# 目录

[一、实验内容](#_bookmark0) 1

[1 . 1 实验选题内容](#_bookmark1) 1

[1 . 1 . 1 Part A 综合装配](#_bookmark2) 1

[1 . 1 . 2 Part B 安全分析与防御](#_bookmark3) 2

[1 . 2 小组分工情况](#_bookmark4) 2

[二、实验环境](#_bookmark5) 2

[三、实验方案设计](#_bookmark6) 3

[3 . 1 Part A 任 务一](#_bookmark7) 3

[3 . 2 Part A 任 务二](#_bookmark8) 4

[3 . 3 Part A 任 务三](#_bookmark9) 4

[3 . 3 . 1 wait 和 exit](#_bookmark10) 7

[3 . 4 Part A 任 务四](#_bookmark11) 8

[3 . 4 . 1 IPC](#_bookmark12) 8

[3 . 4 . 2 内存管理机制](#_bookmark13) 10

[3 . 4 . 3 分页实现思路](#_bookmark14) 14

[3 . 5 Part B 任 务一](#_bookmark15) 15

[3 . 5 . 1 ELF 文件注入](#_bookmark16) 15

[3 . 5 . 2 缓冲区溢出与 ROP](#_bookmark17) 16

[3 . 6 Part B 任 务二](#_bookmark18) 17

1. [. 7 Part B 任 务三](#_bookmark19) 17

[四、实验过程分析](#_bookmark20) 20

1. [. 1 Part A 任 务一](#_bookmark21) 20

[4 . 2 Part A 任 务二](#_bookmark22) 27

[4 . 3 Part A 任 务三](#_bookmark23) 32

[4 . 4 Part A 任 务四](#_bookmark24) 36

[4 . 5 Part B 任 务一](#_bookmark25) 43

[4 . 6 Part B 任 务二](#_bookmark26) 49

1. [. 7 Part B 任 务三](#_bookmark27) 53

[五、实验结果总结](#_bookmark28) 55

[5 . 1 准备工作](#_bookmark29) 55

[5 . 2 实验结果](#_bookmark30) 56

[5 . 2 . 1 Part A](#_bookmark31) 56

[5 . 2 . 2 Part B](#_bookmark32) 59

1. [. 3 个人心得体会](#_bookmark33) 62

[六、指导教师评语及成绩](#_bookmark34) 63

**武汉大学国家网络安全学院实验报告**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **课程名称** | 操作系统设计与实践 | | | **成 绩** |  | **教师签名** |  |
| **实验名称** | 综合装配 & 安全分析 | | | **实验序号** | ﬁnal | **实验日期** | 2020.11.26 |
| **姓 名** | 彭雨昂 | **学号** | 2019302140023 | **专 业** | 信息安全 | **年级班级** | 19 级 8 班 |

# 一、 实验内容

## 1 . 1 实验选题内容

## 1 . 1 . 1 Part A 综合装配

### 任务一：在已有实验代码基础上，将 1-7 章节进行功能综合，形成你自己的一个简易 OS。可以实现如下功能：

* + 可以考虑使用软盘或者硬盘，启动该 OS。
  + 能够实现你在前面章节所实现的，内存分配与释放。
  + 能够进行多进程管理，并实现一个有别于本教材上已列出的多进程调度策略，及一个评价该策略性能的小程序。（例如：实现一个多级反馈队列调度算法，并用其尝试调度 5-8 个任务， 输出性能评价信息。）
  + 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
* 任务二：参照第 10 章、第 11 章内容，理清相关代码结构，以及 OrangeS 所支持的功能，扩展

shell，完成如下任务：

* + 利用当前 OrangeS 所提供的系统调用和 API，编写 2 个以上可执行程序（功能自定），并编译生成存储在文件系统中
  + 在 Shell 中调入你所编写的可执行程序，启动并执行进程（注意使用教材中所提供的系统调用来实现）
  + 进程结束后返回 Shell
* 任务三：改造任务二的 shell，使其能够在同一个 shell 中，支持多任务执行
  + 注意现有内存管理可能不支持多程序支持
  + 可执行程序的装入和内存定位问题需要仔细考虑
* 任务四：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

## 1 . 1 . 2 Part B 安全分析与防御

* 任务一：自我 OS 安全分析
  + 分析提示：可执行文件的篡改、内存破坏漏洞
  + POC 实现：

∗ 编写一个 C 程序，该程序查找 OS 中的可执行文件，对可执行文件添加额外的代码。

∗ 编写一个 C 程序，该程序查找 OS 中的可执行文件，对可执行文件添加额外的代码。

* 任务二：可信防护之静态度量
  + 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：

∗ 这样的度量，是否能够抵御对可执行文件的篡改？

∗ 完整性校验算法，使用奇偶校验算法，是否存在什么问题？

∗ 完整性校验值应该存在哪里？

* 任务三：可信防护之动态度量
  + 对你的 OS 进行扩充，编写一个软件中断，该中断能够响应定时中断触发。
  + 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：

∗ 如何理解“合法”的概念？

∗ 你的实现能否抵御 POC 实现中，第二个攻击？

∗ 这种度量方法的效率如何，存在什么额外的安全问题？

## 1 . 2 小组分工情况

小组内各个成员都积极参与各个部分的讨论，共同完成这次实验，下面是各个成员主要分工情况：

* 彭雨昂参与 Part A 和 Part B 全任务
* 王浩翔参与 Part B 全任务
* 庞紫萱参与 Part A 全任务
* 唐浩淼参与 Part A 全任务

# 二、 实验环境

* Ubuntu 16.04.1
* VMWare Workstation 16 player
* bochs 2.6.8

# 三、 实验方案设计

## 3 . 1 Part A 任务一

该任务主要实现一个多级反馈队列调度算法，并且实现评价该策略性能的小程序。考虑到评价和展示该多级反馈队列的信息，我们会打印大量信息，和第十章的一些输出信息重叠在一起会造成影响， 于是我们这个任务主要基于第六章 r 文件夹的代码。

多级反馈队列的**算法描述**如下：

### 进程在进入待调度的队列等待时，首先进入优先级最高的 *Q*1 等待。

* 首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：*Q*1、*Q*2、*Q*3 三个队列，当且仅当在 *Q*1 中没有进程等待时才去调度 *Q*2，同理，只有 *Q*1、*Q*2 都为空时才会去调度 *Q*3。
* 对于同一个队列中的各个进程，按照 FCFS 分配时间片调度。比如 *Q*1 队列的时间片为 N，那么*Q*1 中的作业在经历了 N 个时间片后若还没有完成，则进入 *Q*2 队列等待，若 *Q*2 的时间片用完后作业还不能完成，一直进入下一级队列末尾，直至完成。
* 在最后一个队列 *Qn* 中的各个进程，按照时间片轮转分配时间片调度。
* 在低优先级的队列中的进程在运行时，又有新到达的作业，此时须立即把正在运行的进程放回当前队列的队尾，然后把处理机分给高优先级进程。换而言之，任何时刻，只有当第 1 到 i-1 队列全部为空时，才会去执行第 i 队列的进程（抢占式）。特别说明，当再度运行到当前队列的该进程时，仅分配上次还未完成的时间片，不再分配该队列对应的完整时间片。

基于此，我们的设计思路是：

* 在进程表中加入所在队列以及所在队列的位置两个字段，这样就可以表示进程所在队列的位置。
* 一个进程不仅需要有一个总时间片了，还应该在进程表中加入在当前队列的剩余时间片。这样当在该队列剩余时间片为 0 的时候，就会被转移至下一个队列（除了在最后一个队列放在队尾）
* 利用 **clocker\_handler()** 和 **schedule()** 函数配合实现多级反馈调度算法

