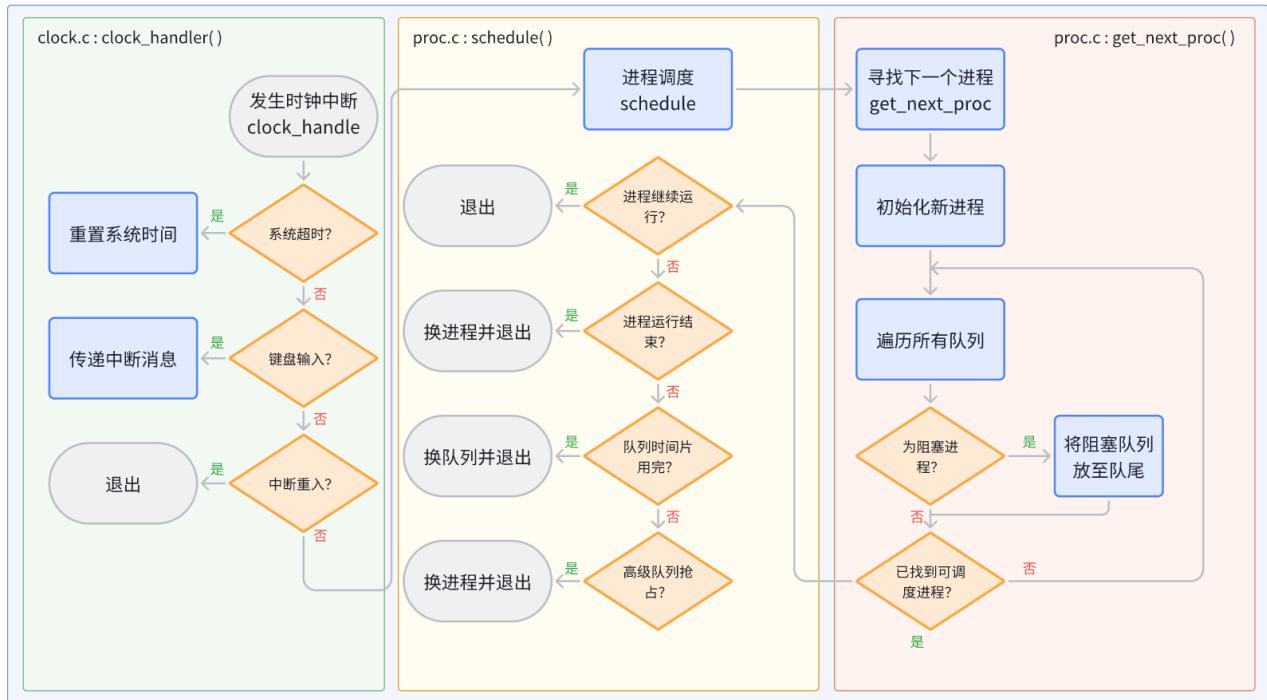


Figure 2: 多级反馈队列调度流程图

硬盘启动系统

系统从软盘启动的过程：

1. BIOS 将引导扇区读入内存 0000:7c00 处跳转到 0000:7c00 处开始执行引导代码
2. 引导代码从软盘中找到 loader.bin 并将其读入内存
3. 跳转到 loader.bin 开始执行
4. loader.bin 从软盘中找到 kernel.bin 并将其读入内存
5. 跳转到 kernel.bin 开始执行到此可认为启动过程结束

在第 1 步中 BIOS 到底读取软盘还是读取硬盘是由 CMOS 设置决定的。通常可以找到一个叫做“Boot Sequence”的选项，从中可以选择首选启动设备是软盘还是硬盘。在第 3 步和第 5 步中对于软盘启动过程的代码是在软盘中寻找 loader.bin 和 kernel.bin，然后将其装载到内存中运行。而对于使用硬盘启动，需要让引导扇区代码从硬盘中寻找 loader.bin，将其装载入内存并运行后，让 loader 从硬盘中寻找 kernel.bin，将 kernel 装载入内存并运行。这些便是软盘和硬盘启动的区别，剩下的几步中软盘和硬盘启动没有分别。

经过总结之后，改为使用硬盘启动系统的全过程为：

1. 计算安装位置

根据硬盘分区，确定目标分区的起始扇区。例如，将 Orange 的系统安装到硬盘镜像 100m.img 的第 7 个分区（对应设备号为 0x23），并记下该分区的起始扇区号）。

2. 设置相关文件

修改 boot/include/load.inc 文件中的 ROOT_BASE，将其设置为目标分区的起始地址。

修改 include/sys/config.h 文件中的 MINOR_BOOT 为目标分区的设备号（如 MINOR_hd2c）。

3. 配置磁盘文件

在 bochs 模拟器配置文件中，将硬盘镜像 100m.img 添加为主硬盘设备。

```
ata0-master : type = disk, path = 100m.img, mode = flat, cylinders = 203, heads = 16, spt = 63
```

4. 写入引导区

使用 dd 命令将引导扇区写入目标分区的引导区。

将 boot/hdboot.bin 写入引导区

```
dd if = boot/hdboot.bin of = 100m.img bs = 1 count = 446 conv = notrunc
```

写入分区表

```
dd if = boot/hdboot.bin of = 100m.img bs = 512 seek = 1 conv = notrunc
```

5. 准备内核文件和其他必要文件

将 hdldr.bin 和 kernel.bin 等必要文件打包为 cmd.tar，方便后续加载。

6. 格式化目标分区并写入文件

使用 mkfs 命令格式化目标分区为 Orange's 文件系统格式

7. 启动 GRUB 配置

如果硬盘未安装 GRUB，引导时需要通过 dd 命令将 GRUB 的 stage1 和 stage2 写入硬盘。

8. 启动系统

从硬盘启动，GRUB 提示符出现时输入命令：

- grub> rootnoverify (hd0,6)
- grub> chainloader +1
- grub> boot

具体的代码实现在/boot/hdboot.asm，这里不再赘述。但值得一提的是在 11 章的代码中，最后也没有选择使用硬盘启动操作系统，而是在实现了硬盘启动的功能之后还是默认用软盘启动系统，要想实现硬盘启动，还需要修改一下系统的引导程序。下面介绍一下如何从零开始让这份代码在本地跑起来，对这份代码做的修改，和对 bug 的修复。

从零开始 run 项目源代码 & 修复一些代码 bug

第一处是为了让我的 Ubuntu 20.04-64 位系统适应项目环境，我需要修改 makefile 的编译参数，让其按 32 位格式编译。在 ASMKFLAGS 参数设置后面的 elf 改为 elf32，CFLAGS 中加入-m32，LDFLAGS 中加上-m elf_i386。

为了使用 make image 编译，可能还需要运行一下 gzip -d 100m.img.gz，解压镜像。然后在这之后就会出现报错：

```

ld: kernel/main.o: in function `get_ticks':
main.c:(.text+0x3b7): undefined reference to `__stack_chk_fail_local'
ld: kernel/main.o: in function `untar':
main.c:(.text+0x78b): undefined reference to `__stack_chk_fail_local'
ld: kernel/main.o: in function `panic':
main.c:(.text+0xcbd): undefined reference to `__stack_chk_fail_local'
ld: kernel/keyboard.o: in function `.L83':
keyboard.c:(.text+0x67a): undefined reference to `__stack_chk_fail_local'
ld: kernel/tty.o: in function `tty_dev_write':
tty.c:(.text+0x621): undefined reference to `__stack_chk_fail_local'
ld: kernel/tty.o: tty.c:(.text+0x7d7): more undefined references to `__stack_chk_fail_local' follow
ld: kernel/bin: hidden symbol `__stack_chk_fail_local' isn't defined
ld: final link failed: bad value
make: *** [Makefile:95: kernel.bin] Error 1

```

这是因为链接器找不到栈保护相关的符号 `__stack_chk_fail_local`, 通常是因为没有提供栈保护实现, 或者在裸机开发中不需要栈保护。通过移除或禁用 `-fstack-protector` 编译选项就可以消除这个报错, 也就是在 `CFLAGS` 中加上 `-fno-stack-protector`。

```

sudo mount -o loop a.img /mnt/floppy/
mount: /mnt/floppy: mount(2) system call failed: Cannot allocate memory.
make: *** [Makefile:75: buildimg] Error 32

```

如果出现这个报错通常是因为 `a.img` 的镜像文件可能被破坏, 导致无法正确挂载, 重新 `gzip -d a.img.gz` 覆盖一下镜像文件应该就可以解决了。在这之后就可以正常编译和挂载镜像了。

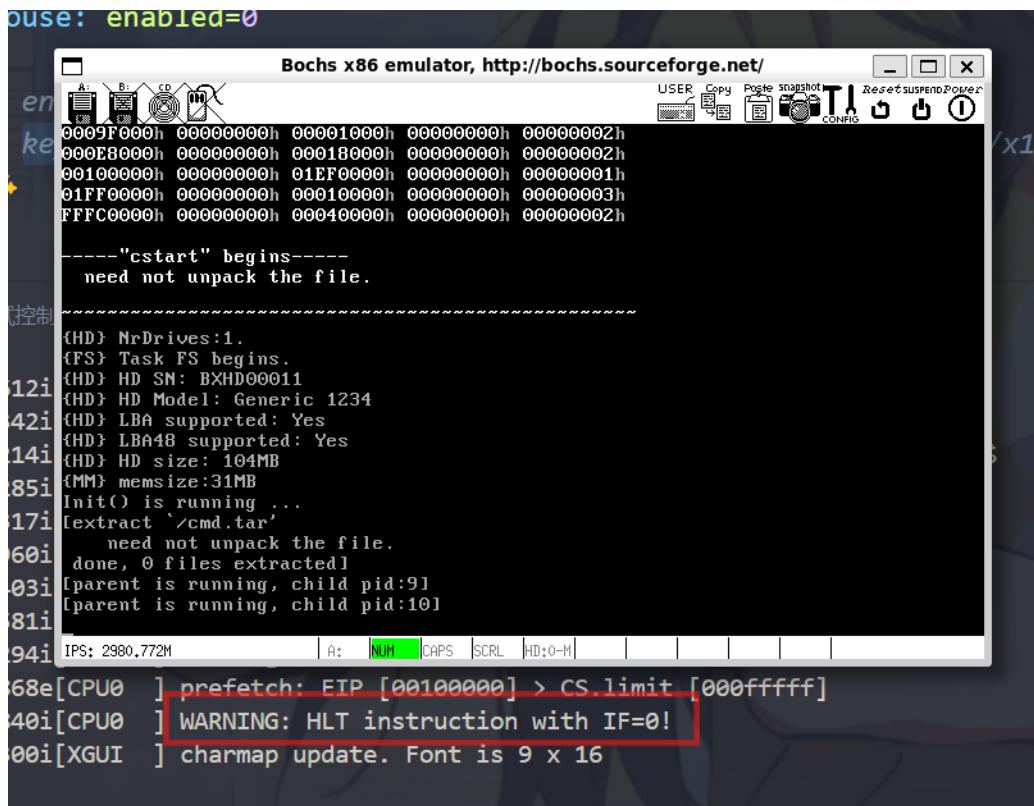
最后完整的 `Makefile` 的参数设置是这样的:

```

1 # Programs, flags, etc.
2 ASM      = nasm
3 DASM     = objdump
4 CC       = gcc
5 LD       = ld
6 ASMBFLAGS = -I boot/include/
7 ASMKFLAGS = -I include/ -I include/sys/ -f elf32
8 CFLAGS   = -I include/ -I include/sys/ -c -m32 -fno-builtin -Wall -fno
             -stack-protector
9 #CFLAGS   = -I include/ -c -fno-builtin -fno-stack-protector -fpack-
             struct -Wall
10 LDFLAGS  = -m elf_i386 -Ttext $(ENTRYPOINT) -Map krnl.map
11 DASMFFLAGS = -D
12 ARFLAGS  = rcs

```

第二处要修改的地方是 `bochssrc`, 这里的 `filename of ROM images` 两个路径建议就将文件放入这两个路径下, 以免每次运行一份代码都要修改 `bochssrc` 里的这两个路径。然后 `keyboard_mapping` 这一行配置是启用了 Bochs 的键盘映射功能, 并指定了使用 `x11-pc-us.map` 文件来控制键盘的行为和输入映射, 我的话是需要注释掉这一行。然后就可以 `bochs -q` 运行代码了。



但是这个时候又会出现一个名为 WARNING: HLT instruction with IF=0! 的 warning，这个时候 os 无法切换 tty，并且无法读入键盘输入，已经是死掉了的状态。出现这个 warning 的原因有很多种，在做实验的过程中还会遇到很多次，这像是一个通用的问题显示，针对不同的引发原因有不同的解决方案。

在这里针对这个问题做的事情是修改 kernel/kliba.asm 中的 disp_str 函数。这个问题在前面的实验中已经被发现了，关哥通过逆向分析，发现了问题。是由于目前新版本的编译器，在支持位置无关重定位时候，使用了 bx 寄存器，而 cstart 调用了 disp_str，该函数内部对 bx 寄存器又有访问，但是函数被调用时没有做 bx 的现场保护，因此导致错乱。在前面的章节中是显示字符串的时候是乱码，在这里导致的是系统卡死。解决方法就是对 bx 寄存器进行保护。

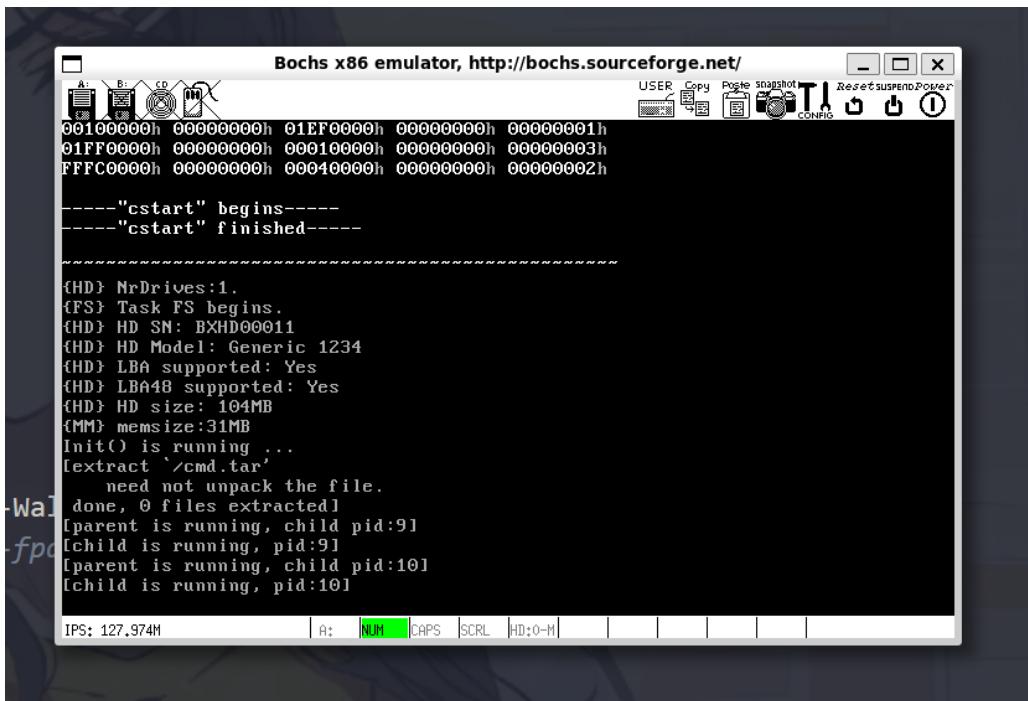
```
.1:
lodsb
test    al, al
jz     .2
cmp al, 0Ah ; 是回车吗?
jnz   .3
push   eax
push   ebx
mov eax, edi
mov bl, 160
div bl
and eax, OFFh
inc eax
mov bl, 160
mul bl
mov edi, eax
pop ebx
```

```
pop eax  
jmp .1
```

18
19

这个问题还有一个比较粗暴地解决方式，在 Makefile 里的 CFLAGS 中加入一个不支持位置无关的参数设置-fno-pie。位置无关可执行文件（PIE）是指可执行文件在内存中的加载地址可以是任意的，而不是固定的。通过这种方式，操作系统可以将进程加载到不同的内存地址，以增加安全性。这样做好处是可以防止某些攻击，如地址空间布局随机化（ASLR）。默认情况下，现代 Linux 系统和编译工具会启用 PIE，这样生成的可执行文件可以在任何内存地址上运行，这为内存保护和安全性提供了更高的保障。这样也能直接规避这个错误。

上面两种方法任选其一之后就可以正常运行起来代码了。

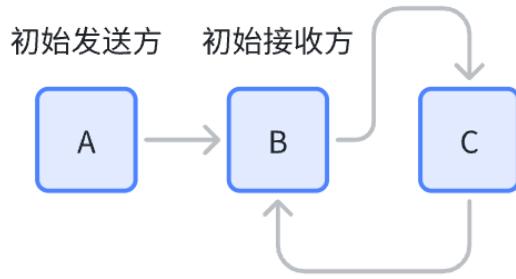


虽然现在正常运行起来了代码，但是系统中还是存在一些隐藏的漏洞，在后面的实验中可能会遇到奇奇怪怪的报错，在这里统一将所有修改都整合一下。

1. fork() 操作的原子性保护。在 mm/forexit.c 文件中 do_fork() 函数的开头和结尾加上开关中断。
2. msg_receive 操作的原子性保护。在 kernel/proc.c 文件中 msg_receive() 函数的开头和两处 return 前加上开关中断。这个原子性保护在后面复杂的进程间通信中很重要，如果不加这个保护系统很容易死掉。
3. 死锁检测算法错误。死锁的本质是一个进程等待的资源被另一个进程占用，且这些进程形成了一个环（循环等待）。但在学习的过程中我们也注意到，死锁一定有环，有环不一定有死锁，存在环路是形成死锁的必要条件。

所以在项目的程序中如果存在死锁也就是存在一个循环的链条：src -> dest -> ... -> src，其中每个进程都在等待下一个进程的资源。

这个死锁检测算法存在的问题在于这个死锁检测算法只能检测到以初始发送方为起止点的大环路，但如果在发送链中存在不经过初始发送方的小环路，则可能导致程序循环判断，陷入死循环，而无法返回是否有死锁的判断。示意图如下：



解决方案是将发送链条上的所有进程 ID 存储到一张哈希表中，如果 p 指针指到的新进程的 ID 与哈希表中的其他 ID 产生了哈希冲突，则表明存在死锁。虽然暂时没有因为这个死锁机制遇到过问题，不过还是将改进代码放在了下面。

```

1 PRIVATE int deadlock_plus(int src, int dest)
2 {
3     struct proc* p = proc_table + dest;
4     clear_hash_table(); // 每次检测死锁前都要清空哈希表
5
6     while (1) {
7         if (p->p_flags & SENDING) {
8             if (hash(p->p_sendto)) {
9                 /* print the chain */
10                int curcle = p->p_sendto;
11                int flag = 0;
12                p = proc_table + dest;
13                printf("=_=%s", p->name);
14                do {
15                    if(p->p_sendto == curcle) {
16                        flag++;
17                    }
18                    assert(p->p_msg);
19                    p = proc_table + p->p_sendto;
20                    printf("->%s", p->name);
21                } while (flag != 2);
22                printf("=_=");
23
24                return 1; // 发生了死锁
25            }
26            p = proc_table + p->p_sendto;
27        }
28        else {
29            break;
30        }
31    }
32    return 0;
33 }
34 /**
35 * *****global.c*****
36 PUBLIC int hash_table[HASH_TABLE_SIZE];