### 每次时钟中断检查当前进程是否还有总时间片，或者是否在当前队列还有时间片。如果有，那么总时间片和当前队列的时间片都需要减一；

* + 如果没有，就需要调用 **schedule()** 函数处理该进程，把它放入下一个队列（或者当前队列队尾），并且选择下一个进程。

评价该算法性能的小程序需要打印响应时间、周转时间和等待时间等等信息。比较可惜的是，因为我们基于第六章实现的，在这里操作系统所有进程还是我们手工准备的，而不能进行 fork 中途加入进程。所以我们多级反馈队列相应也没有实现抢占式算法。这里我们的思路是可以再往进程表中加入提交时间、响应时间、等待时间、周转时间等性能字段。由于是手工准备的进程，那么提交时间都是 0 时刻；响应时间就是第一次处理该进程的时间，我们可以通过判断进程表的响应字段是否被初始化从而获得响应时间；周转时间是进程结束时刻减去提交时刻，结束时刻只需要在时钟中断中，判断到总时间片是否用完获得结束时刻；等待时间就是周转时间减去进程刚开始初始化的需要运行的时间片。

## 3 . 2 Part A 任务二

该部分需要拓展 shell，为 shell 添加应用程序。在 orange 操作系统中，它的实现方式比较简单粗暴。

* 应用程序编写
  + 将应用程序需要使用的库函数单独链接成一个库文件，然后将写好的应用程序和库文件编译链接起来。
* 应用程序安装
  + 将应用程序打包.tar；
  + 将 tar 文件用工具写入磁盘映像的某段特定扇区；
  + 启动系统时，mkfs() 会在文件系统中建立一个新文件 cmd.tar，其中 inode 的 i\_start\_sect

的值会被设置为上一步写入的扇区的扇区号；

* + 某个进程会将 cmd.tar 解包，将其包含的文件存入文件系统。

## 3 . 3 Part A 任务三

为实现对多任务执行的支持，需要对 shell 进行改造，使之可以同时解析和执行多条指令。

,

shell 的代码在 kernel/main.c 中，它由 Init() 进程 fork 出来，如下代码所示，Init 进程打开

shell，分别运行在 TTY1 和 TTY2 上。

**void** Init()

{

**int** fd\_stdin = open("/dev\_tty0", O\_RDWR); assert(fd\_stdin == 0);

**int** fd\_stdout = open("/dev\_tty0", O\_RDWR);

assert(fd\_stdout == 1);

printf("Init() is running ...\n");

/\* extract ‘cmd.tar’ \*/ untar("/cmd.tar");

**char** \* tty\_list[] = {"/dev\_tty1", "/dev\_tty2"};

**int** i;

**for** (i = 0; i < **sizeof**(tty\_list) / **sizeof**(tty\_list[0]); i++) {

**int** pid = fork();

**if** (pid != 0) { /\* parent process \*/

printf("[parent is running, child pid:%d]\n", pid);

}

**else** { /\* child process \*/

了两

个

＼

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

|  |  |  |  |
| --- | --- | --- | --- |
| 22 |  |  | printf("[child is running, pid:%d]\n", getpid()); |
| 23 |  |  | close(fd\_stdin); |
| 24 |  |  | close(fd\_stdout); |
| 25 |  |  |  |
| 26 |  |  | shabby\_shell(tty\_list[i]); |
| 27 |  |  | assert(0); |
| 28 |  | } |  |
| 29 | } |  |  |
| 30 |  |  |  |

31

**while** (1) {

**int** s;

**int** child = wait(&s);

printf("child (%d) exited with status: %d.\n", child, s);

}

assert(0);

1.

}

shell 目前的功能很简单，就是读取命令并且执行之。代码如下所示，shabby\_shell 用 read() 读取

32

33

34

35

36

37 \_)

### 用户输入，然后 fork() 出一个子进程，在子进程中将输入交给 execv() 来执行。如果用户的输入并不是

一个合法的命令，那么 shabby\_shel1 只是将命令回显出来，不做其他任何处理。

,

＼

1 **void** shabby\_shell(**const char** \* tty\_name)

2 {

1. **int** fd\_stdin = open(tty\_name, O\_RDWR);
2. assert(fd\_stdin == 0);
3. **int** fd\_stdout = open(tty\_name, O\_RDWR);
4. assert(fd\_stdout == 1);

7

8 **char** rdbuf[128];

9

10 **while** (1) {

11 write(1, "$ ", 2);

1. **int** r = read(0, rdbuf, 70);
2. rdbuf[r] = 0;

14

1. **int** argc = 0;
2. **char** \* argv[PROC\_ORIGIN\_STACK];
3. **char** \* p = rdbuf;
4. **char** \* s;
5. **int** word = 0;
6. **char** ch;
7. **do** {
8. ch = \*p;

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55 }

56

**if** (\*p != ’ ’ && \*p != 0&& !word) { s = p;

word = 1;

}

**if** ((\*p == ’ ’ || \*p == 0) && word) { word = 0;

argv[argc++] = s;

\*p = 0;

} p++;

} **while**(ch);

argv[argc] = 0;

**int** fd = open(argv[0], O\_RDWR);

**if** (fd == -1) {

**if** (rdbuf[0]) { write(1, "{", 1); write(1, rdbuf, r); write(1, "}\n", 2);

}

}

**else** {

close(fd);

**int** pid = fork();

**if** (pid != 0) { /\* parent \*/

**int** s; wait(&s);

}

**else** { /\* child \*/ execv(argv[0], argv);

}

}

57

58

59 1.}

close(1);

close(0);

#### \_)

我们利用 & 符号分割多条命令，那么我们拓展 shabby\_shell 能够执行多条命令的思路就是：

* + 创建二维字符串数组 multi\_argv[MAX\_SHELL\_PROC][MAX\_SHELL\_PROC\_STACK]
  + 在argv 中保存完所有字符串后，我们再对 argv 进行扫描，把用 & 分割的命令分别保存在multi\_argv

中

* + 用 for 循环进行 fork 出子进程，同时我们要考虑如下问题
    - 父进程利用 for 循环进行 fork，子进程也同样会在该循环
    - 子进程如果抢占了父进程，那么父进程可能无法 fork 完所有子进程，导致无法运行多条命令上述问题会在实验过程分析中仔细考虑与解决。

## 3 . 3 . 1 wait 和 exit

在 shabby\_shell 中，还有一个 wait 函数，wait 和 exit 是一对函数。exit() 执行后杀死进程，wait() 执行后挂起程序，与 fork() 相同，这两个函数工作时将会返回 EXIT 和 WAIT 消息给 MM。在 MM 中，收到的消息分别由 do\_exit() 和 do\_wait() 来处理。

假设进程 P 有子进程 A。而 A 调用 exit()，那么 MM 将会：

* 告诉 FS：A 退出，请做相应处理。
* 释放 A 占用的内存。
* 判断 P 是否正在 WAITING。
  + 如果是

∗ 清除 P 的 WAITING 位；

∗ 向 P 发送消息以解除阻塞（到此 P 的 wait() 函数结束）；

∗ 释放 A 的进程表项（到此 A 的 exit() 函数结束）。

**–** 如果否

### ∗ 设置 A 的 HANGING 位。

* 遍历 proc\_table[]，如果发现 A 有子进程 B，那么：
  + 将 Init 进程设置为 B 的父进程（换言之，将 B 过继给 Init）。
  + 判断是否满足 Init 正在 WAITING 且 B 正在 HANGING。

∗ 如果是：

* + - 清除 Init 的 WAITING 位；
    - 向 Init 发送消息以解除阻塞（到此 Init 的 wait() 函数结束）；
    - 释放 B 的进程表项（到此 B 的 exit( ) 函数结束）。

∗ 如果否：

* + - 如果 Init 正在 WAITING 但 B 并没有 HANGING，那么“握手”会在将来 B 调用

exit() 时发生；

* + - 如果 B 正在 HANGING 但 Init 并没有 WAITING，那么“握手”会在将来 Init 调用

wait() 时发生。

如果 P 调用 wait()，那么 MM 将会：

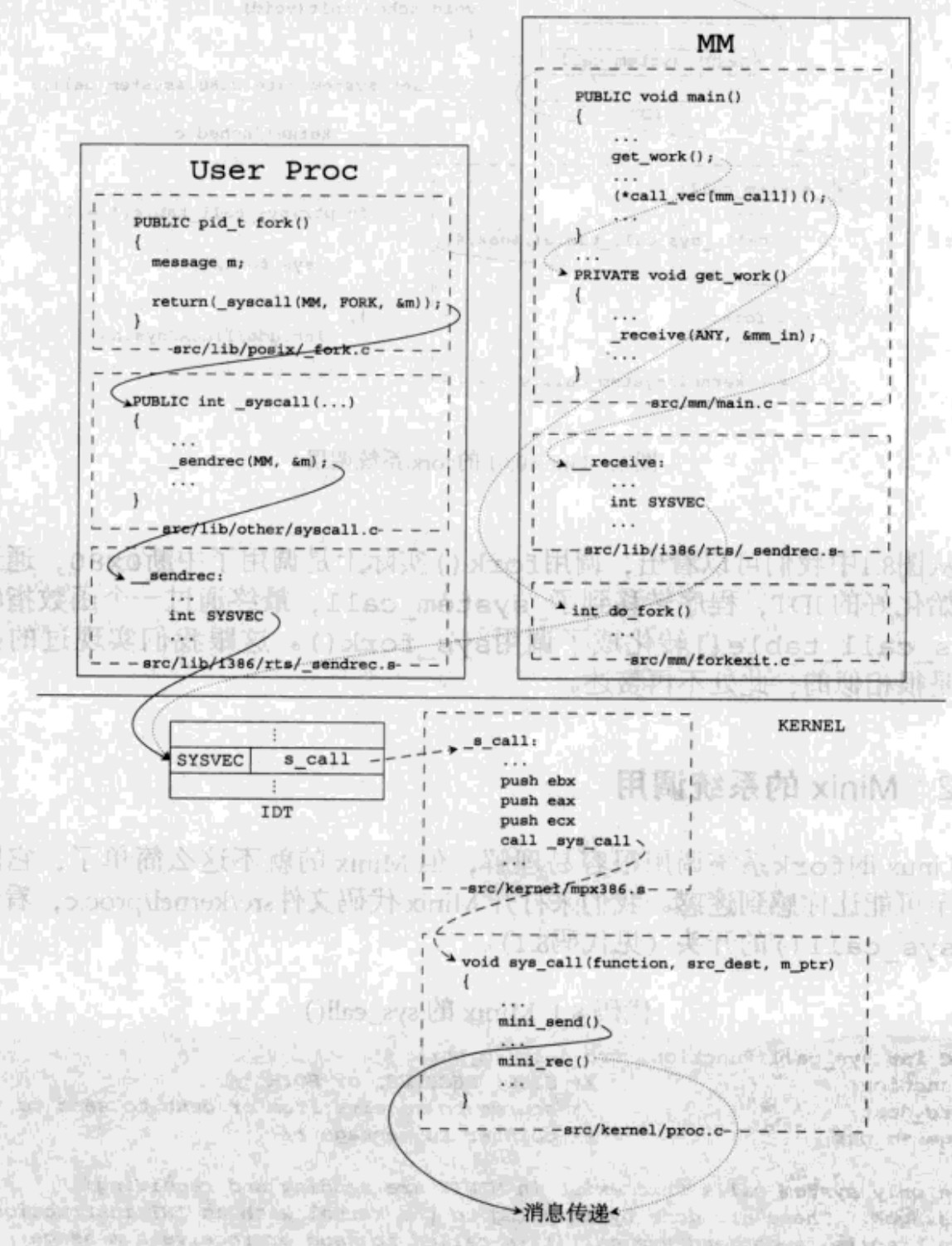
* 遍历 proc\_tabel[]，如果发现 A 是 P 的子进程，并且它正在 HANGING，那么：
  + 向 P 发送消息以解除阻塞（到此 P 的 wait() 函数结束）；
  + 释放 A 的进程表项（到此 A 的 exit() 函数结束）。
* 遍历 proc\_tabel[]，如果发现 A 是 P 的子进程，并且它正在 HANGING，那么：
  + 设 P 的 WAITING 位。
* 如果 P 压根儿没有子进程，则：
  + 向 P 发送消息，消息携带一个表示出错的返回值（到此 P 的 wait() 函数结束）。

## 3 . 4 Part A 任务四

这一个任务属实是炼狱难度了。首先分析当前 oranges 中的内存管理算法，在满足进阶功能的同时，设计实现基于分页的虚拟内存管理算法。oranges 在之前的章节中实现过一个非常简单的分页，但是我们仔细看第十章的代码，发现他并没有使用分页进行管理。

## 3 . 4 . 1 IPC

在回顾 oranges 的内存管理机制前，先简单介绍一下它的 IPC 机制，IPC 机制在后面都非常重要。oranges os 采用的是微内核，不再像 Linux 那样有许许多多的系统调用，在这里系统调用的种类总共有三个，那就 SEND、RECEIVE 和 BOTH。首先看看 Minix 是如何用这三种系统调用实现 fork() 函数的。



用户进程对 fork() 的调用将最终转化成调用内核态的函数 sys\_call()，消息（即图中的 m）的地 址这时已经作为参数被传递进来，sys\_call() 可以据此得知 m 的内容，并在适当的时候将内容传递给MM，MM 的工作其实说起来很简单，它不断地获取并处理消息，所以它能够得到用户进程发送的 m， 并将其存放在 mm\_in 中。当 MM 通过获得的 mm\_in 得知了消息的内容是要进行 fork 操作，它就进一步调用其 do\_fork() 完成整个过程。

因此首先 oranges os 添加了一个系统调用 sendrec。sys\_sendrec 这个函数被设计得相当简单，它可以描述为：把 SEND 消息交给 msg\_send() 处理，把 RECEIVE 消息交给 msg\_receive() 处理。

并且进程表中添加了如下新的字段，所有增加的这些成员都是跟消息机制有关的：

* **p\_ﬂags** 用于标明进程的状态。目前它的取值可以有三种（后续有了 fork 后会增加几种）:
  + **0：**进程正在运行或准备运行。
  + **SENDING：**进程处于发送消息的状态。由于消息还未送达，进程被阻塞。
  + **RECEIVING：**进程处于接收消息的状态。由于消息还未收到，进程被阻塞。
* **p\_msg** 指向消息体的指针。