```

```

37  /************************************************************************/
38  extern int hash_table[];
39
40  /************************************************************************/
41 #define HASH_TABLE_SIZE NR_PROCS+NR_TASKS
42
43  /************************************************************************/
44 int hash(int src) {
45     int index = src % HASH_TABLE_SIZE; // 使用简单取模法生成哈希值
46
47     if (hash_table[index] == src) {
48         return 1; // 冲突，表示可能存在死锁
49     } else if (hash_table[index] == 0) {
50         hash_table[index] = src; // 将进程 ID 存入哈希表
51         return 0; // 正常插入
52     } else {
53         return 1; // 发生了冲突，出现了死锁
54     }
55 }
56
57 void clear_hash_table() {
58     for (int i = 0; i < HASH_TABLE_SIZE; i++) {
59         hash_table[i] = 0; // 重置哈希表
60     }
61 }
62 }
```

内存分配与释放

这部分内容在之前的实验中已经做过，这里做的主要是一个集成的任务，唯一可能有区别和有难度的就是之前做小实验的时候是在 chapter3，代码结构和文件结构相对比较清晰的项目中实现的，现在需要在如今更完善的大 MiniOS 上集成原来的代码。然后第二个是之前实现的时候写了 alloc_pages 和 free_pages 函数但是没有实际地使用起来，这里写了一个测试函数进行调用和检测。

代码的全部实现过程在 boot/loader.asm 中完成，先在数据段增加新的变量：

1. _PageTableName

在分页机制中，页表用于将虚拟地址映射到物理地址。这个变量用于记录当前系统中页表的数量。

2. _BitMap

BitMap（位图）用于跟踪内存页的使用情况。每一位表示一个内存页的状态，1 表示已分配，0 表示未分配。

3. BitMapLen

位图的长度

4. _AvaLinearAddress

用于记录当前可用的线性地址的起始位置。

5. __Linear

用于存储字符串。

```
1127: _PageTableNumber dd 0
1128:
1129:
1130: _BitMap: times 32 db 0xFF ; low 1mb is occupied
1131:      times 32 db 0x00 ; 1mb is available
1132: _BitMapLen equ $ - _BitMap
1133: _AvalinearAddress dd 0x8000_0000
1134: _Linear: db 0Ah, "Linear Address: ",0
1135: _Physical:
1136:
1137;
```

```
1156: _dwLengthLow equ LOADER_PHY_ADDR + _dwLengthLow
1157: _dwLengthHigh equ LOADER_PHY_ADDR + _dwLengthHigh
1158: _dwType equ LOADER_PHY_ADDR + _dwType
1159: _PageTableNumber equ _PageTableNumber - $$
1160: _BitMap equ _BitMap - $$
1161: _AvalinearAddress equ _AvalinearAddress - $$
1162: _Linear equ _Linear - $$
1163: _Physical equ _Physical - $$
1164:
1165:
1166:
```

然后实现 alloc_a_4k_page: 函数来以 4K 为粒度进行页分配。

```
alloc_a_4k_page:
; 返回值: eax : 分配的物理地址
; 物理地址从 0x00000000 开始
; 保护寄存器
push ds
push es

xor eax, eax
mov ax, SelectorFlatRW
mov es, ax
mov ax, SelectorFlatRW
mov ds, ax
; 初始化段寄存器

.search:
bts [BitMap], eax
; 在位图中寻找是否存在空闲页
; bts 指令将位图中 eax 位设置为 1，并
; 返回原来的值
; 如果原来的值为 0，表示找到空闲页，

jnc .find
跳转到 .find
inc eax
; 否则，继续检查下一个位
cmp eax, BitMapLen*8
; 判断是否超过最大位数
jl .search
; 如果没有超过，继续搜索
hlt
; 如果超过，表示没有空闲页，程序结束

.find:
shl eax, 12
; 找到空闲页后，左移 12 位，计算物理
; 地址
pop es
; 恢复 es 寄存器
pop ds
; 恢复 ds 寄存器
ret
; 返回，eax 中包含物理地址
```

实现 alloc_pages 函数和 free_pages 函数，分别实现建立地址间的映射关系和消除地址间的映射关系的功能。代码中写了详细的注释，实现的细节就不再赘述。

```
alloc_pages:
; 传参: eax : 需要分配的页数
; 返回值: ebx : 分配的连续物理页的起
; 始地址
push ds
push es
; 保护寄存器

mov bx, SelectorFlatRW
```

```

mov ds, bx
mov bx, SelectorFlatRW
mov es, bx
; 初始化段寄存器

mov ecx, eax
mov ebx, 4096
mul ebx
数 * 4096)
; ecx = 需要分配的页数
; 每个物理页的大小 (4KB)
; 计算需要分配的内存块大小 (eax = 页
; 数 * 4096)

mov ebx, [es:AvaLinearAddress] ; 读取当前可用的线性内存地址起始地址
add [es:AvaLinearAddress], eax ; 更新分配之后的线性地址
push ebx
; 保存分配的线性地址起始地址

; 设置页目录项和页表项

mov eax, ebx
mov ebx, cr3
and ebx, 0xfffff000
and eax, 0xffc00000
shr eax, 20
add ebx, eax
mov edx, ebx
mov ebx, [ebx]
; ebx = 对应的页表项地址

test ebx, 0x0000_0001
jnz .pde_exist
; 检查页目录项是否存在
; 如果存在，跳转到 .pde_exist

mov ebx, cr3
mov ebx, [ebx]
and ebx, 0xfffff000
shl eax, 10
add ebx, eax
or ebx, 0x0000_0007
mov [edx], ebx
; 获取页目录表地址 (清除低12位)
; 左移10位，得到已使用页的大小
; 设置页目录项的权限位和存在位
; 更新页目录项

.pde_exist:
mov eax, [esp]
and ebx, 0xfffff000
and eax, 0x003ff000
shr eax, 10
add ebx, eax
; 设置页表项
; eax = 分配的线性地址起始地址
; 清除 ebx 的后12位
; 获取页表项索引 (清除高20位)
; 右移10位，得到页表项索引
; 获取到页表项地址

.change_pte:
call alloc_a_4k_page
or eax, 0x00000007
mov [ebx], eax
add ebx, 4
loop .change_pte
; 分配一个4KB的物理页
; 设置页表项的权限位和存在位
; 更新页表项
; 移动到下一个页表项
; 循环，直到所有页表项都设置完毕

pop ebx
pop es
pop ds

```

```

ret                                57
free_pages:
; 传参: eax : 线性地址, ebx : 页数      58
; 返回值: 无                               59
push ds                            60
push es                            61
push ebx                           62
push eax                           63
; 初始化段寄存器                         64
; 65
mov bx, SelectorFlatRW           66
mov ds, bx                         67
mov bx, SelectorFlatRW           68
mov es, bx                         69
; 初始化段寄存器                         70
; 71
; 查找页目录项和页表项
mov ebx, cr3                      72
and ebx, 0xfffffff000             73
and eax, 0xffc00000               74
shr eax, 20                        75
add ebx, eax                      76
; ebx 现在指向页目录项                  77
mov edx, [ebx]                    77
and edx, 0xfffffffff8             78
mov [ebx], edx                   79
; 清除页目录项的最后 3 位并存储回去     79
; 80
mov ebx, [ebx]                    81
; 现在 ebx 指向第一个页表项             81
; 82
; 现在 eax 是线性地址                  83
mov eax, [esp]                    83
add esp, 4                         84
and ebx, 0xfffffff000             85
and eax, 0x003ff000               86
shr eax, 10                        87
add ebx, eax                      88
; 现在 ebx 指向正确的页表项             88
; 89
mov ecx, [esp]                    90
add esp, 4                         91
.change_pte:
; 清除每个页表项的最后 3 位             92
mov eax, [ebx]                    93
and eax, 0xfffffffff8             94
mov edx, eax                      95
shr edx, 12                        96
btr [BitMap], edx                97
mov [ebx], eax                   98
add ebx, 4                         99
; 移动到下一个页表项                  100
loop .change_pte
; 101
pop es                            102
pop ds                            103
ret                                104

```

多级反馈队列调度算法

在选择要实现的进程调度算法之前，先调研一下常见的比较复杂的进程调度算法及它们的特点：

1. 多级反馈队列调度算法

- 将进程分配到多个优先级队列中，每个队列有不同的时间片长度。
- 新进程通常被分配到高优先级队列中。如果进程未在时间片内完成，降低其优先级。

2. 最短剩余时间优先调度

- 最短作业优先 (SJF) 的抢占式版本。
- 当前正在运行的进程会被剩余执行时间更短的新进程抢占。

3. 带优先级的调度算法

- 每个进程分配一个优先级，优先级高的进程优先运行。
- 动态优先级（优先级可以随时间变化）可以缓解饥饿问题。

4. 倒数公平调度

- 使用平衡二叉树（红黑树）存储进程，基于虚拟运行时间选择下一个进程。
- 每个进程按照其权重分配运行时间，确保所有进程公平共享 CPU。

5. EDF 调度 (Earliest Deadline First)

- 适用于实时操作系统，按照任务的截止时间调度，截止时间最早的任务优先运行。

6. 轮转调度结合优先级

- 不同优先级的进程分组，高优先级组优先调度。
- 每个优先级组内使用时间片轮转。

7. 双队列调度

- 将任务分为两类（交互式任务和批处理任务），并分配到两个队列。
- 交互式任务优先调度，批处理任务在空闲时间执行。

综合进程调度算法的特点、当前 Minios 的功能实现、进程调度算法的实现难度和之前小实验对多级反馈队列调度算法的讨论，最后选择了实现多级反馈队列调度算法。

先定义和添加一些宏定义和变量。要实现多级反馈队列，先要定义出一个队列的结构体：结构体中有类型为进程体的进程队列，控制循环队列的头指针和尾指针，队列的长度和队列对应的时间片。

Listing 1: include/proc.h

```
1 #define QUEUE_LEN 50
2 typedef struct s_queue {
3     struct proc* taskqueue[QUEUE_LEN];
4     int front;
5     int rear;
6     int len;
7     int timep;
8 } QUEUE;
```

然后定义出三个队列，并为他们分配内存空间和定义每级队列的时间片长度，并声明为全局变量。

Listing 2: kernel/global.c

```
1 // 三级时间片分别为5,10,20
2 PUBLIC QUEUE queue[3] = {{ {0}, 0, 0, 0, 5 },
3                             { {0}, 0, 0, 0, 10 },
4                             { {0}, 0, 0, 0, 20 }};
```

Listing 3: include/sys/global.h

```
1 extern QUEUE queue[3];
```

除此之外，为了控制进程的调度，还需要在进程体结构中新添加进程需要的运行时间 runtime、是否在队列的标记 inqueue、在哪一级队列 queueenum。

Listing 4: kernel/sys/proc.h

```
1 struct proc {
2     ...
3     int runtime;           // 进程需要的运行时间
4     int inqueue;          // 是否在队列的标记
5     int queueenum;        // 在哪一级队列
6     ...
7 }
```

然后开始实现进程调度过程，对时钟中断处理程序进行修改，多加在每次时钟中断时减少当前运行的进程的 runtime。

Listing 5: kernel/clock.c

```
1 PUBLIC void clock_handler(int irq)
2 {
3     // 定时重置系统ticks
4     if (++ticks >= MAX_TICKS)
5         ticks = 0;
6
7     // 每次时钟中断都检查是否有键盘输入
8     if (key_pressed)
9         inform_int(TASK_TTY);
10
11    // 判断是否为中断重入，是否在内核态
12    if (k_reenter != 0) {
13        return;
14    }
15
16    // 每次时钟中断都减少当前进程的ticks
17    if (p_proc_ready->ticks)
18        p_proc_ready->ticks--;
19
20    // 每次时钟中断都减少当前进程的runtime
21    if (p_proc_ready->runtime) {
22        p_proc_ready->runtime--;
23    }
24}
```

```

25     schedule();
26
27     disp_str("pid:");
28     disp_int(p_proc_ready - &FIRST_PROC);
29     disp_str(",now in queue:");
30     disp_int(p_proc_ready->queuenum);
31     disp_str("\n");
32
33 }

```

具体的进程调度过程中封装了一个用于寻找下一个要运行的进程的函数 get_next_proc() 以及面向进程的出队入队函数。针对可以继续运行当前进程、当前进程运行结束、当前进程的队列时间片用完了需要切换队列、发生了抢占这四种情况下的运行逻辑进行代码实现。

Listing 6: kernel/proc.c

```

1 void schedule() {
2
3     struct proc *next = get_next_proc();
4
5     // 队列时间片还未用完，当前进程还没结束，未发生抢占，则继续运行当前进
6     // 程
7     if (p_proc_ready->ticks && p_proc_ready->runtime &&
8         next->queuenum > p_proc_ready->queuenum) {
9         return;
10    }
11
12    // 当前进程运行结束
13    if (p_proc_ready->runtime == 0) {
14        p_proc_ready = next;
15        outqueue(next);
16        return;
17    }
18
19    // 当前进程的时间片用完了，换到下一级队列
20    if (p_proc_ready->ticks == 0) {
21        // 当前时间片用完且不在最低级的队列则进到下一级队列，否则继续留在
22        // 当前队列
23        if (p_proc_ready->queuenum < 2) {
24            p_proc_ready->queuenum += 1;
25        }
26
27        // // 为了演示效果，将最低级队列出队后放到了最高级队列
28        // if (p_proc_ready->queuenum <= 2) {
29        //     p_proc_ready->queuenum = (p_proc_ready->queuenum + 1) % 3;
30        // }
31
32        // 切换队列或留在当前队列，均重新设置时间片
33        p_proc_ready->ticks = (queue + p_proc_ready->queuenum)->timep;
34
35        // 如果切换队列之后，下一个进程的优先级并不比切换之后的当前进程
36        // 高，则继续运行当前进程
37        if (next->queuenum > p_proc_ready->queuenum) {

```

```

35         return;
36     }
37     // 否则发生抢占
38     else {
39         enqueue(p_proc_ready);
40         p_proc_ready = next;
41         outqueue(next);
42         return;
43     }
44 }
45 }
46
47 // 当前进程的时间片没用完，但是下一个进程的优先级更高，发生抢占
48 if (next->queuenum < p_proc_ready->queuenum) {
49     enqueue(p_proc_ready);
50     p_proc_ready = next;
51     outqueue(next);
52     return;
53 }
54 }
```

对于寻找下一个可以运行的进程的函数，则考虑了初始化进程表中的进程、遍历三级队列寻找下一个运行的进程的功能。

Listing 7: kernel/proc.c

```

1 // 获取下一个要运行的进程
2 struct proc* get_next_proc() {
3     struct proc* p;
4     QUEUE* tempqueue;
5     int now_queue_num = 0;
6
7     // 遍历进程表，初始化进程的ticks,queuenum,runtime
8     for (p = &FIRST_PROC; p <= &LAST_PROC; p++) {
9         if (p->p_flags == 0 && p->enqueue == 0 &&
10             (p != p_proc_ready || queue->len == 0)) {
11             p->ticks = queue->timep;
12             p->queuenum = 0;
13             p->runtime = 100;
14             enqueue(p);
15         }
16     }
17
18     // 遍历三个队列，找到下一个运行的进程
19     now_queue_num = 0;
20     while (now_queue_num <= 2) {
21         tempqueue = queue + now_queue_num;
22         int length = 0;
23         int point = tempqueue->front;
24
25         // 对当前队列的所有进程进行处理，主要是对阻塞的进程进行处理，如果是阻塞的则先出队再入队
26         while (tempqueue->taskqueue[point]->p_flags != 0 &&
```

```

27         length < tempqueue->len) {
28             outqueue(tempqueue->taskqueue[point]);
29             inqueue(tempqueue->taskqueue[point]);
30             length += 1;
31             point = (point + 1) % QUEUE_LEN;
32         }
33
34         // 找到了一个能运行的进程就return
35         if (length < tempqueue->len) {
36             return tempqueue->taskqueue[point];
37         }
38         now_queue_num++;
39     }
40 }
```

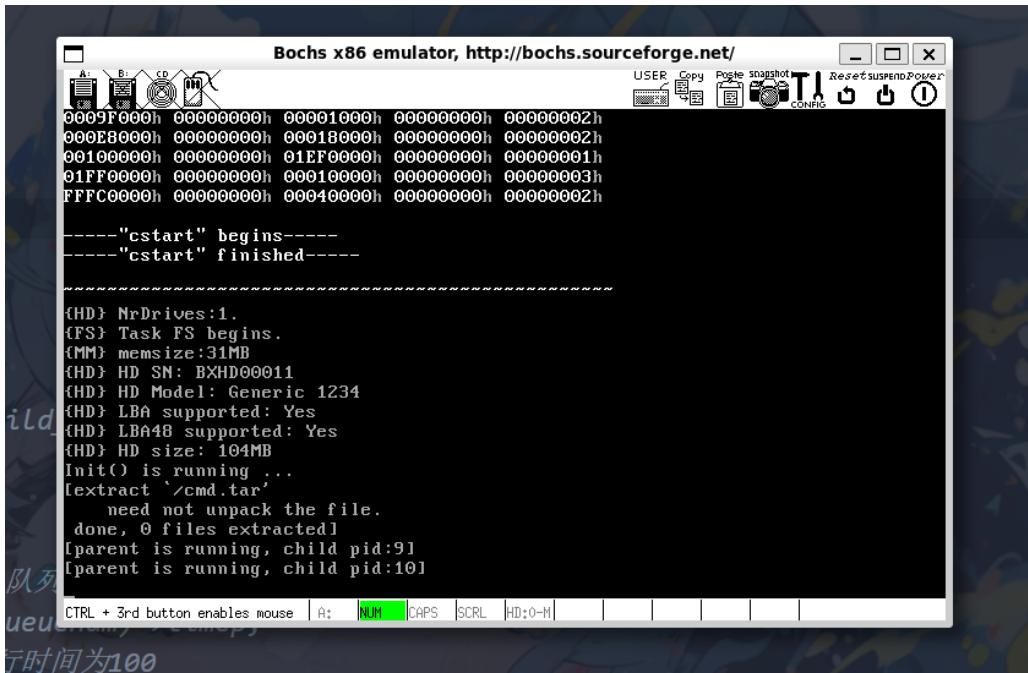
除此之外还有配套的辅助函数，进程出、入队函数，修改发生阻塞时的操作。

Listing 8: kernel/proc.c

```

1 void outqueue(struct proc* p) {
2     QUEUE* tempqueue;
3     tempqueue = queue + p->queuenum;
4     tempqueue->front = (tempqueue->front + 1) % QUEUE_LEN;
5     p->inqueue = 0;
6     tempqueue->len -= 1;
7 }
8
9 int inqueue(struct proc* p) {
10    QUEUE* tempqueue;
11    tempqueue = queue + p->queuenum;
12    tempqueue->taskqueue[tempqueue->rear] = p;
13    tempqueue->rear = (tempqueue->rear + 1) % QUEUE_LEN;
14    p->inqueue = 1;
15    tempqueue->len += 1;
16 }
17 ...
18 PRIVATE void block(struct proc* p) {
19     assert(p->p_flags);
20     inqueue(p);
21     struct proc* next = get_next_proc();
22     p_proc_ready = next;
23     outqueue(next);
24 }
```

在实际运行的过程中，出现了进程加载时卡死的情况。



发现是因为在 fork 创建子进程时，未给 fork 出来的子进程初始化导致子进程未能激活，系统初始化失败卡死的情况。针对这个问题需要对 fork 操作进行修改。