### ...

假设有进程 A 想要向 B 发送消息 M，那么过程将会是这样的：

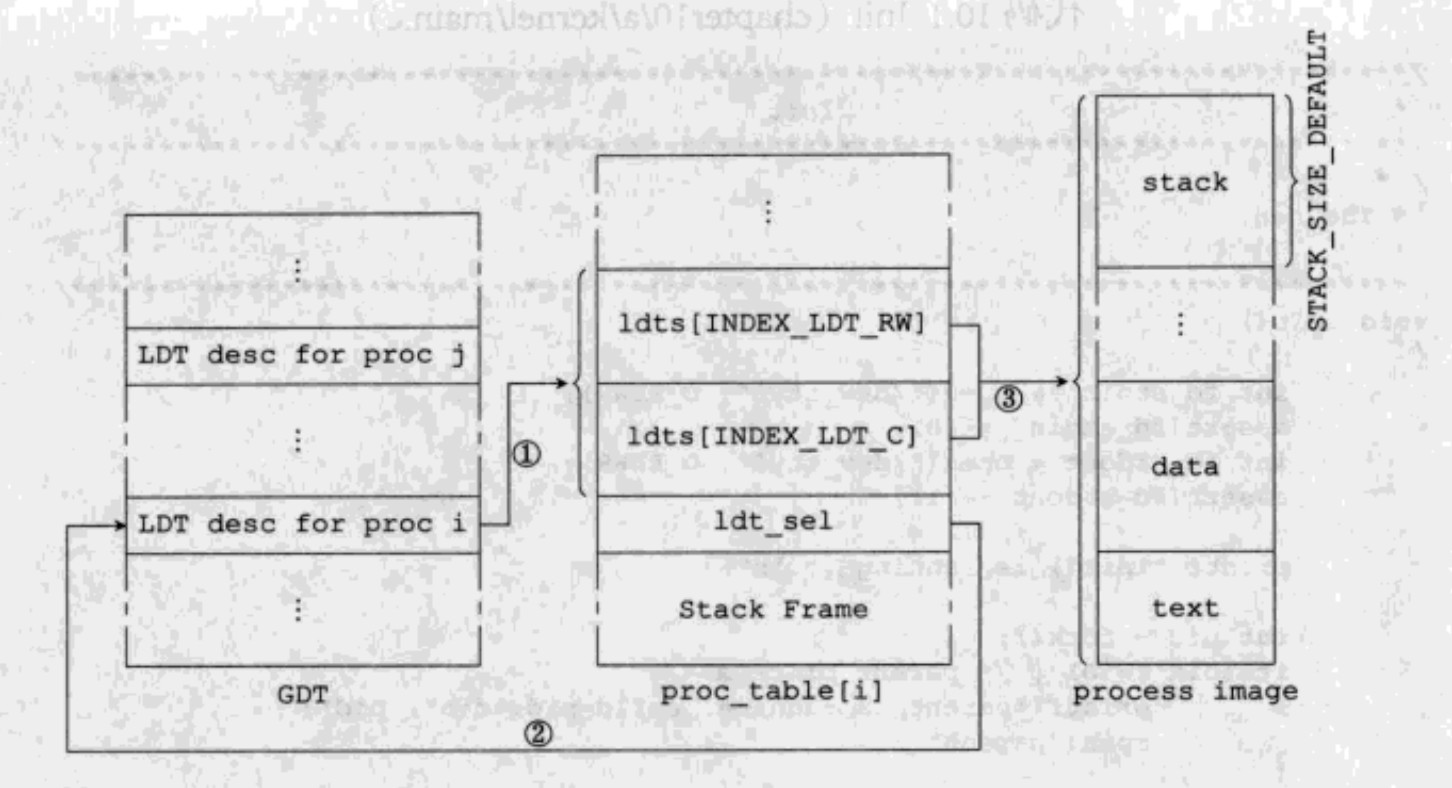
1. A 首先准备好 M。
2. A 通过系统调用 sendrec，最终调用 msg\_send。
3. 简单判断是否发生死锁。
4. 判断目标进程 B 是否正在等待来自 A 的消息：
   * 如果是：消息被复制给 B，B 被解除阻塞，继续运行；
   * 如果否：A 被阻塞，并被加入到 B 的发送队列中。

假设有进程 B 想要接收消息（来自特定进程、中断或者任意进程），那么过程将会是这样的：

1. B 准备一个空的消息结构体 M，用于接收消息。
2. B 通过系统调用 sendrec，最终调用 msg\_receive。
3. 判断 B 是否有个来自硬件的消息（通过 has\_int\_msg），如果是，并且 B 准备接收来自中断的消息或准备接收任意消息，则马上准备一个消息给 B，并返回。
4. 如果 B 想接收来自任意进程的消息，则从自己的发送队列中选取第一个（如果队列非空的话），将其消息复制给 M。
5. 如果 B 是想接收来自特定进程 A 的消息，则先判断 A 是否正在等待向 B 发送消息，若是的话， 将其消息复制给 M。
6. 如果此时没有任何进程发消息给 B，B 会被阻塞。

## 3 . 4 . 2 内存管理机制

在前面几章的实验中，我们知道，分段机制通过 GDT 与 LDT 实现，每个进程有一个 LDT 描述符中包含了段基址，段限长等与分段有关的信息。如下图所示展示了进程，进程表与 GDT，LDT 的关系：



1 ＼

,

随后我们通过 fork() 函数来了解 oranges os 的内存管理机制。fork() 源码如下：

PUBLIC **int** fork()

{

MESSAGE msg;

msg.type = FORK;

send\_recv(BOTH, TASK\_MM, &msg); assert(msg.type == SYSCALL\_RET);

assert(msg.RETVAL == 0);

**return** msg.PID;

1.

}

他通过消息传递机制，发送和接收 TASK\_MM 的消息，并且返回 PID。task\_mm 函数源码如

2

3

4

5

6

7

8

9

10

\_)

11

### 下，

它接收到消息的类型为 FORK，然后调用 do\_fork() 函数。

,

＼

1 PUBLIC **void** task\_mm()

2 {

3 init\_mm();

4

1. **while** (1) {
2. send\_recv(RECEIVE, ANY, &mm\_msg);
3. **int** src = mm\_msg.source;

8

**int** reply = 1;

**int** msgtype = mm\_msg.type;

**switch** (msgtype) {

**case** FORK:

mm\_msg.RETVAL = do\_fork();

**break**; **case** EXIT:

do\_exit(mm\_msg.STATUS); reply = 0;

**break**;

/\* case EXEC: \*/

/\* mm\_msg.RETVAL = do\_exec(); \*/

/\* break; \*/

**case** WAIT: do\_wait();

reply = 0;

**break**; **default**:

dump\_msg("MM::unknown msg", &mm\_msg); assert(0);

**break**;

}

**if** (reply) {

mm\_msg.type = SYSCALL\_RET; send\_recv(SEND, src, &mm\_msg);

}

}

1.

}

do\_fork() 函数内容有些多，其具体分为五部分：

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38 \_)

### 第一部分是分配进程表，从数组 proc\_table[] 中寻找一个空项，用于存放子进程的进程表。接下来将父进程的进程表原原本本地赋给子进程。

* + 第二部分是分配内存。由于子进程是父进程的副本，所以首先需要得到父进程的内存占用情况，这由读取 LDT 来完成。
  + 有了父进程的内存占用情况，就用分配内存的函数为 alloc\_mem() 分配内存。
  + ...

我们仔细去看一下 alloc\_mem() 函数。其中最关键的就是第 22 行，oranges os 简单地只给每个进

程一个 PROC\_IMAGE\_SIZE\_DEFAULT 大小的内存，然后根据 pid 的值按顺序从 PROCS\_BASE

处向上分配内存。

,

＼

1 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2 \* alloc\_mem

3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

4 /\*\*

5 \* Allocate a memory block for a proc.

6 \*

1. \* @param pid Which proc the memory is for.
2. \* @param memsize How many bytes is needed.

9 \*

10 \* @return The base of the memory just allocated.

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12 PUBLIC **int** alloc\_mem(**int** pid, **int** memsize)

13 {

1. assert(pid >= (NR\_TASKS + NR\_NATIVE\_PROCS));
2. **if** (memsize > PROC\_IMAGE\_SIZE\_DEFAULT) {
3. panic("unsupported memory request: %d. "
4. "(should be less than %d)",
5. memsize,
6. PROC\_IMAGE\_SIZE\_DEFAULT);

20 }

21

1. **int** base = PROCS\_BASE +
2. (pid - (NR\_TASKS + NR\_NATIVE\_PROCS)) \* PROC\_IMAGE\_SIZE\_DEFAULT;

24

1. **if** (base + memsize >= memory\_size)
2. panic("memory allocation failed. pid:%d", pid);

27

28 **return** base;

29 1.} \_)

### fork() 后我们还需要通过 exec() 运行新的进程，和 fork 一样也是通过消息传递机制实现，我们直接来看 do\_exec 的源码。最主要的是在 30-44 行代码，这里把新的 elf 文件覆盖原来的内存，其中 va2la 函数就是把虚拟地址转化为线性地址，其实由于没有开启虚拟内存，这里的线性地址等于物理地址。

, ＼

1 PUBLIC **int** do\_exec()

2 {

|  |  |  |
| --- | --- | --- |
| 3 | /\* get parameters from the message | \*/ |
| 4 | **int** name\_len = mm\_msg.NAME\_LEN; /\* | length of filename \*/ |
| 5 | **int** src = mm\_msg.source; /\* caller | proc nr. \*/ |
| 6 | assert(name\_len < MAX\_PATH); |  |
| 7 |  |  |
| 8 | **char** pathname[MAX\_PATH]; |  |

1. phys\_copy((**void**\*)va2la(TASK\_MM, pathname),
2. (**void**\*)va2la(src, mm\_msg.PATHNAME),
3. name\_len);
4. pathname[name\_len] = 0; /\* terminate the string \*/

13

1. /\* get the file size \*/
2. **struct** stat s;
3. **int** ret = stat(pathname, &s);

17 **if** (ret != 0) {

1. printl("{MM} MM::do\_exec()::stat() returns error. %s", pathname);
2. **return** -1;

20 }

21

1. /\* read the file \*/
2. **int** fd = open(pathname, O\_RDWR);

24 **if** (fd == -1)

1. **return** -1;
2. assert(s.st\_size < MMBUF\_SIZE);
3. read(fd, mmbuf, s.st\_size);
4. close(fd);

29

1. /\* overwrite the current proc image with the new one \*/
2. Elf32\_Ehdr\* elf\_hdr = (Elf32\_Ehdr\*)(mmbuf);
3. **int** i;
4. **for** (i = 0; i < elf\_hdr->e\_phnum; i++) {
5. Elf32\_Phdr\* prog\_hdr = (Elf32\_Phdr\*)(mmbuf + elf\_hdr->e\_phoff +
6. (i \* elf\_hdr->e\_phentsize));
7. **if** (prog\_hdr->p\_type == PT\_LOAD) {
8. assert(prog\_hdr->p\_vaddr + prog\_hdr->p\_memsz <
9. PROC\_IMAGE\_SIZE\_DEFAULT);
10. phys\_copy((**void**\*)va2la(src, (**void**\*)prog\_hdr->p\_vaddr),
11. (**void**\*)va2la(TASK\_MM,
12. mmbuf + prog\_hdr->p\_offset),
13. prog\_hdr->p\_filesz);

43 }

44 }

45

1. /\* setup the arg stack \*/
2. **int** orig\_stack\_len = mm\_msg.BUF\_LEN;
3. **char** stackcopy[PROC\_ORIGIN\_STACK];
4. phys\_copy((**void**\*)va2la(TASK\_MM, stackcopy),
5. (**void**\*)va2la(src, mm\_msg.BUF),
6. orig\_stack\_len);

52

u8 \* orig\_stack = (u8\*)(PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK);

**int** delta = (**int**)orig\_stack - (**int**)mm\_msg.BUF;

**int** argc = 0;

**if** (orig\_stack\_len) { /\* has args \*/

**char** \*\*q = (**char**\*\*)stackcopy;

**for** (; \*q != 0; q++,argc++)

\*q += delta;

}

phys\_copy((**void**\*)va2la(src, orig\_stack), (**void**\*)va2la(TASK\_MM, stackcopy), orig\_stack\_len);

proc\_table[src].regs.ecx = argc; /\* argc \*/ proc\_table[src].regs.eax = (u32)orig\_stack; /\* argv \*/

/\* setup eip & esp \*/

proc\_table[src].regs.eip = elf\_hdr->e\_entry; /\* @see \_start.asm \*/ proc\_table[src].regs.esp = PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK;

strcpy(proc\_table[src].name, pathname);

**return** 0;

1.}

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78 \_)

## 3 . 4 . 3 分页实现思路

### 关于分页机制的细节我们不在这里过多赘述，在 x86 中，cpu 支持硬件实现的两级页表，通过 cr3 寄存器存储的页目录基址寻址到页目录，接下来从页目录中查找到相应页表对应的地址，接下来从页表中查找对应的页表项 pte，最后再寻址到物理内存。

我们可以发现，orange’s 虽然启用了操作系统的分页机制并且初始化了页表和页目录，但并没有利用这个分页机制进行内存管理，在 kernel 的代码中，我们可以发现，操作系统在引导过程中在 0x100000 即 64KB 位置建立了连续的页表和页目录，并且将系统的全部内存范围都做了线性映射，在之后操作系统的整个生命周期中，这个页表都是固定的，不会再发生变化，cr3 的值也不会发生变化。

参照 linux 中对于内存管理的实现，在 linux 中，每个进程都会有自己的页表，通过页表而不是分段机制管理自己的可用内存，因此要把分页机制充分利用，我们不止需要一个固定的页表，因此我们需要为每个进程创建自己的页表，并且利用页表来进行进一步的内存映射，例如消除内存空洞等，我们可以为每个进程创建独立的地址映射，并把地址映射保存在进程的页表中，在进程上下文切换时切换页表，这样就可以充分利用操作系统的分页机制，进一步完善内存管理机制。因此我们实现的分页的整体

思路为：

* 进程表的 stackframe 中要添加页目录基址 pde
* 在 exec 中装入 elf 文件的时候，添加一层线性地址到物理地址的映射。
* 还需要在 exec 后面放入页表，该页表和上面的映射相关。

## 3 . 5 Part B 任务一

该任务要编写一个 C 程序，该程序查找 OS 中的可执行文件，对可执行文件添加额外代码；还需要编写一个程序，可对存在内存破坏漏洞的代码进行缓冲区溢出，控制返回地址到指定位置。

## 3 . 5 . 1 ELF 文件注入

shellcode 是一段能够被执行的机器码，其注入方式主要有代码空洞和新增节两种，下面简单介绍两种方式的注入流程：

**代码空洞注入**

* 以可读写方式打开文件;
* 找一个大小合适的 code cave;
* 将 shellcode 嵌入到 code cave 中;
* 修改 entrypoint 指向 shellcode 的开头

**新增节注入** 我们首先介绍 ELF 文件格式：



### 文件开始处是一个 ELF 头部 (ELF Header)，用来描述整个文件的组织。节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等。

程序头部表 (Program Header Table)，如果存在的话，告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区头部表 (Section Heade Table) 包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