Listing 9: mm/forkexit.c

```

1 PUBLIC int do_fork() {
2     ...
3     /* birth of the child */
4     MESSAGE m;
5     m.type = SYS_CALL_RET;
6     m.RETVAL = 0;
7     m.PID = 0;
8     send_recv(SEND, child_pid, &m);
9
10    // 获取新创建的进程结构体
11    struct proc* new_proc = &proc_table[child_pid];
12
13    // 初始化新进程的队列属性
14    new_proc->inqueue = 0;
15    new_proc->queuenum = 0;
16    new_proc->ticks = (queue + new_proc->queuenum)->timestep;
17    new_proc->runtime = 100;
18
19    // 将新进程加入队列
20    inqueue(new_proc);
21
22
23    return 0;
24    ...
25 }
```

目录树结构管理代码

这部分内容可以下载 tree，然后在项目目录下运行 tree，即可展示出项目的目录树结构。项目的运行方法并没有修改。

```
1 // 正常启动 OS
2 > make image
3 > bochs -q
4
5 // 要使用Bochs调试功能就使用指定路径的Bochs，项目目录下已写好运行脚本
6 > ./run-page.fish
```

植入后门实现彩蛋

为了让实现的 OS 具有一些个人的标识，我在 keyboard 程序中植入了一个小的后门程序，用于输出独特的用户标识，并获取 shell 权限。

要实现一个彩蛋，首先得对键盘输入进行识别，并且定义一个特定后门字符串，这个事情就在 keyboard.c 中加入一个输入缓冲，用于读取键盘输入，以与后门字符串进行比对，从而实现彩蛋的内容。

```
1 #define TARGET_STRING "innerpeace"           // 特定后门字符串
2 #define TARGET_LENGTH (sizeof(TARGET_STRING) - 1) // 字符串的长度
3
4 PRIVATE char input_buffer[TARGET_LENGTH];    // 输入缓冲区
5 PRIVATE int input_index = 0;                  // 输入索引
```

然后在读取键盘输入的过程中，将读取到的字符与后门字符串进行比对，如果匹配成功则将匹配指针往后移动，否则将重置匹配指针，重新开始匹配。

所有字符都匹配成功之后，触发彩蛋程序，输出定制 logo，并传递彩蛋后门触发消息。

Listing 10: kernel/keyboard.c

```
1 // 检查输入的字符是否匹配特定字符串
2 if (key >= ' ' && key <= '~') {
3     input_buffer[input_index] = key;
4
5     // 检查当前输入的字符是否与目标字符串的对应字符匹配
6     if (input_buffer[input_index] == TARGET_STRING[input_index]) {
7         input_index++;
8         // 检查是否匹配整个特定字符串
9         if (input_index == TARGET_LENGTH) {
10             // 检查当前是否是 tty0
11             if (tty == &tty_table[0]) {
12                 for (int i = 0; i < 4; i++) {
13                     printf("\n");
14                 }
15                 printf("%s\n", logo);
16
17                 for (int i = 0; i < 10; i++) {
18                     printf("\n");
19                 }
20                 printf("Welcome to my Mini-OS!\nA new shell has been
21                         created in tty2!\n");
```

```

21         printf("\n\n\n");
22
23         backdoor = 1;
24
25     }
26     // 输出完再次重置输入缓冲区
27     input_index = 0;
28     memset(input_buffer, 0, TARGET_LENGTH);
29 }
30 } else {
31     // 如果不匹配，重置输入缓冲区
32     input_index = 0;
33     memset(input_buffer, 0, TARGET_LENGTH);
34 }
35 }
```

修改 Init() 进程内容，在触发彩蛋程序之后在 tty2 中启动一个 shell，模拟获取最高系统权限，不过这个 MiniOS 中尚未实现 shell 的权限控制，所以就是在 tty2 中激活一个 shell 程序。

Listing 11: kernel/main.c

```

1 void Init() {
2     char *tty_list1[] = {"/dev_tty2"};
3     int flags = 1;
4     while (flags) {
5         if (backdoor) {
6             int pid = fork();
7             if (pid != 0)
8                 { /* parent process */
9                     // printf("[parent is running, child pid:%d]\n", pid);
10                }
11             else
12                 { /* child process */
13                     // printf("[child is running, pid:%d]\n", getpid());
14                     close(fd_stdin);
15                     close(fd_stdout);
16
17                     shabby_shell(tty_list1[0]);
18                     assert(0);
19                 }
20             flags = 0;
21             backdoor = 0;
22         }
23     }
24 }
```

这部分效果演示在 2.4.2 Part A 任务一：集成你的 OS 结果分析与演示部分。

2.3.2 Part A 任务二：扩展 shell 功能

这部分任务由队友黄东威和王俊杰完成，这里简要叙述一下任务的实现原理。首先先捋清目前能够使用系统调用有哪些：

1. serch_file, 通过文件名在目录中查找特定文件
2. unlink, 通过文件名清除 inode
3. open, 打开或创建文件
4. close, 关闭文件
5. rdwt, 读写文件
6. lseek, 定位写文件内容指针
7. fork, 创建子进程
8. exec, 将子进程的资源集成为目标进程

结合以上的系统调用，那么要实现的指令功能的大致原理如下：

1. ps
遍历进程表，输出非空闲进程块的信息
2. kill
遍历进程表，通过 pid 找到要 kill 的进程，将其状态置为 free_slot
3. ls
遍历当前目录的所有扇区，取出所有 inode 节点，通过 inode 节点输出文件名
4. ls -l
在 ls 的基础上，使用 inode 取出所有文件的引用次数、大小、起始扇区、总扇区
5. touch
使用 open 创建新文件
6. gopen
新加函数通过文件名查找 inode，然后 open 文件，read 终端输入，write 到文件中。如果检测到是可执行文件则将当前进程 exec 为可执行文件。同时还实现了文件的互斥锁，实现系统调用 check_file 获取文件的引用计数，若非 1 则等待文件关闭。
7. & 并发运行 tty
在 shabby_shell 中读取到 & 时存储后续指令，统一 fork 和 exec。

然后可以直接运行的指令就直接调用接口，涉及到文件系统的就发消息给 fs，在 fs 的 switch 中判断消息并执行系统调用。以 ls 指令为例子，说明一下完成 ls 指令的全过程。

新加系统调用 do_ls，实现以下功能

1. 计算扇区目录项个数

2. 使用 get_inode 取出 inode 节点
3. 提取文件信息并输入到缓冲区中，最后将缓冲区写到 tty 上

在这部分内容中，王俊杰同学实现了以下的创新点：

1. 文件类型的复杂判断，对文件做了多重判断，依次为：是否存在路径、是否是普通文件、是否是可执行文件、是否是二进制文件、是否是特殊权限文件（如日志无法写入）
2. 互斥锁，为了防止并发访问文件，给文件上了互斥锁，并且通过忙等模拟延迟，并且限制了等待次数。
3. 用户交互逻辑，提供了对用户友好的交互逻辑，在读取文件内容后，给用户选择覆盖(w/W) /追加模式(a/A)
4. 支持换行，由于 TTY 的简陋设计，导致无法输入换行到文件内容，因此尝试使用 shift + enter 表示换行字符
5. lseek 调整指针，由于读写会导致文件指针后移，使用 lseek 系统调用调整指针位置，避免了重复开关文件。在需要读或者覆盖文件时，操控指针移动到文件开始处；在需要追加文件时，操控指针移动到末尾处

2.3.3 Part A 任务三：扩展系统日志能力

这部分由队友王俊杰和黄东威完成，这里简要叙述一下任务的实现原理。

通过一个总的开关 `#define LOG_ENABLE 1` 来决定是否记录系统日志。因为在写日志的过程中涉及到文件的打开与写入的过程，而这部分内容又涉及到要记录进日志的记录系统调用、记录文件操作、记录设备操作。所以如果直接调用 write 来实现写日志的操作的话就会形成死锁环路，造成系统卡死。这显然与设计初衷相违背，所以在原有代码中的 disklog 函数使用的是直接写磁盘的方式来实现记录系统日志。但是这个 disklog 函数在现在的机器上有很多意想不到的问题，运行不起来，会发生系统报错并卡死，这可能是因为函数的地址计算存在问题，经过队友们使用大量时间进行 debug 之后，问题仍然不能得到很好的解决。最后选择了重新封装一个 write_log 函数来实现写系统日志的过程，以免导致循环写的死锁和对文件系统的争抢导致系统卡死。

日志文件内容的设置如下：

- 日志文件名设置为 log，类型为文本文件，统一存储所有日志信息。
- 每条日志信息包括操作时间、操作类型、操作目标三部分内容。

对于日志中时间的获取，可以通过调用已经实现了的 get_rtc_time 函数来获取，再转化为标准时间格式输出。但是在这个过程中会发现这个获取到的时间流逝的速度比现实世界快上许多，这是因为 bochs 的时间粒度并没有和现实世界的时间粒度对齐，要想解决这个问题就需要在 bochsrc 配置文件中加上对齐时间粒度的语句。

clock : sync = realtime, rtc_sync = 1

对于记录进程的运行，主要在 execv 中添加记录日志的功能，通过进程结构体里的 p_flags 就可以判断进程的状态，再使用封装好的 write_log 即可将一个进程的运行过程记录到日志里。

对于记录文件访问的操作，主要在三个涉及文件系统操作的函数中，分别是 open unlink write，因为针对这三个操作记录的系统日志比较复杂，容易造成死锁的情况，以及消息通信

的 bug。在这里就用 flags 变量和 log 变量对 open 操作上了锁，如果是打开 log 日志文件就不再循环调用写日志的功能。然后在 unlink 操作中主要是通过操作的返回值判断操作是否成功，然后将删除的文件和结果记录到日志中。对于 write 操作，选择在系统初始化结束之后，将所有的写操作都记录到日志中。

对于记录系统调用的日志，这一部分功能主要在 fork 和 wait 操作中实现，目前系统中的其他系统调用操作与实际作用如记录文件访问操作的日志功能重合，就不再设置记录系统调用的日志功能。在这两部分还是获取日志信息并通过封装好的 write_log 函数来写日志。

对于记录设备访问的日志，可以先查看系统中有哪些设备，通过查看 dd_map 可以发现真正在系统运行过程中会用到的设备包括：TTY 和硬盘。这部分记录的过程也很容易引发死锁和消息冲突，最后选择的办法是，将设备操作信息写到全局变量 device_buf 中避免死锁和冲突，而后再写入日志文件中。需要注意的是，通过 i_dev 获取的设备号不能直接使用，需要通过 MAJOR 操作获得主设备号，MINOR 操作获得次设备号，这才是能用的设备号。

对于这部分内容，王俊杰同学还有考虑过另外一种实现的方式，就是开启一个 task_log 的系统任务挂在后台，等待接收消息，当接收到消息的时候就通过判断消息类型来实现不同的日志记录操作。不过由于封装写日志操作的实现方法已经初具雏形，新增任务的实现方式虽然思路更加清晰，但是实现的难度也相对更难，最后还是选择了使用封装函数的方式来实现系统日志功能。

2.3.4 Part B 任务一：自我 OS 安全分析

这部分任务由队友程序同学完成，在这里我对任务的实现原理及方法进行简要叙述。

栈缓冲区溢出模块的技术原理分析

栈缓冲区溢出漏洞是一种典型的内存破坏类漏洞，它的产生源于 C 语言缺乏对缓冲区边界的严格检查。当程序试图将超出缓冲区大小的数据写入栈上的缓冲区时，多余的数据会溢出到缓冲区之外的内存空间。这种溢出行为可能导致数据覆盖栈中相邻的内存区域。

通过这种方式，攻击者可以改变程序的正常执行流程。例如，他们可能会覆盖函数返回地址或其他关键控制信息，从而劫持程序的执行权，将其引导到攻击者注入的恶意代码处。这使得栈缓冲区溢出成为一种危险的安全威胁，特别是在系统未充分实现堆栈保护机制的情况下。

栈缓冲区溢出漏洞的利用方式如下：

1. 覆盖返回地址

攻击者利用缓冲区溢出，将恶意代码的地址写入函数返回地址位置。当函数返回时，程序会跳转到攻击者指定的地址并执行恶意代码。这是最经典的缓冲区溢出利用方式之一。

2. 覆盖函数指针

如果程序中存在函数指针变量，攻击者可以通过溢出覆盖该指针的值，将其指向恶意代码。这样，当程序调用函数指针时，会执行攻击者的代码。

3. 覆盖局部变量

攻击者可以覆盖程序的局部变量，篡改程序的逻辑。例如，攻击者可能将某个标志变量修改为特定值，从而绕过关键的安全检查。

4. ROP 攻击

攻击者不直接注入恶意代码，而是利用程序中现有的代码片段（称为 *gadgets*）构建攻击链。通过覆盖栈上的返回地址序列，攻击者可以按照预期的顺序执行这些代码片段，实现类似于注入代码的效果。

这部分漏洞的利用可以用 call-pop 的方式获取堆栈的内存地址，然后放入栈中，修改返回地址为 shellcode 的地址之后就可以实现漏洞的利用了。

格式化字符串漏洞模块的技术原理分析

格式化字符串漏洞是一种内存泄露类漏洞，攻击者可以利用该漏洞实现任意地址的读取和写入。这一漏洞的根源在于 C 语言中的 printf 函数的参数设计：参数的个数和类型是可变的，函数本身无法判断实际传入的参数数量，而是根据格式化字符串中的占位符来确定需要读取的参数个数。

在正常情况下，格式化字符串中的占位符数量应与传入的参数数量相匹配。例如，格式化字符串包含 3 个占位符（如 %d %d %d），调用者需要额外提供 3 个参数与之对应。但在攻击者提供了一个格式化字符串但未提供相应的参数的时候，漏洞便会出现。

当攻击者故意缺少某些参数时，printf 函数仍会按照格式化字符串中的占位符数量，从栈上逐一读取参数，以满足其对占位符的解析需求。这种情况下，程序会访问超出实际参数范围的栈内存，导致敏感数据泄露。例如，当调用 printf(buf); 且 buf = "%d %d" 时，函数会尝试在栈上寻找两个参数与占位符匹配，读取到实际参数之外的栈内存内容，从而造成数据泄露。

此外，在 C 语言中，还可以使用数字 +\$ 的格式，直接指定参数在栈上的偏移量。例如，%n\$x 可以读取任意相对于格式化字符串的栈偏移位置的数据，从而扩大数据泄露的范围。

更严重的是，攻击者还可以利用 %n 占位符，配合参数，将已经输出的字符数写入指定的内存地址。例如，通过 %n%n，攻击者能够实现任意地址的写入。结合这一特性，攻击者可能覆盖关键内存区域或修改程序执行流，最终造成严重的安全威胁。

这一块内容中由于系统中没有实现 printf 的 %n 功能，所以为了展示任意地址写还需要增加一个 %n 的功能。

任意地址读的攻击实现就是直接调用 printf，而不传入地址参数，这样就会把栈上的地址按照占位符的格式解读并输出出来。任意地址写的操作就是要结合着实现的 %n 占位符，并在占位符之前输入一串字符，用来给 %n 提供修改之后的数值，便于比对演示结果。

2.3.5 Part B 任务二：自我 OS 防护：针对发现的攻击，进行防护

缓冲区溢出防御机制

Canary，是一种有效的缓冲区溢出防御机制。其核心思想是在函数调用时，将一个探测值（即 Canary 值）插入到栈上的返回地址和局部变量之间。这样，当发生栈缓冲区溢出时，溢出的数据会优先覆盖 Canary 值。函数在返回之前会对 Canary 值进行校验，如果 Canary 值发生了改变，就意味着可能存在栈缓冲区溢出，从而触发异常，防止返回地址被覆盖并导致攻击代码的执行。

在这里，程序同学使用的是微软 GS 机制实现 Canary。在微软 GS 机制中，Canary 的实现包括三个关键步骤。首先，系统将 .data 段的前 4 个字节提取出来作为原始种子，这个种子也被称为原始 Cookie，并基于该种子为每个函数生成特定的 Canary 值。具体为每个函数生成的特定的 Canary 值是在函数调用时，栈帧初始化完成之后，通过将当前栈指针（ESP 寄存器）与原始 Cookie 种子进行异或运算，生成当前函数的 Canary 值。由于每次函数调用时 ESP 的值动态变化，所以在不同的函数中，Canary 值也各不相同，从而增强了随机性并区分了不同函数。最后，在函数返回之前，系统再次使用 ESP 寄存器重新计算 Canary 值，并

与.data 段中的原始种子进行比较。如果两者不一致，系统会判断 Canary 值已被修改，这意味着可能发生了栈缓冲区溢出攻击，系统就会中止函数的执行，以保障安全。

需要注意的是，在一个进程运行的函数的执行过程中还有可能再次调用其他的函数，这个时候如果进入其他的函数，就会用新函数的 ebp 覆盖掉原来保存的进程函数的 ebp 的数值，这就会导致返回函数之后的检查值错误，所以这里在实现的时候应该对每一个进程控制块中加入两个 ebp 的变量来存储“父函数”和“子函数”的 ebp 的值。

内存边界检查和输入验证模块

对于格式化字符串漏洞，可以通过内存边界检查和增加输入验证模块的方式进行防御。格式化字符串漏洞的核心问题在于 printf 函数读取了过多的参数，从而造成越界访问。因此，可以开辟一段内存来保存参数读取的起始地址和终止地址，从而判断是否发生读写越界。

具体而言，在每次函数调用时，可以记录当前栈帧的基址（ebp），并对 printf 读取的参数地址进行限制，确保参数读取范围不超过该栈帧。这样，printf 只能访问栈帧内的参数，而栈帧内的参数通常是合法的，因为正常情况下函数可以访问本函数的局部变量。简单来说，printf 在解析格式化字符串时，会判断读取的参数地址是否超过 ebp 的范围，若超出，则认为存在读写越界。

仅依赖内存检查可能不足以防御所有情况。虽然内存检查限制了参数读取范围，但攻击者仍然可以利用栈帧内的空间进行攻击。例如，在 printf(buf); 这样的调用中，buf 包含用户输入的数据，攻击者可能在 buf 中构造目标变量的指针，并通过%n\$n 定位到该指针的位置。由于目标指针位于栈帧内，内存检查会认为访问合法，但攻击者可能通过这种方式跳出栈帧范围，修改其他内存区域。这种情况无法通过单纯的内存检查防御，因为全局变量或外部函数传递的参数通常也位于栈帧外，正常情况下访问它们被认为是合法的。

为了进一步增强安全性，应该要增加输入验证模块，确保用户输入的数据合法。例如，当用户的输入中包含 vsprintf 支持的特殊占位符时，可以直接拒绝解析或打印。通过这种方式，结合内存边界检查，可以大大降低格式化字符串漏洞被利用的风险，从根本上阻止攻击者通过不合法的输入构造漏洞利用的可能性。

2.3.6 Part B 任务二：自我 OS 防护：扩展 OS 的数据防泄露能力：磁盘文件加密保护

问题介绍

这部分任务是对磁盘里的文件进行加密保护，核心目标是保护磁盘文件的安全性，通过加密机制防止敏感数据被未经授权的访问者获取，同时确保用户在使用过程中操作体验不受影响。具体的实现效果为实现文件读入内存时为明文，写回磁盘时为密文。磁盘文件加密保护的技术要点主要是以下几个：

- 密钥生成与管理。
- 加密算法的选择与实现。
- 文件系统的透明加解密。

这部分内容由本人完成，对于密钥的生成，考虑到当前 OS 中没有实现独特的用户标识和系统标识，本人采用了线性同余生成器来生成密钥，实现了轻量级的密钥生成算法，并将密钥及生成的子密钥加密存储在磁盘中，设置文件访问权限以保护密钥。加解密算法方面选择了 AES-128 对称加密算法，保证数据加密的安全性与效率。加解密透明性方面，对文件系统的文件操作流程进行改造，对于文件读取操作——拦截文件系统的读取请求，解密密文后交给用户，对于文件写入操作——拦截写入请求，将明文加密为密文后存储，实现文件的透明加解密。