注意：尽管图中显示的各个组成部分是有顺序的，实际上除了 *ELF* 头部表以外，其他节区和段都没有规定的顺序

于是，我们 ELF 文件新增节注入思路如下：

* 存储相关原始数据，如原文件入口地址
* 修正 ELF 头部中的 e\_shoﬀ ，增加 PAGESIZE 大小（操作系统页式系统，一页默认 4k）
* 修正第一个程序头部表，第一个头部特殊对待，因为要插入自己注入的程序，所以要把第一个头部扩容，把 p\_ﬁlesz 和 p\_memsz 增加 PAGESIZE 大小或者注入程序的大小
* 修正 ELF 头部中的 e\_entry ，指向 p\_vaddr + p\_ﬁlesz
* 修正程序头部表偏移地址 p\_oﬀset ，增加 PAGESIZE 大小
* 修正节区 sh\_oﬀset ，增加 PAGESIZE 大小
* 修正注入程序机器码，如：修正数据段存储地址（从新的 elfh.e\_entry 开始计算程序首地址），最后要加上跳转指令，即跳转到原来的 e\_entry（刚开始记录的）
* 首先存储原来目标节区头到末尾的数据，然后插入修正后的注入程序的机器码，之后通过插入 0

的方式让插入区块扩充到 PAGESIZE （4k）

* 再把存储的数据接在后面插入

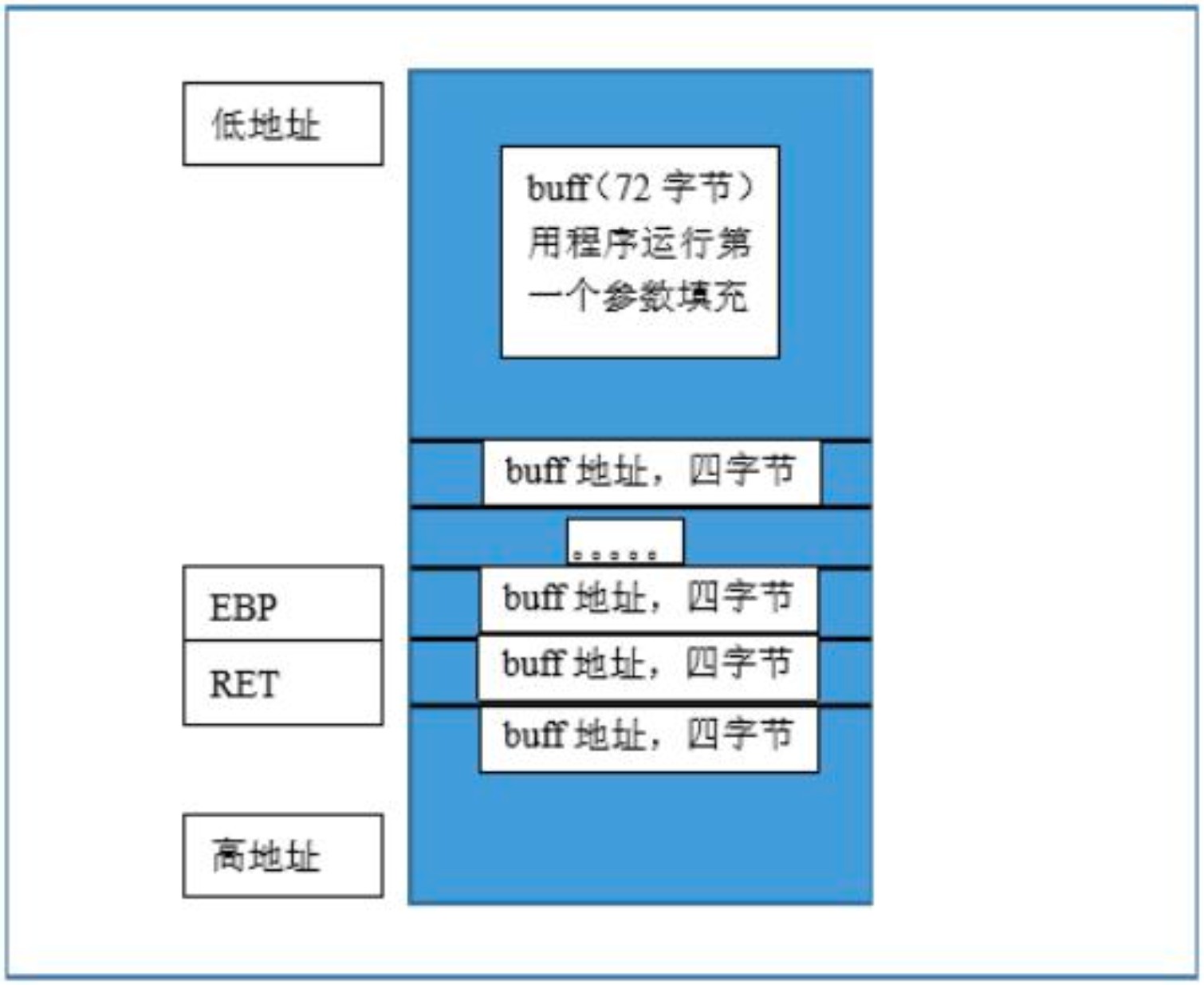
## 3 . 5 . 2 缓冲区溢出与 ROP

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。这种问题是一种特定的缓冲区溢出漏洞，类似的还有堆溢出，bss 段溢出等溢出方式。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。此外，我们也不难发现，发生栈溢出的基本前提是

* 程序必须向栈上写入数据。
* 写入的数据大小没有被良好地控制。在内存中，数据的存放位置如下：



在计算机向缓冲区内填充数据位数时，如果超过了缓冲区本身的容量，溢出的数据会覆盖在合法数据上。那么尝试输入特定数据并覆盖合法数据，执行设计好的程序功能。假设我们定义了一个数组buff，借其进行缓冲区溢出攻击，那么在攻击完成后，最终 buﬀ 地址会覆盖 main 函数的返回地址, 成功覆盖返回地址后，main 函数结束就会跳转到 buff，执行我们的 shellcode。