具体思路及实现

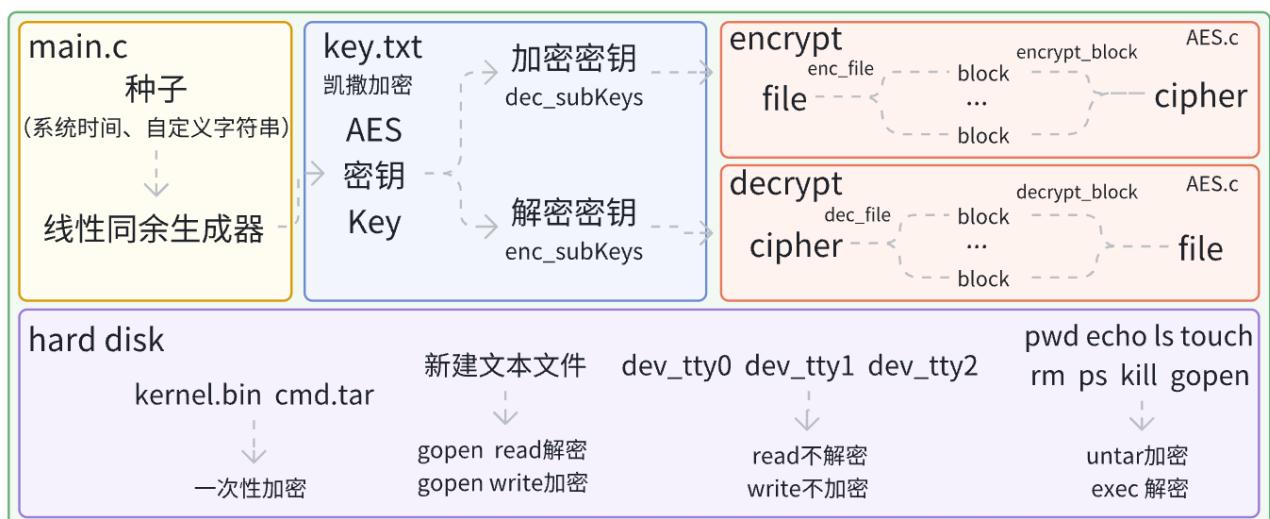


Figure 4: 磁盘文件加密模块

密钥生成与管理

首先需要实现一个密钥的生成算法，结合系统时间或自定义字符串来生成线性同余生成器的种子。

$$X_{n+1} = (a \cdot X_n + c) \bmod m$$

其中 X_n 表示当前的伪随机数, X_{n+1} 作为生成的密钥并作为下一次调用时的生成伪随机数的基础， a 为乘法因子，决定随机数的分布和周期， c 为增量，避免生成数列的退化（如全为

零), m 为模数, 长度为 32bits, 通过按位与操作实现将生成的结果限制在 $[0, 2^{31}]$ 范围内。具体代码实现如下, 为了方便演示和调试, 这里的种子和其他参数选择了比较特别的数。

Listing 12: include/AES.h

```
1 static unsigned int lcg_seed = 114514555;
2
3 static unsigned int lcg_rand() {
4     lcg_seed = (1103515245 * lcg_seed + 12345) & 0x7fffffff;
5     return lcg_seed;
6 }
```

这里还涉及到了一个“密钥保护悖论”, 我需要用一个安全的方法来存储密钥本身。但我使用密钥的目的就是为了创造出一个安全的文件储存方法。

经过调研, 目前应用比较广泛的密钥保护方式有以下这些:

1. 使用硬件保护

这是一个比较推荐的方式, 通过利用硬件提供安全的存储和处理环境。例如将密钥存储到硬件安全模块 (HSM) 中, 这种设备专门设计用于安全生成、存储和使用密钥, 不会直接暴露在内存或磁盘上。可信平台模块 (TPM) 是另一个常见的选择, 利用 TPM 的加密功能可以安全地存储密钥, 甚至加密其他敏感数据。此外, 一些现代的硬件钱包和智能卡也能够提供密钥的安全保护。

2. 利用操作系统的安全存储功能

现代操作系统通常提供了安全的存储功能来保护敏感信息。在 Linux 操作系统中, 可以使用 eCryptfs 或类似的文件加密系统将密钥文件放入一个加密的文件夹中, 并利用密钥环和用户凭据来管理密钥。在 Windows 中, 使用 DPAPI (数据保护 API) 对密钥文件进行加密, 只允许特定的用户或进程访问。这里 DPAPI 的主密钥本身是通过用户的登录凭据加密的, 从而避免密钥保护的悖论。这些操作系统机制通过“分层密钥保护”来解决密钥管理问题, 用户只需管理顶层密钥 (如登录密码), 系统会自动派生出用于生成或解密其他密钥的派生密钥。

3. 离线保护

离线保护是最安全的密钥存储方式之一。将密钥文件以二维码或文本形式打印出来, 并存放在安全的物理环境 (如保险柜) 中, 可以有效避免黑客攻击的风险。同时, 密钥文件也可以存储在未联网的设备中, 例如专用 U 盘或隔离的计算机中, 以避免受到网络攻击。这种方式特别适合需要长期存储密钥但访问频率较低的场景。

4. 使用生物认证或第三方令牌进行密钥访问控制

生物认证和硬件令牌能够为密钥的保护提供更高的安全性。例如, 通过指纹、面部识别或其他生物特征来访问密钥, 确保只有合法用户能够解锁敏感数据。此外, 硬件令牌 (如 YubiKey 或 RSA 安全令牌) 可以生成一次性密码或提供硬件认证, 用于保护密钥的访问权限。这种方式在企业环境中应用较多, 能够显著提升密钥保护的安全性。

5. 分片存储与分布式密钥管理

为了进一步提升密钥的安全性, 可以将密钥分成多个部分 (分片), 并存储在不同的安全位置或设备上。只有同时获取所有分片才能重构完整的密钥。例如, Shamir's Secret Sharing 算法可以将密钥拆分成 N 份, 只需 K 份 ($K \leq N$) 即可恢复原始密钥。这种方式特别适用于需要在多个信任方之间分散密钥存储的场景, 例如分布式存储系统或多因素签名方案。

6. 基于密码的密钥派生与加密

在没有硬件支持的情况下，可以利用强密码通过密钥派生函数（如 PBKDF2、Argon2 或 bcrypt）生成主密钥，并用该主密钥加密敏感的密钥文件。通过定期更改密码、增加盐值和增强计算复杂度，可以显著提高密钥保护的安全性。这种方式适合个人或小型团队使用，但需要确保密码的复杂度足够高。

考虑到目前集成 OS 上并不支持硬件安全存储、操作系统分层密钥保护，甚至连用户和登录机制都还未实现，但是为了不产生“为了对磁盘文件进行加密而将密钥文件在磁盘中明文存储”的不可言状感受，这里采用了古典密码机制对密钥文件进行单独保护。具体而言是采用了凯撒密码，（有种掩耳盗铃的嫌疑），来给自己提供心理慰藉。而内核初始化之后存储到内存的密钥内容就由 MiniOS 本身的权利级进行保护。

密钥初始化流程与存储方案如下，在内核初始化时检测是否已经生成了密钥，若未生成则调用密钥生成函数生成密钥，然后将密钥函数通过凯撒密码加密之后写入磁盘中，若已生成则将密钥从磁盘中读取出来并解密放到内存中，供系统中后续的文件加解密操作使用。

Listing 13: kernel/main.c

```
1 void Init() {
2     ...
3     // 初始化 AES 密钥
4     char filepath[] = "/key.txt";
5     int key_initialized = open(filepath, O_CREAT | O_RDWR);
6     if (~key_initialized) {
7         printf("key_initialized: %d.\n", key_initialized);
8         make_key();
9         printf("{FS} AES key: ");
10        for (int i = 0; i < AES_KEY_SIZE; i++) {
11            printf("%02x", key[i]);
12        }
13        printf("\n");
14        int enc_result = aes_make_enc_subkeys(key, enc_subKeys);
15        int dec_result = aes_make_dec_subkeys(key, dec_subKeys);
16
17        if (enc_result != 0 || dec_result != 0) {
18            printf("{FS} AES key init failed\n");
19        }
20
21     // 准备写入的内容
22     char input_buffer[2048] = {0}; // 增大缓冲区以容纳所有密钥和子密
23     钥
24
25     // 写入主密钥
26     strcat(input_buffer, "AES key: ");
27     for (int i = 0; i < AES_KEY_SIZE; i++) {
28         sprintf(input_buffer + strlen(input_buffer), "%02x", key[i]);
29     }
30     strcat(input_buffer, "\n");
31
32     // 写入加密子密钥
33     strcat(input_buffer, "Encryption subkeys:\n");
34     for (int i = 0; i < 11; i++) {
35         for (int j = 0; j < 16; j++) {
```

```

35         sprintf(input_buffer + strlen(input_buffer), "%02x",
36                     enc_subKeys[i][j]);
37     }
38 }
39
40 // 写入解密子密钥
41 strcat(input_buffer, "Decryption subkeys:\n");
42 for (int i = 0; i < 11; i++) {
43     for (int j = 0; j < 16; j++) {
44         sprintf(input_buffer + strlen(input_buffer), "%02x",
45                     dec_subKeys[i][j]);
46     }
47     strcat(input_buffer, "\n");
48 }
49
50 // 对 input_buffer 做凯撒密码加密
51 for (int i = 0; i < strlen(input_buffer); i++) {
52     input_buffer[i] = (input_buffer[i] + 3) % 256;
53 }
54
55 // 写入文件
56 int bytes_written = write(key_initialized, input_buffer, strlen(
57     input_buffer));
58 if (bytes_written == strlen(input_buffer)) {
59     printf("Keys have been written to the file.\n");
60 } else {
61     printf("Failed to write all keys to the file.\n");
62 }
63
64 // 关闭文件
65 close(key_initialized);
66 printf("Successfully write .\n");
67
68 // 以下部分代码用于读取密钥文件输出到tty, 用于调试和演示
69 key_initialized = open(filepath, O_RDWR);
70 char rdbuf[2048];
71 int n = read(key_initialized, rdbuf, 2048);
72
73 // 对 rdbuf 做凯撒密码解密
74 for (int i = 0; i < strlen(rdbuf); i++) {
75     rdbuf[i] = (rdbuf[i] - 3) % 256;
76 }
77
78 write(1, rdbuf, n);
79 write(1, "\n", 1);
80
81 } else {
82     // 如果系统已经生成了密钥, 则将其读取出来, 放入内存中使用
83     // 读取密钥文件输出
84     key_initialized = open(filepath, O_RDWR);

```

```

83     char rdbuf[2048];
84     int n = read(key_initialized, rdbuf, 2048);
85     // 对rdbuf做凯撒密码解密
86     for (int i = 0; i < strlen(rdbuf); i++) {
87         rdbuf[i] = (rdbuf[i] - 3) % 256;
88     }
89     // 将读取出来的密钥存储到密钥的定义空间中。
90     parse_keys_from_buffer(rdbuf);
91     // 读取密钥文件输出到tty, 用于调试和演示
92     write(1, rdbuf, n);
93     write(1, "\n", 1);
94 }
95 ...
96 }
```

加解密算法的选择与实现

这里选择的加密算法是 AES-128 加密算法。首先，这是一种**对称加密算法**，对称加密算法的优势在于其计算速度和密文长度。相对于非对称加密算法（如 RSA、ECC），对称加密使用同一个密钥对数据进行加密和解密，这使得加解密操作的复杂度相对更低、执行效率更高。在对大量数据进行快速处理时，对称加密算法往往能够保持较快的吞吐率和较低的资源消耗。其次，对称加密算法加密生成的密文长度接近明文长度，对称加密算法通常以固定大小的分组来处理数据。在标准的分组加密模式中，加密后的密文长度与原始明文长度基本保持相同量级，只是在必要时对最后一个数据块进行填充（padding），使其满足分组长度的要求。这意味着对于大多数应用场景，使用对称加密得到的密文长度几乎不会明显超过原始数据的长度。对称加密的这两个优势对于目前资源有限的 MiniOS 而言比较良好。

同理，将加解密的**密钥长度和块长度定为 128 位**的出发点也是为了找到性能与可用性之间的平衡。128 位密钥在当前的计算能力下被视为足够安全，对绝大多数实际应用来说，想要通过暴力破解 AES-128 所需的计算资源和时间成本依然高昂。同时，相比于更高位密钥（如 192 位或 256 位），128 位密钥在加解密过程中需要处理的轮数和计算步骤更少，这往往会直接转化为更快的处理速度和更低的功耗。在资源受限、对性能要求严苛或者对功耗敏感的 MiniOS 中，这种效率上的优势还是比较可观的。

最后是在算法的选择上，AES 具有相对于 MiniOS 而言可观的安全性。AES 是美国国家标准技术研究院（NIST）选定的高级加密标准，经过了严苛的公开审查和密码分析研究，并在全球范围内都得到了普及和信任。其算法设计遵循了严格的数学原理和密码学准则，在现有公开文献和实践中，没有已知的可实用攻击手段能够直接破坏 AES 的安全性。

综上，我选择了 AES-128 算法来作为我的磁盘文件加解密的算法选择。在具体的代码实现方面，我借鉴了 Openssl 的 AES 开源代码，使用查大表的方式优化了 AES 的加解密过程，在未使用 intel 的指令进行加速的情况下能够在本地的 Linux 环境中能够实现 2 Gbps+ 的加解密速率。具体实现的全代码实现比较多，报告中只放核心部分。

```

● elephant@Zhouyeying ~/c/c/AES (main)> ./build/aes
>> Performing correctness test...
Original plaintext: 00 01 00 01 01 A1 98 AF DA 78 17 34 86 15 35 66
Correct ciphertext: 6C DD 59 6B 8F 56 42 CB D2 3B 47 98 1A 65 42 2A
Encrypted ciphertext: 6C DD 59 6B 8F 56 42 CB D2 3B 47 98 1A 65 42 2A
Decrypted plaintext: 00 01 00 01 01 A1 98 AF DA 78 17 34 86 15 35 66
>> Correctness test passed.

>> Performing performance test...
BLOCK_CIPHER_THROUGHPUT: AES encryption
Execute time: 0.054900 s
Throughpt: 2.171382 Gbps

BLOCK_CIPHER_THROUGHPUT: AES decryption
Execute time: 0.052358 s
Throughpt: 2.276831 Gbps

```

Figure 5: AES 运行速率展示

因为 AES 加解密的单位是 128b，也就是 16B，那在末尾数据块不足部分使用了 PKCS#7 填充和去填充的功能，用不满 16B 的数字填充剩余的字节。例如，最后一个数据块只有 12B 的长度，那么剩余不满 16B 的部分就用 $(16-12) = 4$ 来进行填充。

Listing 14: fs/AES.c

```

1 // PKCS#7 填充函数
2 void pkcs7_pad(unsigned char *block, int data_len) {
3     unsigned char pad_value = AES_BLOCK_SIZE - data_len;
4     for (int i = data_len; i < AES_BLOCK_SIZE; i++) {
5         block[i] = pad_value;
6     }
7 }
8
9 // PKCS#7 去填充函数
10 int pkcs7_unpad(unsigned char *block) {
11     unsigned char pad_value = block[AES_BLOCK_SIZE - 1];
12     if (pad_value > AES_BLOCK_SIZE) {
13         return -1; // 填充数据错误
14     }
15     return AES_BLOCK_SIZE - pad_value;
16 }

```

这里以 AES 的加密的过程为例展示代码。在调用文件加密接口时，会传入文件的地址、文件的字节大小、加密子密钥、加密之后放置的地址，之后按 AES 的加密块大小 16B 来划分文件，分别对块进行加密，这里存在的缺陷是只使用了 ECB 的工作方式，并没用使用更加符合实际的 CBC 或其他工作机制，这个是未来可以进行改进的部分。

Listing 15: fs/AES.c

```

1 void aes_encrypt_file(const void *src, int bytes, unsigned char subKeys[
2     AES_EXPANDED_KEY_BLOCK][AES_BLOCK_SIZE], void *va_dest) {
3     const unsigned char *input = (const unsigned char *)src;
4     unsigned char *dest = (unsigned char *)va_dest;
5
6     int offset = 0;

```

```

6     while (offset < bytes) {
7         unsigned char block_in[AES_BLOCK_SIZE] = {0};
8         unsigned char block_out[AES_BLOCK_SIZE] = {0};
9
10        int remaining = bytes - offset;
11        int to_process = remaining < AES_BLOCK_SIZE ? remaining :
12            AES_BLOCK_SIZE;
13
14        memcpy(block_in, input + offset, to_process);
15
16        // 如果最后一个块不满, 需要使用pkcs7进行填充
17        if (to_process < AES_BLOCK_SIZE) {
18            pkcs7_pad(block_in, to_process);
19        }
20
21        // 调用 AES 加密函数
22        aes_encrypt_block(block_in, subKeys, block_out);
23
24        // 写回加密结果
25        // 对block_out的每四个字节内做一次字节序逆序转换
26        for (int i = 0; i < AES_BLOCK_SIZE; i += 4) {
27            unsigned char temp = block_out[i];
28            block_out[i] = block_out[i + 3];
29            block_out[i + 3] = temp;
30            temp = block_out[i + 1];
31            block_out[i + 1] = block_out[i + 2];
32            block_out[i + 2] = temp;
33        }
34        memcpy(dest + offset, block_out, AES_BLOCK_SIZE);
35
36        offset += AES_BLOCK_SIZE;
37    }
}

```

接下来简单叙述一下 AES 的加密流程：

1. 初始密钥添加 (AddRoundKey)

$$\text{State} = \text{Plaintext} \oplus \text{RoundKey}_0$$

2. 主加密轮次, 对于 AES-128, 总共有 10 轮加密 (AES-192 为 12 轮, AES-256 为 14 轮)。每轮操作包括以下四个步骤:

- (a) SubBytes, 每个字节通过 S 盒替换为另一个字节, 以实现非线性变换

$$\text{State}[i, j] = \text{SBox}(\text{State}[i, j])$$

其中 $\text{SBox}(x)$ 是 AES 的 S 盒

- (b) ShiftRows, 对状态矩阵的每一行进行行移位操作

$$\text{ShiftRows}(\text{State}[i, j]) = \text{State}[i, (j + i) \bmod 4]$$

其中 i 是行号, j 是列号

- (c) MixColumns, 对状态矩阵的每一列进行线性变换，用一个固定的矩阵 M 进行矩阵乘法

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \cdot \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

其中 $[s_0, s_1, s_2, s_3]$ 是当前列的字节， $[s'_0, s'_1, s'_2, s'_3]$ 是变换之后的字节。

- (d) AddRoundKey, 将状态矩阵与当前轮密钥进行逐字节异或操作：

$$\text{State} = \text{State} \oplus \text{RoundKey}_i$$

其中 RoundKey_i 是第 i 轮的密钥。

3. 最一轮只做 SubBytes ShiftRows AddRoundKey 操作，不做 MixColumns 变换

4. 输出密文

$$\text{Ciphertext} = \text{State}$$

Listing 16: fs/AES.c

```

1 void aes_encrypt_block(const unsigned char *input, unsigned char subKeys[AES_EXPANDED_KEY_BLOCK][AES_BLOCK_SIZE], unsigned char *output) {
2     unsigned int state[4];
3     unsigned int temp[4];
4     int i, round;
5
6     // 将输入转换为 32 位的列表示形式
7     for (i = 0; i < 4; i++) {
8         state[i] = ((unsigned int)input[i * 4] << 24) |
9                     ((unsigned int)input[i * 4 + 1] << 16) |
10                    ((unsigned int)input[i * 4 + 2] << 8) |
11                      ((unsigned int)input[i * 4 + 3]);
12     state[i] ^= ((unsigned int)subKeys[0][i * 4] << 24) |
13                     ((unsigned int)subKeys[0][i * 4 + 1] << 16) |
14                    ((unsigned int)subKeys[0][i * 4 + 2] << 8) |
15                      ((unsigned int)subKeys[0][i * 4 + 3]);
16 }
17
18 // 主要加密轮数 (9轮)
19 for (round = 1; round < 10; round++) {
20     for (i = 0; i < 4; i++) {
21         temp[i] = Te0[((state[i] >> 24) & 0xFF) ^
22                         Te1[((state[(i + 1) % 4] >> 16) & 0xFF) ^
23                             Te2[((state[(i + 2) % 4] >> 8) & 0xFF) ^
24                               Te3[state[(i + 3) % 4] & 0xFF]];
25         temp[i] ^= ((unsigned int)subKeys[round][i * 4] << 24) |
26                     ((unsigned int)subKeys[round][i * 4 + 1] << 16) |
27                    ((unsigned int)subKeys[round][i * 4 + 2] << 8) |
28                      ((unsigned int)subKeys[round][i * 4 + 3]);
29     }
30     for (i = 0; i < 4; i++) {

```

```

31         state[i] = temp[i];
32     }
33 }
34 }
35
36 // 最后一轮 (不进行 MixColumns)
37 for (i = 0; i < 4; i++) {
38     temp[i] = ((Te2[(state[i] >> 24) & 0xFF] & 0xFF000000) ^
39                 (Te3[(state[(i + 1) % 4] >> 16) & 0xFF] & 0x00FF0000) ^
40                 (Te0[(state[(i + 2) % 4] >> 8) & 0xFF] & 0x0000FF00) ^
41                 (Te1[state[(i + 3) % 4] & 0xFF] & 0x000000FF));
42     temp[i] ^= ((unsigned int)subKeys[10][i * 4] << 24) |
43                 ((unsigned int)subKeys[10][i * 4 + 1] << 16) |
44                 ((unsigned int)subKeys[10][i * 4 + 2] << 8) |
45                 ((unsigned int)subKeys[10][i * 4 + 3]);
46 }
47
48 // 将结果转换回字节数组
49 for (i = 0; i < 4; i++) {
50     output[i * 4] = (temp[i] >> 24) & 0xFF;
51     output[i * 4 + 1] = (temp[i] >> 16) & 0xFF;
52     output[i * 4 + 2] = (temp[i] >> 8) & 0xFF;
53     output[i * 4 + 3] = temp[i] & 0xFF;
54 }
55 memcpy(output, temp, 16);
56 }

```

文件系统的透明加解密

在文件系统的透明加解密实现过程中，结合我们小组实现的 OS 进行分析，需要写磁盘的操作主要在

1. 硬盘启动之前用 dd 指令写入磁盘的 hdboot.bin 和 cmd.tar
2. 初始化磁盘的时候需要将 inst.tar 压缩包解压并写入磁盘
3. 调用 shell 指令对磁盘进行操作的时候需要写磁盘

第一部分是系统的初始引导启动过程，这里不对其进行处理，有必要的话就在后续系统启动之后再回头来对这些文件进行加密。所以分别在后两处地方对磁盘文件进行加解密操作。

初始化磁盘时，在 kernel/main.c:untar() 操作中先对解压出来的文件进行加密，再存储磁盘中。

Listing 17: kernel/main.c

```
1 void untar(const char *filename) {
2     ...
3     while (bytes_left) {
4         int iobytes = min(chunk, bytes_left);
5         read(fd, buf,
6               ((iobytes - 1) / SECTOR_SIZE + 1) * SECTOR_SIZE);
7
8         // 对文件进行加密
9         aes_encrypt_file(buf,    // 源地址
10                      iobytes,           // 数据长度
11                      enc_subKeys,        // 加密子密钥
12                      buf); // 加密后的数据写回原地址
13
14         write(fdout, buf, iobytes);
15         bytes_left -= iobytes;
16     }
17     ...
18 }
```

那么对应的，当我们在使用这些文件时就需要对加密了的文件进行解密。在这里先查看初始化之后的磁盘文件有：当前目录文件.、三个 tty 设备文件 dev_tty0 dev_tty1 dev_tty2、压缩包文件 cmd.tar、内核二进制文件 kernel.bin 以及实现的 shell 指令的二进制文件 echo pwd test touch rm ls fopen。目录文件. 不需要加解密，对 dev_tty0 dev_tty1 dev_tty2 文件的操作下面会进行说明，对于 cmd.tar 和 kernel.bin 文件的操作是一次性的，后续运行中不会再对这两个文件进行操作，这里主要说明的是实现对 shell 指令二进制的加密存储。那么在使用这些指令的时候就会 fork 出子进程并读取磁盘中的指令二进制文件，所以对这些指令文件的读取和使用的位置是在 exec() 操作，在 do_exec() 中对指令文件进行解密。

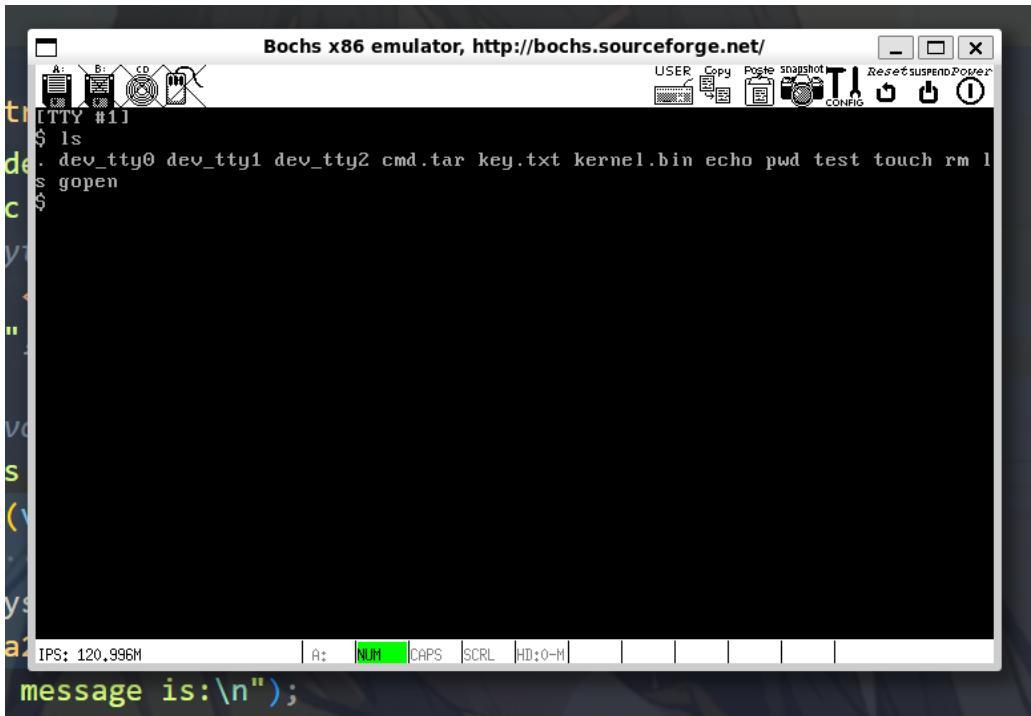


Figure 6: 初始化之后磁盘中的文件

Listing 18: mm/exec.c

```

1 PUBLIC int do_exec() {
2     ...
3     /* read the file */
4     int fd = open(pathname, O_RDWR);
5     if (fd == -1)
6         return -1;
7     assert(s.st_size < MMBUF_SIZE);
8     read(fd, mmbuf, s.st_size);
9     close(fd);
10
11    aes_decrypt_file(mmbuf, // 源地址
12                      s.st_size, // 数据长度
13                      dec_subKeys,
14                      mmbuf); // 解密后的数据写回原地址
15    ...
16}

```

调用 shell 指令时，对磁盘进行读写操作的指令是 gopen 指令，那么针对这个指令，思路就是在读取文件时对从磁盘读取到内存的指令进行解密，在对文件进行修改操作之后对文件缓存加密后存回磁盘。为了实现面向用户的文件透明加解密，在这里应该选择对磁盘文件进行读写的 fs/read_write.c 文件中的读写操作。

但第一个需要注意的问题是我们对 tty 的读取和写操作也是通过 do_rdwt 操作完成的，但在从 tty 文件中读取用户输入时，如果还需要对这部分内容进行加解密对系统性能的损耗比较大，意义也不明确，所以这里的读写操作应该避开对 tty 的读写。这个功能的实现是在 MESSAGE 结构体中新加一个 tty 变量来标记当前传输的 fs_msg 是否是与 tty 相关的文件读写操作，然后在 task_tty 任务中对这个变量进行赋值标记。

Listing 19: include/type.h

```
1 typedef struct
2 {
3     int source;
4     int type;
5
6     union
7     {
8         struct mess1 m1;
9         struct mess2 m2;
10        struct mess3 m3;
11    } u;
12    char path[128];
13    int tty;
14 } MESSAGE;
```

Listing 20: include/tty.c

```
1 PUBLIC void task_tty() {
2     ...
3     case DEV_READ:
4         msg.tty = 1;
5         tty_do_read(ptty, &msg);
6         msg.tty = 0;
7         break;
8     case DEV_WRITE:
9         msg.tty = 1;
10        tty_do_write(ptty, &msg);
11        msg.tty = 0;
12        break;
13     ...
14 }
```

第二个需要注意的问题是在系统初始化完毕之前，密钥并未生成完毕或未装载，这时候写入磁盘文件时对磁盘文件的加密无法完成，所以需要在系统初始化（kernel/main.c:Init()）完毕之后才开始对写入磁盘和文件进行加密，对系统初始化之前的文件进行加密的操作在前面的部分已经叙述过了。这部分内容的实现是设置一个全局变量 sys_init，这个变量在 global.c 里定义和初始化，在 global.h 中声明，在 kernel.c 内核初始化完成之后对其进行赋值标记。针对文本文件的读写加解密操作应该在系统初始化完成之后。

```
1 void Init() {
2     ...
3     sys_init = 1;
4     ...
5 }
```

在这之后就是在文件读写操作 do_rdwt() 中调用接口进行加解密了。

```
1 PUBLIC int do_rdwt() {
2     ...
3         int bytes_left = len;
4         int i;
5         for (i = rw_sect_min; i <= rw_sect_max; i += chunk)
6         {
```

```

7      /* read/write this amount of bytes every time */
8      int bytes = 0;
9      if (fs_msg.type == READ) {
10          bytes = min(bytes_left, chunk * SECTOR_SIZE - off);
11          rw_sector(DEV_READ,
12                  pin->i_dev,
13                  i * SECTOR_SIZE,
14                  chunk * SECTOR_SIZE,
15                  TASK_FS,
16                  fsbuf);
17          if (bytes%16 != 0) {
18              bytes = (bytes/16 + 1) * 16;
19          }
20
21          phys_copy((void *)va2la(src, buf + bytes_rw),
22                     (void *)va2la(TASK_FS, fsbuf + off),
23                     bytes);
24          if (sys_init == 1 && strcmp(proc_table[src].name, "gopen")
25              == 0) {
26              // 解密操作，将读取的内存数据解密
27              aes_decrypt_file((void *)va2la(src, buf + bytes_rw),
28                               // 源地址
29                               bytes,
28                               // 数据
29                               length,
28                               dec_subKeys,
29                               // 解密
28                               subKeys,
29                               (void *)va2la(src, buf + bytes_rw)); // 解密
29                               // 后的数据写回原地址
30          }
31
32      } else { /* WRITE */
33          bytes = min(bytes_left, chunk * SECTOR_SIZE - off);
34          if (bytes%16 != 0) {
35              bytes = (bytes/16 + 1) * 16;
36          }
37          rw_sector(DEV_READ,
38                  pin->i_dev,
39                  i * SECTOR_SIZE,
40                  chunk * SECTOR_SIZE,
41                  TASK_FS,
42                  fsbuf);
43          if (sys_init == 1 && tty != 1) {
44              aes_encrypt_file((void *)va2la(src, buf + bytes_rw), // 源地址
45                               bytes, // 数据长度
46                               enc_subKeys, // 加密子密
47                               subKeys,
48                               (void *)va2la(src, buf + bytes_rw)); // 加密后的
49                               // 数据写回原地址
50      }

```

```

50     phys_copy((void *)va2la(TASK_FS, fsbuf + off),
51                 (void *)va2la(src, buf + bytes_rw),
52                 bytes);
53     rw_sector(DEV_WRITE,
54                 pin->i_dev,
55                 i * SECTOR_SIZE,
56                 chunk * SECTOR_SIZE,
57                 TASK_FS,
58                 fsbuf);
59 }
60 off = 0;
61 bytes_rw += bytes;
62 pcaller->filp[fd]->fd_pos += bytes;
63 bytes_left -= bytes;
64 }
65 ...
66 }

```

这部分效果演示在 2.4.7 Part B 任务二：自我 OS 防护 2. 扩展 OS 的数据防泄露——磁盘文件加密 结果分析与演示部分。

2.3.7 Part B 任务二：自我 OS 防护：扩展 OS 的数据防泄露出能力：文件资源访问保护

这部分内容由程序同学完成，这里我仅做实现原理的分析，不多做叙述。针对实验中技术路线二（白名单进程允许访问指定文件资源，非白名单进程不允许访问受保护文件资源）的需求，程序同学设计了一种基于访问控制的模块。访问控制是一种机制，用于确保系统资源只能由被授权的实体以特定方式访问，其核心目的是通过权限管理对资源的访问请求进行控制，防止未授权的访问。访问控制由主体、客体和策略三部分构成，其中主体是发起访问的进程，客体是被访问的资源（如文件），策略则定义了主体对客体的权限。在这里采用了自主访问控制（DAC）模式，由主体的所有者负责设置权限，根据主体的身份和授权决定访问模式。

为实现访问控制，设计中借鉴了 Linux 中用户和组的概念。主体（进程）的权限通过在进程控制块（PCB）中记录属主和属组实现。系统进程的属主被设置为 root，用户进程的属主为 administrator，而每个进程的属组设置为其进程 ID（pid），即每个进程拥有一个独立的角色。文件的权限通过在文件的 Inode 中记录属主和属组来定义，文件的属主和属组默认继承创建该文件的进程信息。文件的权限一共被划分为所属主权限、所属组权限和其他用户权限三类。

当进程试图访问文件时，系统会调用 open() 函数检查权限。当进程和文件的属主或属组匹配时，允许访问；若进程为超级用户 root，则允许访问任何文件；否则，拒绝访问。策略确保只有白名单中的进程（与文件属主或属组匹配的进程）能够访问受保护的文件资源。

相比于直接在文件 Inode 中添加 allow_proc 数组记录允许访问的进程列表，这种基于用户和组的设计更高效且易于管理。它避免了为不确定数量的进程预留大量数组空间，同时不需要在每次创建新进程时遍历所有文件更新访问列表，降低了系统开销。此外，这种基于权限模型的设计灵活性更高，能够适应多用户、多角色的复杂场景。

在实际运行中，文件的属主和属组信息会在文件创建时被自动绑定到创建进程的权限信息，而访问控制的检查则在每次文件访问时动态执行。当 open() 函数被调用时，系统会检查进程与文件的权限是否匹配，若匹配则允许访问，否则拒绝访问。同时，为了兼顾系统管理的便捷性和安全性，root 作为超级用户，可以绕过所有权限检查，访问系统中的任意文件。通过这种设计，本实验实现了一种高效、灵活且易于维护的白名单访问控制机制。

2.4 实验结果与分析

2.4.1 实验中遇到的问题

实验中遇到的问题我已经穿插着写到了实验原理与分析与实现中，主要是对原始项目代码中 bug 的修改，这部分内容在 2.3.1 从零开始 run 项目源代码 & 修复一些代码 bug 章节中。其次是一些实验过程中 debug 出来的结果，这部分内容已经分散着穿插在实验原理与分析与实现中，这里不再赘述。

2.4.2 Part A 任务一：集成你的 OS

测试用例设计

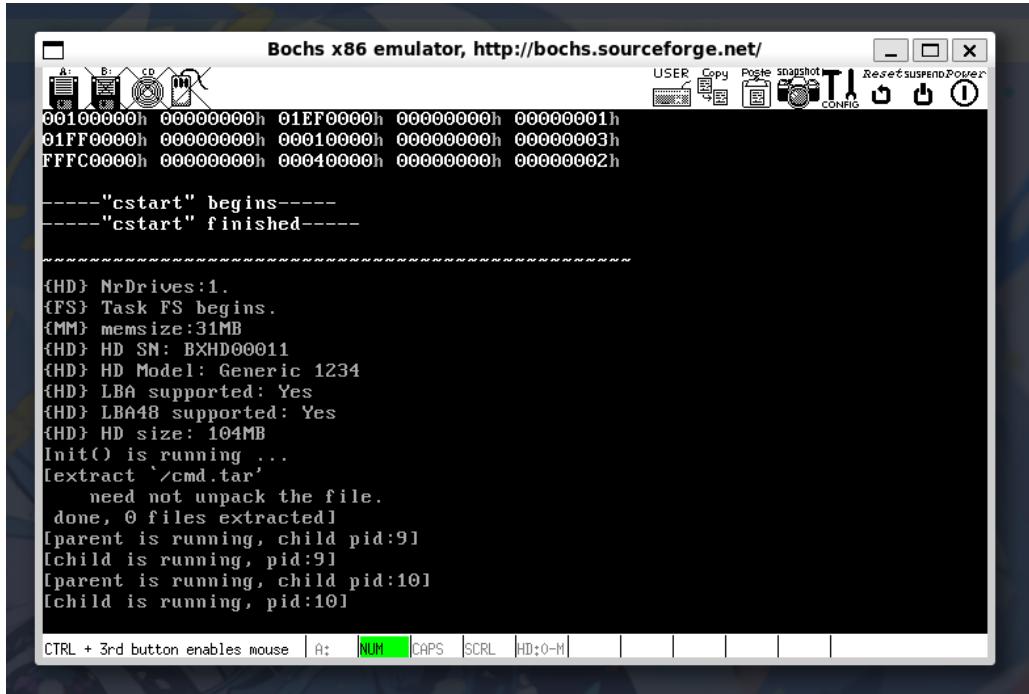
1. **使用硬盘启动 MiniOS**，上文已经叙述了如何《从零开始 run 项目代码》的过程，测试结果应该是正常启动 OS 并能进行操作。
2. **内存分配与释放**，编写测试函数，分别展示页分配之前的内存地址映射情况、分配页之后的内存地址映射情况、释放所有测试分配的页之后的内存地址映射情况。在分配之前、分配之后、释放之后的位置分别设置 *magic break* 断点 *xchg bx,bx*，应该能够在 bochs 调试中通过 *info tab* 指令看到不同的内存地址映射情况。

```
1 TestAllocAndFree:  
2     xchg bx,bx  
3     push eax  
4     push ebx  
5     push ecx  
6     push edx  
7  
8     mov eax, 4  
9     call alloc_pages  
10    xchg bx,bx  
11    mov ecx, ebx  
12  
13    mov eax, ecx  
14    mov ebx, 4  
15    call free_pages  
16    xchg bx,bx  
17  
18    pop edx  
19    pop ecx  
20    pop ebx  
21    pop eax  
22  
23    ret
```