## 3 . 6 Part B 任务二

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。

因此我们思路如下：

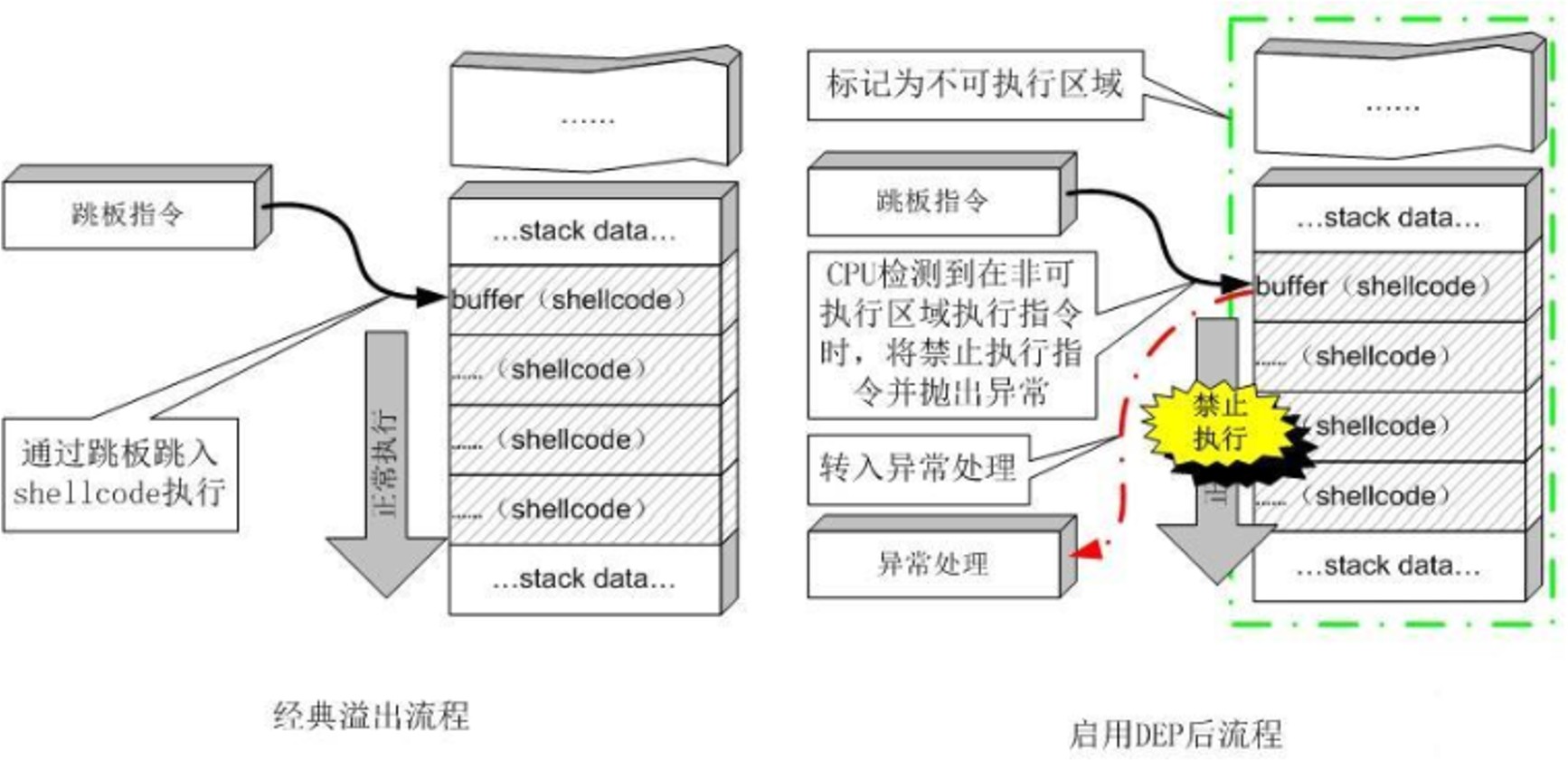
* 在 untar 把应用程序装载到 os 时，计算一个校验码
* 在 exec 运行时，也计算一遍校验码并且和之前的校验码进行比较

如果采用简单的奇偶校验，就不能有效防止对可执行文件的篡改。因为攻击者篡改完可执行文件后，可以计算当前校验码和原始校验码的差别，然后通过修改 elf 文件中一些代码空洞，使得校验码和原始校验码相等。因此我们的校验码计算借助了 AES 加密（在 Part A 任务二中实现了 AES，当时没看到 Part B 任务二，不然肯定会实现 MD5），加大了攻击者篡改难度。我们把 untar 时候的校验码直接保存在内核内存中，因此攻击者其实可以看到校验码，但由于我们利用了更复杂的校验码计算方法， 即使看到校验码，也无法轻易去攻击。

## 3 . 7 Part B 任务三

迄今为止，发生次数最多、最常见的安全漏洞仍是基于栈的缓冲区溢出，对于这种类型漏洞的最常见利用方式是：在栈中精心构造二进制串溢出原有数据结构进而改写函数返回地址，使其跳转到位于栈中的 Shellcode 执行。如果使栈上数据不可执行，那么就可以阻止这种漏洞利用方式的成功实施。而DEP 就是通过使可写内存不可执行或使可执行内存不可写来消除类似的威胁。

DEP - 数据执行保护的缩写，Data Execution Prevention。他是一套软硬件技术，能够在内存上执行额外检查以帮助防止在系统上运行恶意代码。其基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入 shellcode 时，程序会尝试在数据页面上执行指令，此时 CPU 就会抛出异常，而不是去执行恶意指令。



在这次实验中，我们模拟这种思路，对这一模式进行了简化，我们对程序运行时的情况进行判断， 判断是否在栈区运行，如果是则强制退出程序，因此这里的关键在于判断程序是否超出栈区。

在 OrangeOS 中，内核栈与用户的进程栈显然是不同，用户的进程程序主要是由内核中的 Init 进程 fork() 而来，这两者使用的堆栈是不同的，因此，我们在检查是否产生栈溢出时必须对这两种情况分类讨论：

* 在原本的程序 kernel/global.c 和 kernel/main.c 中定义了内核层次程序的堆栈分配情况，在 in- clude/proc.h 中定义了每个堆栈段的大小，从这里可以看出内核的程序初始化时堆栈已经存在具

体的界限为每个程序 16kb，并且栈的大小已经确定，部分代码截取如下：

,

＼

1 /\*================= global.c ======================\*/

1. PUBLIC **struct** task task\_table[NR\_TASKS] = {
2. /\* entry stack size task name \*/

4 /\* \*/

1. {task\_tty, STACK\_SIZE\_TTY, "TTY" },
2. {task\_sys, STACK\_SIZE\_SYS, "SYS" },
3. {task\_hd, STACK\_SIZE\_HD, "HD" },
4. {task\_fs, STACK\_SIZE\_FS, "FS" },
5. {task\_mm, STACK\_SIZE\_MM, "MM" }};

10

1. PUBLIC **struct** task user\_proc\_table[NR\_NATIVE\_PROCS] = {
2. /\* entry stack size proc name \*/

13 /\* \*/

1. {Init, STACK\_SIZE\_INIT, "INIT" },
2. {TestA, STACK\_SIZE\_TESTA, "TestA"},
3. {TestB, STACK\_SIZE\_TESTB, "TestB"},
4. {TestC, STACK\_SIZE\_TESTC, "TestC"}};

18

19 PUBLIC **char** task\_stack[STACK\_SIZE\_TOTAL];

20

21

22 /\*================= main.c ======================\*/

23

24 //将堆栈段的地址定义为 基址 + 大小

25 **char**\* stk = task\_stack + STACK\_SIZE\_TOTAL;

26

27 /\* 中间的进程初始化过程省略 \*/

28 **for** (i = 0; i < NR\_TASKS + NR\_PROCS; i++, p++, t++) {

29

30 /\* 省略LDT和GDT的初始化过程 \*/

1. p->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK) | rpl;
2. p->regs.eip = (u32)t->initial\_eip;

33 p->regs.esp = (u32)stk; //这里将stk的地址赋值给esp

34 p->regs.eflags = eflags;

35

36 p->ticks = p->priority = prio;

37

38 p->p\_flags = 0;

39 p->p\_msg = 0;

1. p->p\_recvfrom = NO\_TASK;
2. p->p\_sendto = NO\_TASK;
3. p->has\_int\_msg = 0;
4. p->q\_sending = 0;
5. p->next\_sending = 0;

45 }

46

47 /\*================= proc.c ======================\*/

1. **#define** STACK\_SIZE\_DEFAULT 0x4000 /\* 16KB \*/
2. **#define** STACK\_SIZE\_TTY STACK\_SIZE\_DEFAULT
3. **#define** STACK\_SIZE\_SYS STACK\_SIZE\_DEFAULT
4. **#define** STACK\_SIZE\_HD STACK\_SIZE\_DEFAULT
5. **#define** STACK\_SIZE\_FS STACK\_SIZE\_DEFAULT
6. **#define** STACK\_SIZE\_MM STACK\_SIZE\_DEFAULT
7. **#define** STACK\_SIZE\_INIT STACK\_SIZE\_DEFAULT
8. **#define** STACK\_SIZE\_TESTA STACK\_SIZE\_DEFAULT
9. **#define** STACK\_SIZE\_TESTB STACK\_SIZE\_DEFAULT
10. 1.**#define** STACK\_SIZE\_TESTC STACK\_SIZE\_DEFAULT \_)