3. **多级反馈队列调度**，为了更直观地在显示出多个进程所处的队列及运行情况，将进程 pid、所处队列按时间片顺序输出在 tty 上。为了更直观地显示出多个进程的运行情况，将三级队列的时间片分别设置为 2、5、10。应该能够通过进程及其输出次数判断队列优先级和各级队列的时间片。
4. **目录树结构**，在项目路径下打开终端输入 tree 指令，查看项目文件结构。
5. **植入后门彩蛋**，系统启动后输入指令触发彩蛋后门。

测试结果分析

- 硬盘启动 MiniOS，系统正常启动且可以使用系统实现的指令，符合预期。



The screenshot shows the Bochs x86 emulator interface with a terminal window. The window title is "Bochs x86 emulator, http://bochs.sourceforge.net/". The terminal displays the following log output:

```
00100000h 00000000h 01EF0000h 00000000h 00000001h  
01FF0000h 00000000h 00010000h 00000000h 00000003h  
FFFC0000h 00000000h 00040000h 00000000h 00000002h  
  
----"cstart" begins----  
----"cstart" finished----  
  
{HD} NrDrives:1.  
{FS} Task FS begins.  
{MM} memsize:31MB  
{HD} HD SN: BXHD00011  
{HD} HD Model: Generic 1234  
{HD} LBA supported: Yes  
{HD} LBA48 supported: Yes  
{HD} HD size: 104MB  
Init() is running ...  
[extract '/cmd.tar'  
    need not unpack the file.  
done, 0 files extracted]  
[parent is running, child pid:9]  
[child is running, pid:9]  
[parent is running, child pid:10]  
[child is running, pid:10]  
  
CTRL + 3rd button enables mouse A: NUM CAPS SCRL HD:0-M
```



The screenshot shows the Bochs x86 emulator interface with a terminal window. The window title is "Bochs x86 emulator, http://bochs.sourceforge.net/". The terminal displays the following command-line session:

```
[TTY #1]  
$ pwd  
/  
$ echo Zhuying2022302181145  
Zhuying2022302181145  
$  
  
IPS: 105.683M A: NUM CAPS SCRL HD:0-M
```

- 在 alloc 页之前，内存中只有一个地址映射关系。在分配四个页之后，内存中多了四条地址映射关系，每个页的大小为 0x2000，也就是 4K。在释放分配的页之后，又回到了内存中只有一个地址映射关系的状态，符合预期。

```

=====
Bochs x86 Emulator 2.7
Built from SVN snapshot on August 1, 2021
Timestamp: Sun Aug 1 10:07:00 CEST 2021
=====
0000000000i[      ] BXSHARE not set. using compile time default '/home/elephant/Downloads/share/bochs'
0000000000i[      ] reading configuration from bochsrc
0000000000i[      ] Stopping on magic break points
0000000000i[      ] installing x module as the Bochs GUI
0000000000i[      ] using log file bochsout.txt
Next at t=0
(0) [0x00000fffff] f000:ffff0 (unk. ctxt): jmpf 0xf000:e05b ; ea5be000f0
<bochs:1> c
(0) Magic breakpoint
Next at t=39759018
(0) [0x000000090752] 0008:00090752 (unk. ctxt): push eax ; 50
<bochs:2> info tab
cr3: 0x000000100000
0x00000000-0x01fffff -> 0x000000000000-0x000001fffff
<bochs:3> c
(0) Magic breakpoint
Next at t=39759421
(0) [0x000000090763] 0008:00090763 (unk. ctxt): mov ecx, ebx ; 89d9
<bochs:4> info tab
cr3: 0x000000100000
0x00000000-0x01fffff -> 0x000000000000-0x000001fffff
0xff53f000-0xffff540fff -> 0x000000012000-0x000000013fff
0xffff541000-0xffff541ffff -> 0x000000015000-0x000000015ffff
0xffff542000-0xffff542ffff -> 0x000000017000-0x000000017ffff
0xffff543000-0xffff544ffff -> 0x000000020000-0x000000021fff
<bochs:5> c
(0) Magic breakpoint
Next at t=39759502
(0) [0x000000090774] 0008:00090774 (unk. ctxt): pop edx ; 5a
<bochs:6> info tab
cr3: 0x000000100000
0x00000000-0x01fffff -> 0x000000000000-0x000001fffff

```

Figure 7: 内存分配与释放测试效果

3. 为了演示效果，我缩小了队列的时间片，三级队列的时间片分别是 2、5、10。因为 TestA B C 三个进程是无限循环的进程，在系统启动后输出的信息很容易被覆盖，无法捕捉到它们在高级队列的输出演示信息，所以我设置进程在执行完三级队列的时间片之后再重置一次回到 0 级队列。

输出队列调度情况：可以看到进程 6 和 8 在运行完之后，我为了演示效果，将两个进程的状态重置，然后重新进入 0 级队列，然后按进入 0 级队列的先后顺序进程 6 和 8 各运行 2 个时间片之后，又按先后顺序进入 1 级队列各运行 5 个时间片，再进入 2 级队列，而二级队列里有先前还未完成的进程 7，所以进程 7 先运行 10 个时间片（屏幕未显示完）。

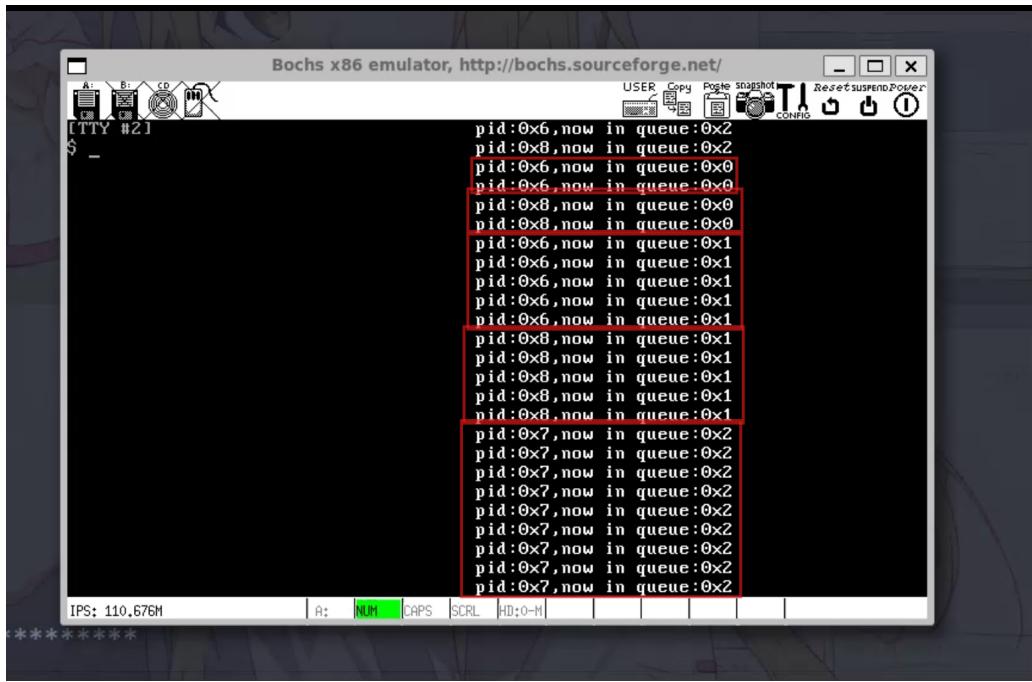


Figure 8: 多级反馈队列测试效果

绘制演示部分三个进程调度的甘蔗图，如下：

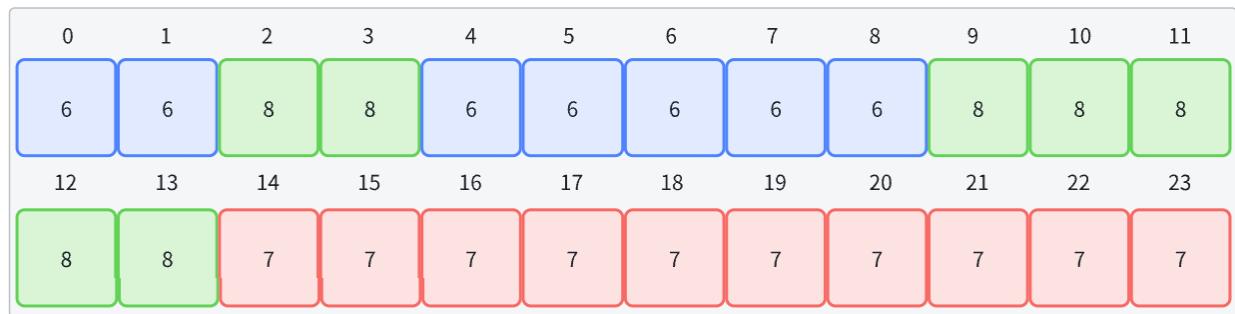


Figure 9: 演示进程调度过程

4. 在虚拟机终端中执行 tree 指令。

```

1 .
2   80m.img
3   Makefile
4   a.img
5   bochsrc
6   boot
7     boot.asm
8     boot.bin
9     include
10       fat12hdr.inc
11       load.inc
12       pm.inc

```

```
13      loader.asm
14      loader.bin
15  command
16      Makefile
17      echo
18      echo.c
19      edit
20      gopen
21      gopen.c
22      inst.tar
23      kernel.bin
24      ls
25      ls.c
26      open
27      pwd
28      pwd.c
29      rm
30      rm.c
31      start.asm
32      test
33      test.c
34      touch
35      touch.c
36  command.sh
37  fs
38      AES.c
39      disklog.c
40      link.c
41      main.c
42      misc.c
43      open.c
44      read_write.c
45  include
46      stdio.h
47      string.h
48  sys
49      AES.h
50      config.h
51      console.h
52      const.h
53      fs.h
54      global.h
55      hd.h
56      keyboard.h
57      keymap.h
58      proc.h
59      protect.h
60      proto.h
61      sconst.inc
62      tty.h
63      type.h
```

```
64     kernel
65         clock.c
66         console.c
67         global.c
68         hd.c
69         i8259.c
70         kernel.asm
71         keyboard.c
72         klib.c
73         kliba.asm
74         main.c
75         proc.c
76         protect.c
77         start.c
78         systask.c
79         tty.c
80         kernel.bin
81         kernel.elf
82         kernel.elf.debug
83         krnl.map
84         lib
85             check_file.c
86             close.c
87             exec.c
88             exit.c
89             fork.c
90             getpid.c
91             ls.c
92             lseek.c
93             misc.c
94             open.c
95             orangescrt.a
96             printf.c
97             read.c
98             stat.c
99             string.asm
100            syscall.asm
101            syslog.c
102            unlink.c
103            vsprintf.c
104            wait.c
105            write.c
106        mm
107            exec.c
108            forkexit.c
109            main.c
110        qemu_start.sh
111        scripts
112            genlog
113            splitgraphs
114
```

```
115 10 directories, 102 files
```

- 一开始 tty2 的 shell 程序是被我取消了的，可以看到 tty2 上并没有运行 shell 的命令提示符 \$。

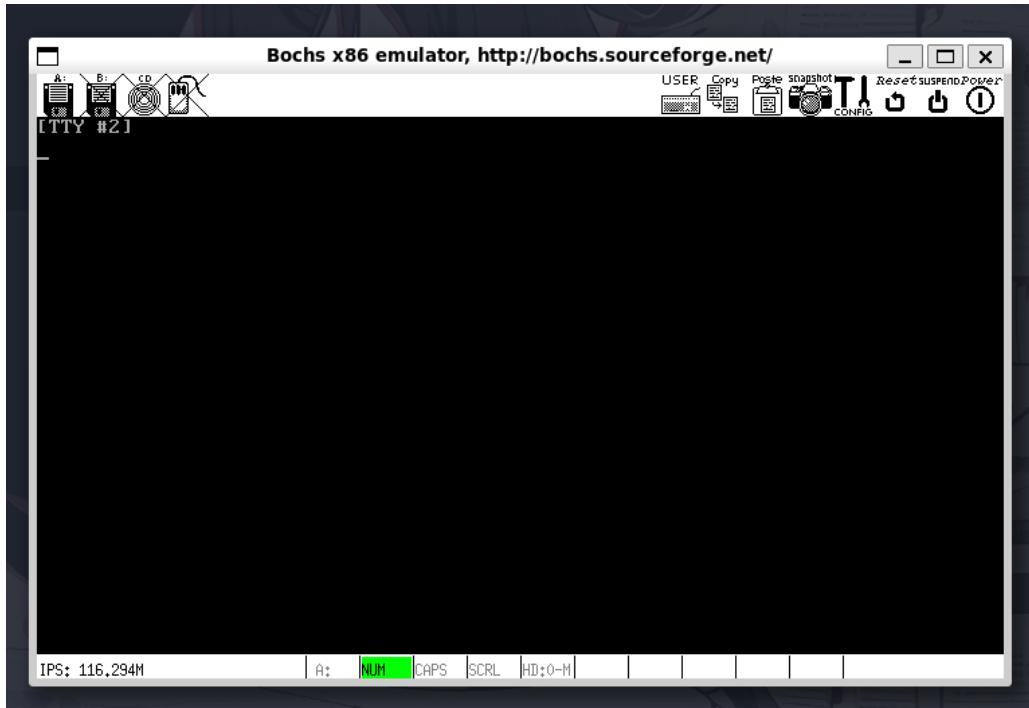


Figure 10: 彩蛋触发前 tty2 的内容

在这之后键盘输入彩蛋字符串 *innerpeace*，可以看到 tty0 上显示了 logo。

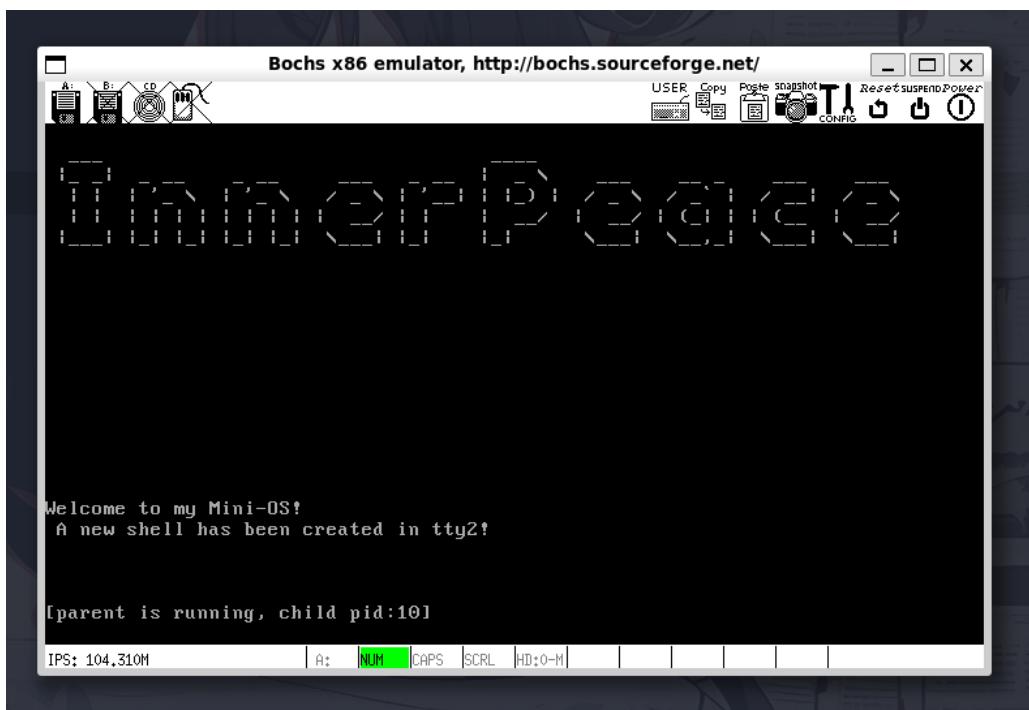


Figure 11: 显示 logo

同时，tty2 上激活了一个 shell。

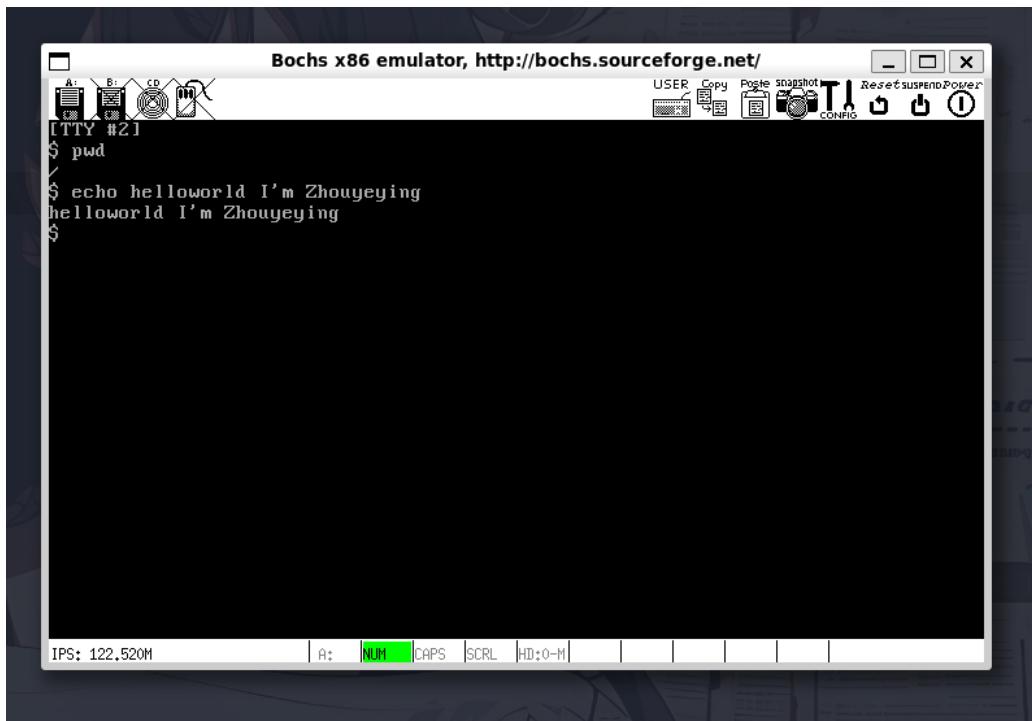


Figure 12: shell 激活

2.4.3 Part A 任务二：拓展 shell

测试用例设计

- ps: 查看进程
- kill: 结束进程
- touch: 生成文件
- ls: 查看目录下所有文件
- rm: 删除文件
- fopen : 打开和编辑文件

测试结果分析

1. ps: 查看进程

```
[TTY #1]
$ ps
PID  NAME  FLAGS
0   TTY    RECEIVING
1   SYS    Unknown
2   HD     RECEIVING
3   FS     RECEIVING
4   MM     RECEIVING
5   INIT   Unknown
6   TestA  Unknown
7   TestB  Unknown
8   TestC  Unknown
9   INIT_9  WAITING
10  INIT_10 RECEIVING
11  ps     RECEIVING
$ kill -10
Process 10 has been killed.
$ ps
PID  NAME  FLAGS
0   TTY    RECEIVING
1   SYS    Unknown
2   HD     RECEIVING
3   FS     RECEIVING
4   MM     RECEIVING
5   INIT   Unknown
IPS: 134.406M  | A: NUM CAPS SCRL HD:0-M | | | | | | | |
```

2. kill: 结束进程

```
ps      READING
$ kill -10
Process 10 has been killed.
$ ps
 PID  NAME  FLAGS
 0   TTY   RECEIVING
 1   SYS   Unknown
 2   HD    RECEIVING
 3   FS    RECEIVING
 4   MM    RECEIVING
 5   INIT  Unknown
 6   TestA Unknown
 7   TestB Unknown
 8   TestC Unknown
 9   INIT_9 Unknown
10   ps    RECEIVING
$
```

IPS: 135.400M A: NUM CAPS SCRL HD:0-M

3. touch: 生成文件

```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
[TTY #1]
$ touch
Usage: touch <filename> [filename...]
$ touch 1 2 3 4
Successfully created 1.
Successfully created 2.
Successfully created 3.
Successfully created 4.
$
```

IPS: 80.811M A: NUM CAPS SCRL HD:0-M

4. ls: 查看目录下所有文件

The screenshot shows a Bochs x86-64 emulator window with a terminal session. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The terminal window displays the following command-line session:

```
[TTY #1]
$ ls
. dev_tty0 dev_tty1 dev_tty2 cmd.tar kernel.bin log echo pwd touch rm ls ps kill
gopen attack test
$ ls -l
d----- 2 272 269 2048 1 .
c----- 3 0 1024 0 2 dev_tty0
c----- 5 0 1025 0 3 dev_tty1
c----- 3 0 1026 0 4 dev_tty2
----- 1 1048576 32768 2048 5 cmd.tar
----- 1 89904 2317 2048 6 kernel.bin
----- 1 859 4365 2048 7 log
----- 1 52872 6413 2048 8 echo
----- 1 52856 8461 2048 9 pwd
----- 1 52988 10509 2048 10 touch
----- 1 57360 12557 2048 11 rm
----- 1 57020 14605 2048 12 ls
----- 1 52936 16653 2048 13 ps
----- 1 53060 18701 2048 14 kill
----- 1 70668 20749 2048 15 gopen
----- 1 56060 22797 2048 16 attack
----- 1 52988 24845 2048 17 test
$ -
```

The status bar at the bottom shows "IPS: 77.448M" and a keyboard indicator with "NUM" highlighted.

5. rm: 删除文件

The screenshot shows a Bochs x86-64 emulator window with a terminal session. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The terminal window displays the following command-line session:

```
[TTY #1]
$ rm
Usage: rm <filename> [filename...]
$ touch 1 2
Successfully created 1.
Successfully created 2.
$ rm 1 2 3
File 1 deleted.
File 2 deleted.
Failed to delete file 3.
$ -
```

The status bar at the bottom shows "IPS: 77.030M" and a keyboard indicator with "NUM" highlighted.

6. gopen : 打开和编辑文件

The screenshot shows a terminal window within the Bochs x86-64 emulator. The window title is "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The terminal output is as follows:

```
Successfully created 1.  
$ gopen 1  
[INFO] This is a text file. Opening...  
[INFO] File is empty.  
[INPUT] Do you want to (A)ppend or (O)verwrite the file?  
a  
[INFO] File content will be appended.  
[INPUT] Enter new content to write to the file:  
HDW HDW  
[INFO] 2 bytes written to the file.  
[INFO] Updated content:  
HDW HDW  
$ gopen 1  
[INFO] This is a text file. Opening...  
[INFO] Current content:  
HDW HDW  
[INPUT] Do you want to (A)ppend or (O)verwrite the file?  
a  
[INFO] File content will be appended.  
[INPUT] Enter new content to write to the file:  
666  
[INFO] 4 bytes written to the file.  
[INFO] Updated content:  
HDW HDW 666  
$
```

The terminal interface includes a menu bar with options like USER, Copy, Paste, Snapshot, Reset, Suspend, Power, and CONFIG. At the bottom, there are status indicators for IP: 71.614M, A:, NUM, CAPS, SCRL, and HD:0-M.

2.4.4 Part A 任务三：系统日志框架

测试用例设计

- gopen log: 查看日志
- touch 1: 创建文件
- gopen 1: 编辑文件
- rm 1: 删除文件

测试结果分析

1. gopen log: 只能查看 log , 没有修改权限。同时正确的由 fd 得到 filename, 写入文件名。

Bochs x86-64 emulator, http://bochs.sourceforge.net/

```
[TTY #1]
$ gopen log
[INFO] This is a text file. Opening...
[INFO] Current content:
<2024-12-12 03:27:21> [Fork]: parent name = INIT pid = 5, child pid = 9
<2024-12-12 03:27:25> [Fork]: parent name = INIT pid = 5, child pid = 10
<2024-12-12 03:28:48> [Open]: Filename = gopen
<2024-12-12 03:28:48> [Fork]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:28:49> [Close]: Filename = gopen

[INFO] You have no access to modify this file.
$ _
```

IPS: 72.761M A: NUM CAPS SCRL HD:0-M

2. touch 1: 成功 fork 了子进程，创建并关闭了文件 1

Bochs x86-64 emulator, http://bochs.sourceforge.net/

```
[INFO] You have no access to modify this file.
$ touch 1
Successfully created 1.
$ gopen log
[INFO] This is a text file. Opening...
[INFO] Current content:
<2024-12-12 03:27:21> [Fork]: parent name = INIT pid = 5, child pid = 9
<2024-12-12 03:27:25> [Fork]: parent name = INIT pid = 5, child pid = 10
<2024-12-12 03:28:48> [Open]: Filename = gopen
<2024-12-12 03:28:48> [Fork]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:28:49> [Close]: Filename = gopen
<2024-12-12 03:28:49> [Wait]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:45:16> [Open]: Filename = touch
<2024-12-12 03:45:16> [Fork]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:45:17> [Close]: Filename = touch
<2024-12-12 03:45:20> [Create]: Filename = 1
<2024-12-12 03:45:21> [Write]: Filename = dev_tty1, Bytes = 24
<2024-12-12 03:45:21> [Close]: Filename = 1
<2024-12-12 03:45:21> [Wait]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:45:57> [Open]: Filename = gopen
<2024-12-12 03:45:57> [Fork]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:45:58> [Close]: Filename = gopen

[INFO] You have no access to modify this file.
$ _
```

IPS: 73.798M A: NUM CAPS SCRL HD:0-M

3. gopen 1: 写入文件 1 时，日志记录了打开文件 1，从 tty 输出文本内容，并且读取输入内容，写入文件 1，最后读取并关闭文件

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
A: B: CD D: X: USER Copy Paste Snapshot Reset Suspend Power CONFIG

<2024-12-12 03:41:28> [Open ]: Filename = gopen
<2024-12-12 03:41:29> [Read ]: Filename = gopen, Bytes = 72052
<2024-12-12 03:41:29> [Close]: Filename = gopen

<2024-12-12 03:41:30> [Open ]: Filename = 1
<2024-12-12 03:41:30> [Write]: Filename = dev_tty1, Bytes = 39
<2024-12-12 03:41:30> [Write]: Filename = dev_tty1, Bytes = 22
<2024-12-12 03:41:30> [Write]: Filename = dev_tty1, Bytes = 58
<2024-12-12 03:41:52> [Read ]: Filename = dev_tty1, Bytes = 1
<2024-12-12 03:41:52> [Write]: Filename = dev_tty1, Bytes = 38
<2024-12-12 03:41:52> [Write]: Filename = dev_tty1, Bytes = 49
<2024-12-12 03:42:18> [Read ]: Filename = dev_tty1, Bytes = 512
<2024-12-12 03:42:19> [Write]: Filename = 1, Bytes = 3
<2024-12-12 03:42:19> [Write]: Filename = dev_tty1, Bytes = 36
<2024-12-12 03:42:19> [Read ]: Filename = 1, Bytes = 3
<2024-12-12 03:42:19> [Write]: Filename = dev_tty1, Bytes = 24
<2024-12-12 03:42:19> [Write]: Filename = dev_tty1, Bytes = 3
<2024-12-12 03:42:20> [Write]: Filename = dev_tty1, Bytes = 1
<2024-12-12 03:42:20> [Close]: Filename = 1

<2024-12-12 03:42:20> [Wait ]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:42:53> [Open ]: Filename = gopen
<2024-12-12 03:42:54> [Read ]: Filename = gopen, Bytes = 72052
<2024-12-12 03:42:54> [Close]: Filename = gopen

[INFO] You have no access to modify this file.
$
```

IPS: 73.572M A: NUM CAPS SCRL HD:0-M

4. rm 1: 成功删除文件，显示日志

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
A: B: CD D: X: USER Copy Paste Snapshot Reset Suspend Power CONFIG

<2024-12-12 03:47:00> [Write]: Filename = dev_tty1, Bytes = 58
<2024-12-12 03:47:10> [Read ]: Filename = dev_tty1, Bytes = 1
<2024-12-12 03:47:11> [Write]: Filename = dev_tty1, Bytes = 38
<2024-12-12 03:47:11> [Write]: Filename = dev_tty1, Bytes = 49
<2024-12-12 03:47:24> [Read ]: Filename = dev_tty1, Bytes = 512
<2024-12-12 03:47:24> [Write]: Filename = 1, Bytes = 3
<2024-12-12 03:47:25> [Write]: Filename = dev_tty1, Bytes = 36
<2024-12-12 03:47:25> [Read ]: Filename = 1, Bytes = 3
<2024-12-12 03:47:26> [Write]: Filename = dev_tty1, Bytes = 24
<2024-12-12 03:47:26> [Write]: Filename = dev_tty1, Bytes = 3
<2024-12-12 03:47:26> [Write]: Filename = dev_tty1, Bytes = 1
<2024-12-12 03:47:27> [Close]: Filename = 1
<2024-12-12 03:47:27> [Wait ]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:47:57> [Open ]: Filename = rm
<2024-12-12 03:47:59> [Read ]: Filename = rm, Bytes = 62336
<2024-12-12 03:47:59> [Close]: Filename = rm
<2024-12-12 03:47:59> [Delete]: Filename = 1
<2024-12-12 03:48:00> [Write]: Filename = dev_tty1, Bytes = 16
<2024-12-12 03:48:00> [Wait ]: parent name = INIT_9 pid = 9, child pid = 11
<2024-12-12 03:48:43> [Open ]: Filename = gopen
<2024-12-12 03:48:44> [Read ]: Filename = gopen, Bytes = 72052
<2024-12-12 03:48:44> [Close]: Filename = gopen

[INFO] You have no access to modify this file.
$
```

IPS: 76.082M A: NUM CAPS SCRL HD:0-M

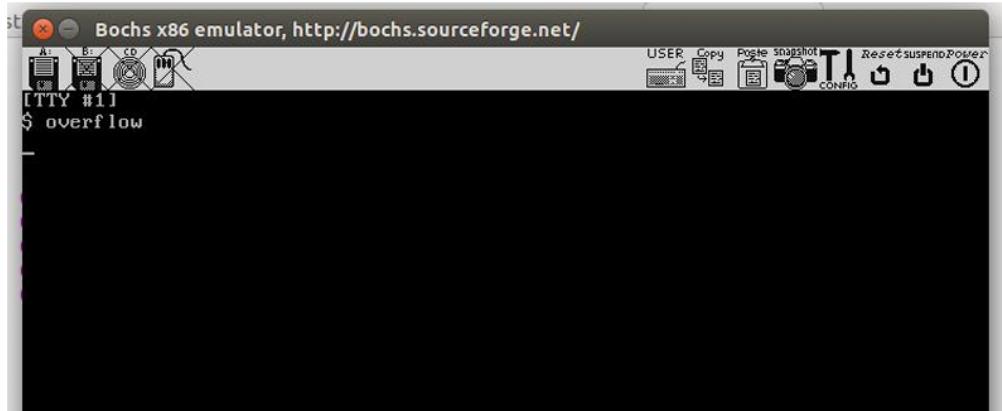
2.4.5 Part B 任务一：自我 OS 分析

测试用例设计

- 栈缓冲区溢出模块命令，终端中打印出一个“stackoverflow!”字符串，同时代码进入死循环，攻击成功。
- 格式化字符串模块。任意地址读尝试泄露栈上数据，任意地址写尝试修改变量 a 的值。

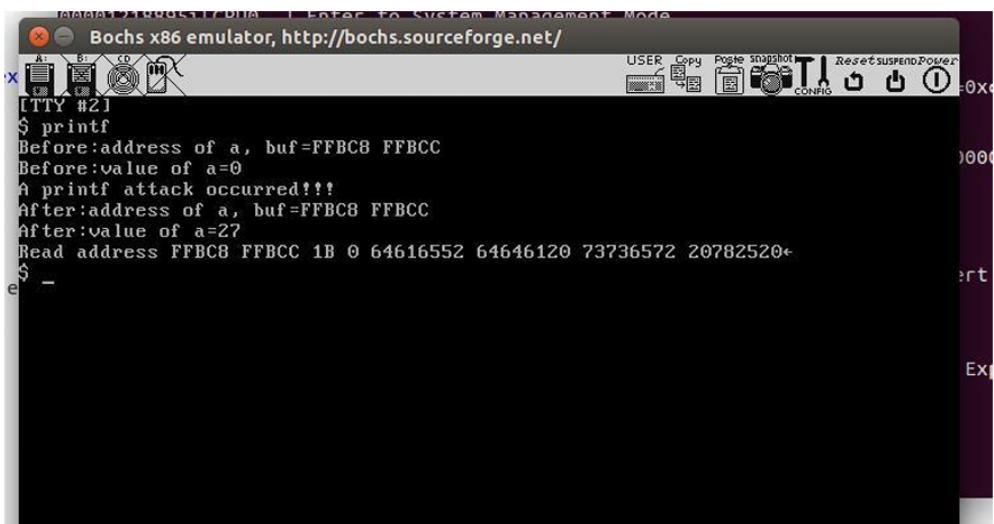
测试结果分析

1. 栈缓冲区溢出模块



```
[parent is running, child pid:9]
[child is running, pid:9]
[parent is running, child pid:10]
[child is running, pid:10]
stackoverflow!
```

2. 格式化字符串模块



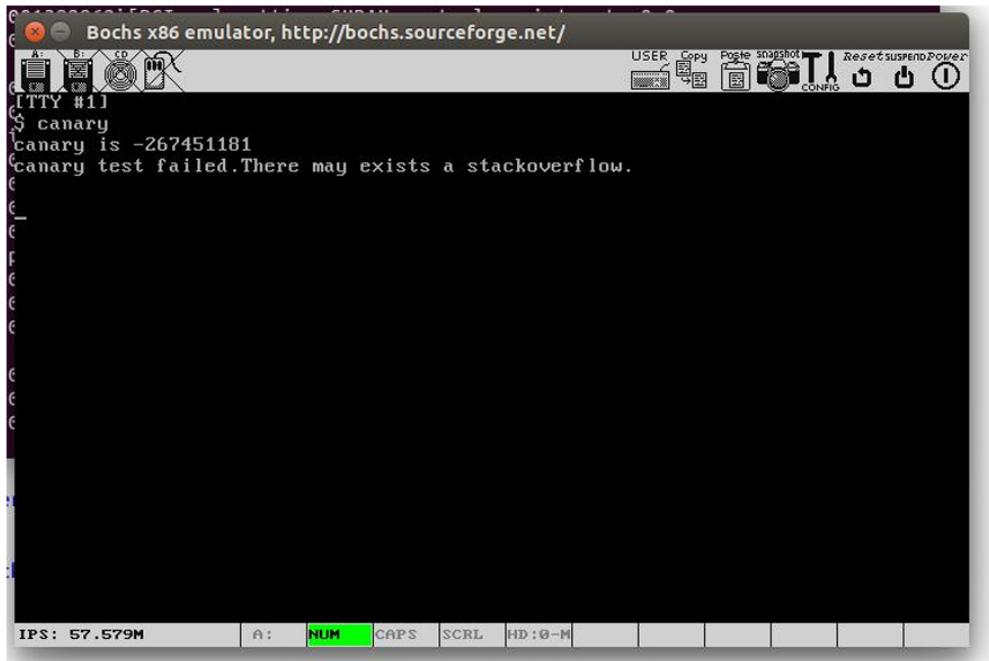
2.4.6 Part B 任务二：自我 OS 防护：针对发现的攻击，进行防护

测试用例设计

1. Canary 模块，运行 canary 命令，终端中打印出一个”canary test failed. There may exists a stackoverflow.” 字符串，shellcode 没有执行，则说明防御成功。
2. 内存边界检查模块，防护后再次运行 printf_check 命令，a 变量的值未发生改变

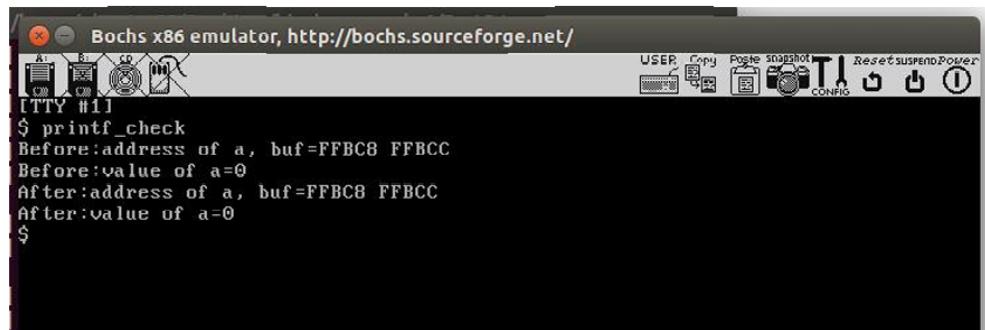
测试结果分析

1. Canary 模块



The screenshot shows a terminal window titled "Bochs x86 emulator, http://bochs.sourceforge.net/" with the command "canary" entered. The output indicates a failure: "canary is -267451181" and "canary test failed. There may exists a stackoverflow." The terminal interface includes a toolbar at the top with icons for USER, Copy, Paste, snapshot, Reset, Suspend, Power, and CONFIG. The bottom status bar shows "IPS: 57.579M" and a key indicator where "NUM" is highlighted.

2. 内存边界检查模块



The screenshot shows a terminal window titled "Bochs x86 emulator, http://bochs.sourceforge.net/" with the command "printf_check" entered. The output shows memory addresses and values before and after a operation: "Before: address of a, buf=FFBC8 FFBCC", "Before: value of a=0", "After: address of a, buf=FFBC8 FFBCC", and "After: value of a=0". The terminal interface includes a toolbar at the top with icons for USER, Copy, Paste, snapshot, Reset, Suspend, Power, and CONFIG. The bottom status bar shows "IPS: 57.579M" and a key indicator where "NUM" is highlighted.

2.4.7 Part B 任务二：自我 OS 防护 2. 扩展 OS 的数据防泄露能力

技术路线一：对磁盘文件进行加密保护，要求文件打开时，读入内存的为明文，如果进行编辑，写回磁盘的为密文。密码算法、密钥管理可自行设计。

测试用例设计

使用自定义字符串“12345678”作为种子，通过线性同余生成器生成初始密钥，预期得到的密钥串：

key : 2A1BB891F6F764CD8293D0C9CEEFFC85

通过密钥串生成 10 轮加解密用的加解密密钥，预期得到的加解密密钥串：

Encryption subkeys:

2A1BB891F6F764CD8293D0C9CEFFC85
F4AB2F1A025C4D7802C9FB14E20C679B
41AEB65383D821AFDB3D26C516B9D91B
93C912C6FA1F50D173C2B56B53D2C4C2
2D90FA50B08278D7D52DC6BD35038A4E
459F4915D3CEBF5EE97F3D8285D342E
8FBC91E3AC3B0F1DE677BC2FC59D3C01
1A02F73091336FDF417A2D3C29354DF4
B169DAFA11D0817029D2D6A12E2E2AAA

Decryption subkeys:

2A1BB891F6F764CD8293D0C9CEFFC85
B169DAFA11D0817029D2D6A12E2E2AAA
1A02F73091336FDF417A2D3C29354DF4
8FBC91E3AC3B0F1DE677BC2FC59D3C01
459F4915D3CEBF5EE97F3D8285D342E
2D90FA50B08278D7D52DC6BD35038A4E
93C912C6FA1F50D173C2B56B53D2C4C2
41AEB65383D821AFDB3D26C516B9D91B
F4AB2F1A025C4D7802C9FB14E20C679B
B44C365515D35763B79AB7D928D6A2EA

在 shell 打开磁盘中存放密钥的 key.txt 文件时，预期系统显示被加密的密钥内容，拒绝用户的修改请求。在打开其他文件时，预期正常显示文件内容，根据文件加密的透明性用户无法感受到加解密过程，在打开文本文件时正常显示文本内容；在使用 shell 指令时，预期系统正常执行指令。

测试结果分析

1. 成功生成密钥并写入 key.txt 文件。

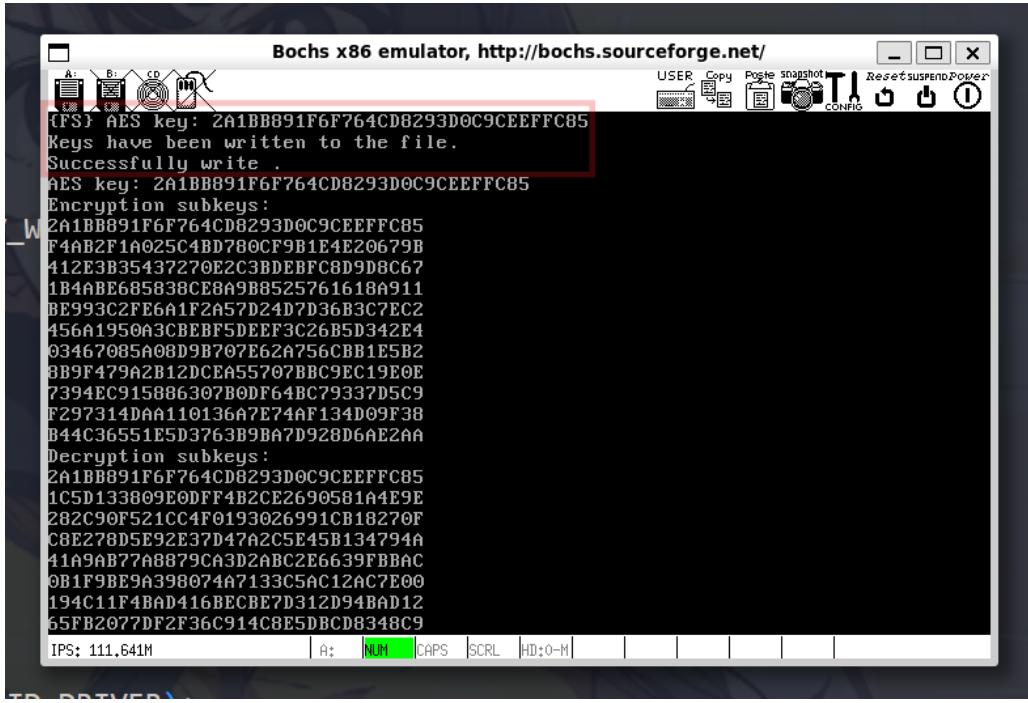


Figure 13: 密钥生成成功

2. 密钥生成内容、加密钥、解密钥生成结果与预期一致。



Figure 14: 密钥生成内容

3. 尝试打开密钥文件 key.txt，显示加密内容，并且不允许用户进行修改。

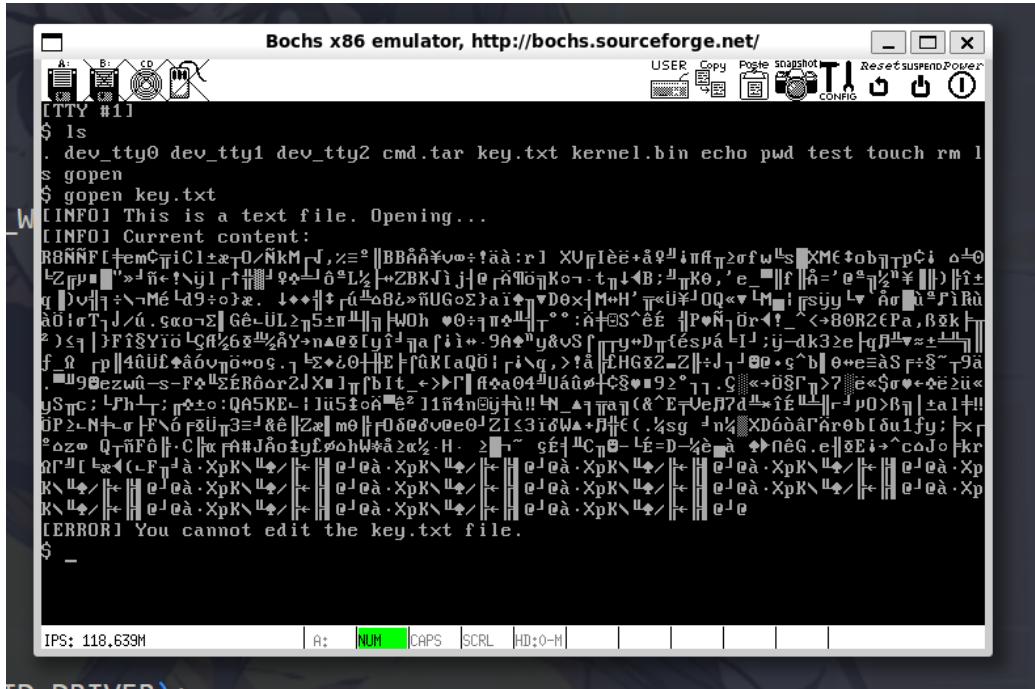


Figure 15: 打开密钥文件

4. 打开并编译一个文本文件，无法直接从 shell 中看出加解密过程，但为了演示在 tty0 输出加解密前后的文本内容。

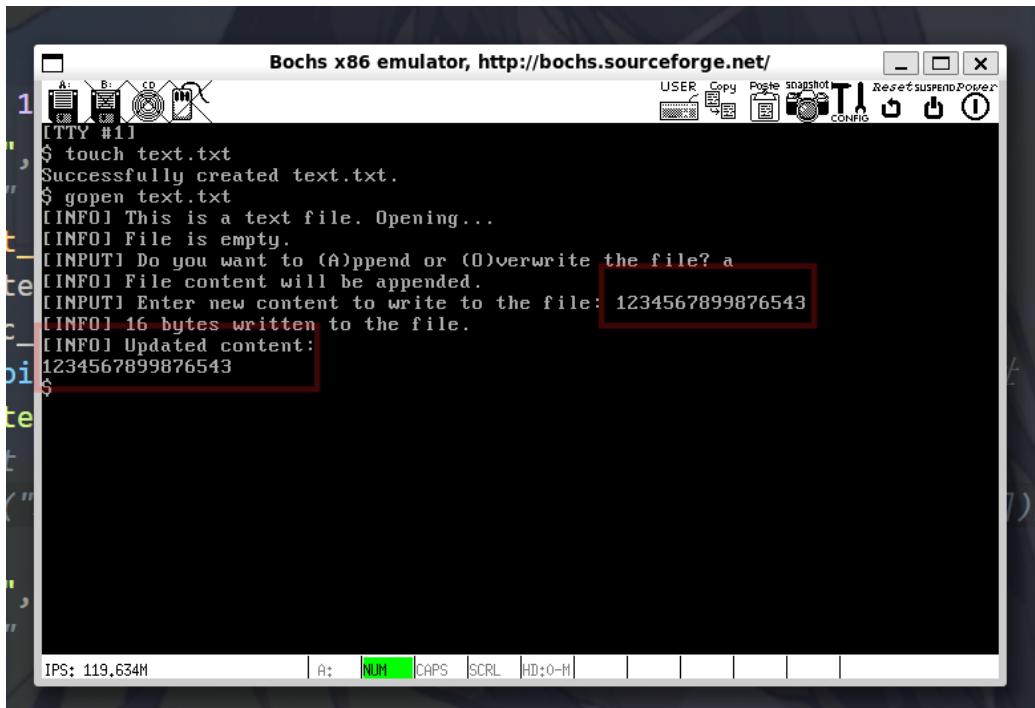


Figure 16: 文件加解密透明过程

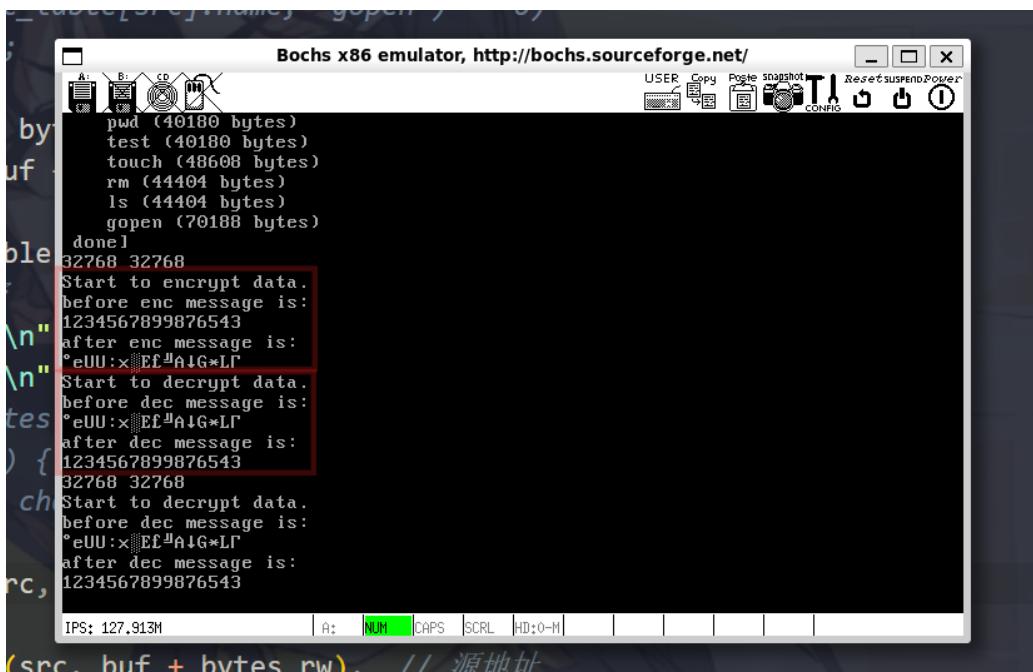


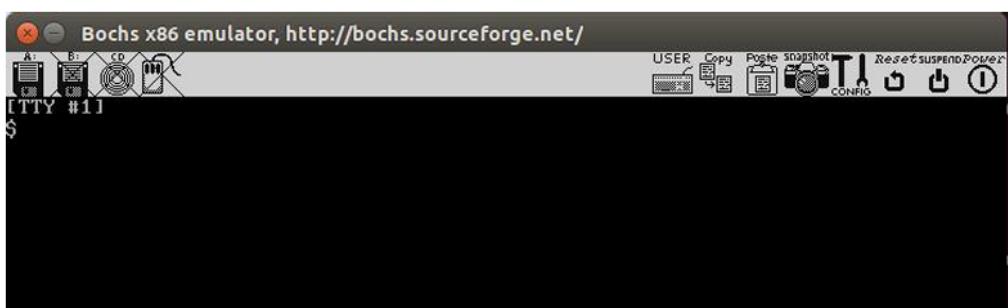
Figure 17: 文件加解密前后字符串内容

2.4.8 Part B 任务二：自我 OS 防护：扩展 OS 的数据防泄露能力：文件资源访问保护 测试用例设计

分别使用高权限的 Init 进程和低权限的 TestA 进程打开 tty 文件，预期高权限的 Init 进程能够打开，低权限的 TestA 不能打开，输出错误信息。

测试结果分析

- ## 1. Init 能够打开



- ## 2. TestA 打开失败并输出错误信息。

```
{FS} dev size: 0x9D41 sectors
{FS} devbase:0x9DFE00, sb:0x9E0000, imap:0x9E0200, smap:0x9E0400
      inodes:0x9E1800, 1st_sector:0xA01800
*****process does not have access to this file!!!*****
*****in pid [6], filename [dev_tty0]*****
```

3 实验总结与心得

3.1 实验总结

在操作系统最后的大实验中，我们小组的基本分工是 A1 集成 OS、A2 拓展 shell、A3 日志系统、B1+B2 自我 OS 安全分析与防护、B3 拓展 OS 安全防护的形式，不过 B3 只做一个技术路线相对容易，这样分工也需要有五个人，就将 A1 整合 OS 和 B3 拓展安全防护绑定在了一起，由一个同学负责。然后任务分配按兴趣优先，自由选择，最后我负责的部分是 A1+B3。其中每个人在负责的部分中占主导地位，但是在遇到问题时也和其他成员进行了探讨。

对于我负责的部分，我的完成情况为：

1. 集成 OS

- 使用软盘或者硬盘引导 OS 启动，这部分内容主要是修改项目 bug 并跑通了项目。
- 集成内存分配与释放功能。
- 实现多级反馈队列调度算法。
- 构造了一个系统后门实现彩蛋。

2. 磁盘文件加密保护

- 构建了密钥生成算法和密钥管理机制。
- 将 AES 加解密部署到项目中。
- 实现磁盘文件透明加解密。

3.2 个人心得体会

特别感谢严飞老师一学期以来的悉心指导，在完成所有实验课程内容的过程中，遇到的 bug 和难题在向严飞老师寻求帮助的时候都会得到老师的悉心指导和帮助，严飞老师是为数不多能亲自去搭建学生的代码环境复现 bug 并提出解决方案的老师。虽然一路上磕磕绊绊，但还是在严飞老师的指导下成功编写出来了一个“跑得起来”并且能够实现自己添加的功能的 MiniOS，不说成就感有多大，但起码对这门课程、对 MiniOS 的实现，满足感还是足足的。

同时也要感谢我的队友，黄东威、王俊杰和程序。他们在一个学期的学习和讨论中没有放弃，不断地鞭策我继续前进和钻研下去，在相互解答疑惑的过程中也不断增进对项目的理解。在完成小实验和大实验的过程中，我们都尽力做好了该做的工作。特别感谢在最后的大实验中，为我的 debug 过程提出建议的王俊杰同学和提供工具使用支持的黄东威同学，在他们的帮助下我的实验过程才步入现代化的过程。

同时，也必须得感叹一下计算机科学与技术的飞速发展和进步。从底层硬件，理论的逻辑电路和实践的电子电路，到应用计算机组成原理的大组件的构成，然后到编译器将汇编指令解析、优化生成机器码，再到分配调度硬件资源的操作系统，最后才到我们接触的操作系统上的各个应用的使用，甚至能够在强大的前端支持下生成和显示出各种精美的图形化界面，这才构成了我们最简单的微机，更不要说使用计算机网络实现世界互联、在微机上使用各种高级的算法、开发各种强大的软件，甚至到今天能够帮助人类学习和生活的人工智能已经崭露头角。计算机的实现和发展就是一个万丈高楼平地起的过程，让人不禁感叹它的宏伟与壮观。

而操作系统的学习，就是一个在学习为这幢高楼打好地基的过程。在上这门课之前我都不敢想象一个从零开始搭建自己的 OS 的过程，虽然是用了 OrangeS 的框架，但在老师的引导下，这也是我们一点一点代码啃下来的过程。虽然称不上构建的 OS 是一栋美丽华贵的高楼，但起码是一幢封顶的火柴盒了。越是亲身实践经历，越是感受到计算机的精巧和前人的辛勤耕耘，这才有了我们今天如此强大计算机，可以自豪的说我们现在就是站在了巨人的肩膀上看世界。操作系统实验是我学习计算机科学与技术的一个重要的过程，是让我纵观和学习“巨人”的一个过程，我也希望这是一个为我打好基础，争取未来为这个巨人、这幢高楼的美化和发展做出自己的贡献的过程。

感谢一路走来的自己，说实话 OS 实验的任务并不轻松，需要课下花费很多时间去钻研代码、学习知识，但走到今天，轻舟已过万重山，感谢这段将 OS 理论与更先进的 OS 实践结合起来的过程。希望未来的我也能继续一步一个脚印地、努力地走下去。

References

- [1] 于渊. *OrangeS: 一个操作系统的实现*. 电子工业出版社, 2009 年 6 月.
- [2] CloudBlaze. (2020, June 19). [操作系统原理与实现]Bochs 与 GDB. CSDN 博客. Retrieved December 12, 2024, from <https://blog.csdn.net/cloudblaze/article/details/106451123>
- [3] 随书源码: Yyu. (n.d.). osfs00. GitHub. Retrieved December 12, 2024, from <https://github.com/yyu/osfs00>
- [4] openssl. openssl. GitHub. Retrieved December 12, 2024, from <https://github.com/openssl/openssl>

实验报告评分标准

实验报告的考评依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性、以及实验答辩情况等六个维度综合考评，建议分值和标准如下。

95-100	<ul style="list-style-type: none">• 实验内容完整，有较多超出实验任务要求的内容；• 实验步骤非常详尽，能够体现非常完整的实验过程；• 实验结果不仅符合预期要求，而且结果分析详尽、透彻；• 实验心得与分析深刻，展现出对实验原理和方法的深入理解；• 实验报告结构严谨，格式规范；• 实验答辩非常优秀，实验总体达到了很高水平。
90-94	<ul style="list-style-type: none">• 实验内容完整，有一定超出实验任务要求的内容；• 实验步骤详尽，能够体现完整的实验过程；• 实验结果符合预期要求，结果分析详尽；• 实验心得与分析较为深入，能够很好地反映实验原理和方法的理解；• 实验报告结构清晰，格式规范；• 实验答辩优秀，实验总体达到优秀水平。
85-89	<ul style="list-style-type: none">• 实验内容较为完整，较好完成了实验任务要求的内容；• 实验步骤较为详尽，能够体现较为完整的实验过程；• 实验结果符合预期要求，结果分析合理；• 实验心得与分析较为到位，能够反映实验原理和方法的理解；• 实验报告结构较为完整，格式规范；• 实验答辩良好，实验总体符合要求。

80-84	<ul style="list-style-type: none"> • 实验内容较为完整，完成了实验任务要求的内容 • 实验步骤较为详尽，能够体现实验过程 • 实验结果大致符合预期要求，结果分析基本合理； • 实验心得与分析较为到位，但仍有提升空间； • 实验报告结构较为完整，格式较为规范； • 实验答辩顺利，实验总体符合基本要求。
70-79	<ul style="list-style-type: none"> • 实验内容基本完整，完成了实验任务要求的大部分内容 • 实验步骤基本详尽，能够基本体现实验过程 • 实验结果基本符合预期要求，结果分析基本到位； • 实验报告结构基本完整，格式基本正确； • 实验心得与分析尚可，但需要进一步深入； • 实验答辩有较多改进空间，总体达到了基本要求。
60-69	<ul style="list-style-type: none"> • 实验内容有不够充实，实验任务完成有明显缺失； • 实验步骤不够详尽，基本体现实验过程，但存在一定问题； • 实验结果部分符合预期要求，结果分析不够全面； • 实验心得与分析不够深入，未能完全展现对实验原理和方法的理解； • 实验报告结构不够完整，格式有待增强； • 实验答辩能回答问题，总体基本达到要求，但需要较大改进。

60 以下	<ul style="list-style-type: none">• 实验内容严重缺失、实验态度不够端正；• 实验步骤不够详尽，不能够体现完整的实验过程；• 实验结果存在严重的错误，无法达到预期，无结果分析或者错误明显；• 实验心得与思考无或者非常不深入；• 实验报告很不完整，格式存在很大问题；• 实验报告很不完整，格式存在很大问题；
-------	--

教师评语评分

评语：

评 分：_____

评阅人：

年 月 日