### 而普通用户进程则与上述进程不同，普通用户运行的程序由 exec 函数来执行，堆栈的分布情况定义在 mm/exec.c 中，在这里的 esp 定义为内存总大小减去堆栈初始值，因此对于普通用户程序而言堆栈并没有一个很好的界限，因此我们只能选择一个较大数作为普通程序的堆栈界限代码如下：

, ＼

1 /\*================= exec.c ======================\*/

2 PUBLIC **int** do\_exec()

3 {

4 /\* ................. \*/

5

1. /\* setup eip & esp \*/
2. proc\_table[src].regs.eip = elf\_hdr->e\_entry; /\* @see \_start.asm \*/
3. proc\_table[src].regs.esp = PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK;
4. strcpy(proc\_table[src].name, pathname);
5. **return** 0;

11 }

12

13 /\*================= proc.h ======================\*/

1. **#define** PROCS\_BASE 0xA00000 /\* 10MB \*/
2. **#define** PROC\_IMAGE\_SIZE\_DEFAULT 0x100000 /\* 1MB \*/
3. 1.**#define** PROC\_ORIGIN\_STACK 0x400 /\* 1KB \*/ \_)

# 四、 实验过程分析

## . 1 Part A 任务一

### 首先我们需要给我们进程表添加如下字段，queue 和 pos 表示该进程在第几个队列的第几个位置。

ft、st 和 tt 分别表示在第一、二和三队列剩余的时间片。in\_ticks、out\_ticks 和 resp\_ticks 是为了输

出性能评价，分别表示提交时刻、结束时刻和响应时刻。

,

＼

1. **typedef struct** s\_proc {
2. STACK\_FRAME regs; /\* process registers saved in stack frame \*/

3

1. u16 ldt\_sel; /\* gdt selector giving ldt base and limit \*/
2. DESCRIPTOR ldts[LDT\_SIZE]; /\* local descriptors for code and data \*/

6

1. **int** ticks; /\* remained ticks \*/
2. **int** priority;
3. **int** queue;
4. **int** pos;

11

1. **int** ft;
2. **int** st;
3. **int** tt;

15

**int** in\_ticks; **int** out\_ticks; **int** resp\_ticks;

u32 pid; /\* process id passed in from MM \*/

**char** p\_name[16]; /\* name of the process \*/

}PROCESS;

1.

我们还需要在 global.h 中定义如下字段。其中 ﬁrst\_num、second\_num 和 third\_num 表示每个

16

17

18

19

20

21

22 \_)

队

### 列当前进程的个数，定义这些字段可以更好对队列进行操作。ﬁrst\_ticks、second\_ticks 和 third\_ticks

指的是在每个队列的时间片。剩下两个是为了更好调试以及更好输出性能评价信息，再之后用到的时

候在进行解释。

,

＼

1. EXTERN **int** first\_num;
2. EXTERN **int** second\_num;
3. EXTERN **int** third\_num;
4. EXTERN **int** first\_ticks;
5. EXTERN **int** second\_ticks;
6. EXTERN **int** third\_ticks;

7

1. EXTERN **int** dbg\_disp\_time;
2. 1.EXTERN **int** finish\_proc\_num; \_)

### 上面的常数和进程的一些新的字段都需要在下面进行初始化。42-48 行初始化进程都在第一队列， 并且在各个队列的剩余时间片都是满的，也就是 10、20 和 30（在 68-70 行初始化了每个队列有多少时间片）。并且定义了任务提交时间都是 0 时刻，也就是 os 刚启动的时刻。并且初始化响应时间为-1，这样如果后面处理了任务，但是发现响应时间还是-1，那就可以把当前时刻赋予给响应时间字段。56-62

行定义了一个任务的总时间片，64-66 初始化了每个队列当前有多少任务。

,

＼

1 /\*======================================================================\*

2 kernel\_main

3 \*======================================================================\*/

4 PUBLIC **int** kernel\_main()

5 {

6 disp\_str("-----\"kernel\_main\" begins \n");

7

1. TASK\* p\_task = task\_table;
2. PROCESS\* p\_proc = proc\_table;
3. **char**\* p\_task\_stack = task\_stack + STACK\_SIZE\_TOTAL;
4. u16 selector\_ldt = SELECTOR\_LDT\_FIRST;
5. **int** i;
6. **for** (i = 0; i < NR\_TASKS; i++) {
7. strcpy(p\_proc->p\_name, p\_task->name); // name of the process

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 15  16  17  18  19  20  21  22  23  24  25 | p\_proc->pid = i; // pid  p\_proc->ldt\_sel = selector\_ldt;  memcpy(&p\_proc->ldts[0], &gdt[SELECTOR\_KERNEL\_CS »3],  **sizeof**(DESCRIPTOR));  p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_TASK «5; memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS »3],  **sizeof**(DESCRIPTOR));  p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_TASK «5; p\_proc->regs.cs = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK) | | | | | |
| 26 | | SA\_TIL | RPL\_TASK; |  |  |  |  |  |
| 27 | p\_proc->regs.ds = ((8 \* | 1) | & | SA\_RPL\_MASK | & | SA\_TI\_MASK) |
| 28 | | SA\_TIL | RPL\_TASK; |  |  |  |  |  |
| 29 | p\_proc->regs.es = ((8 \* | 1) | & | SA\_RPL\_MASK | & | SA\_TI\_MASK) |
| 30 | | SA\_TIL | RPL\_TASK; |  |  |  |  |  |
| 31 | p\_proc->regs.fs = ((8 \* | 1) | & | SA\_RPL\_MASK | & | SA\_TI\_MASK) |
| 32 | | SA\_TIL | RPL\_TASK; |  |  |  |  |  |
| 33 | p\_proc->regs.ss = ((8 \* | 1) | & | SA\_RPL\_MASK | & | SA\_TI\_MASK) |
| 34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | | SA\_TIL | RPL\_TASK;  p\_proc->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK)  | RPL\_TASK;  p\_proc->regs.eip = (u32)p\_task->initial\_eip; p\_proc->regs.esp = (u32)p\_task\_stack;  p\_proc->regs.eflags = 0x1202; /\* IF=1, IOPL=1 \*/  p\_proc->queue = 1; p\_proc->pos = i; p\_proc->ft = 10;  p\_proc->st = 20;  p\_proc->tt = 30;  p\_proc->in\_ticks = 0;  p\_proc->resp\_ticks = -1;  p\_task\_stack -= p\_task->stacksize; p\_proc++;  p\_task++; selector\_ldt += 1«3;  }  proc\_table[0].ticks = proc\_table[0].priority = 150; proc\_table[1].ticks = proc\_table[1].priority = 120; | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 58 | proc\_table[2].ticks | = | proc\_table[2].priority | = | 130; |
| 59 | proc\_table[3].ticks | = | proc\_table[3].priority | = | 110; |
| 60 | proc\_table[4].ticks | = | proc\_table[4].priority | = | 100; |
| 61 | proc\_table[5].ticks | = | proc\_table[5].priority | = | 80; |
| 62 | proc\_table[6].ticks | = | proc\_table[6].priority | = | 160; |
| 63 |  |  | | | |
| 64 | first\_num = 7; |
| 65 | second\_num = 0; |
| 66 | third\_num = 0; |
| 67 |  |
| 68 | first\_ticks = 10; |
| 69 | second\_ticks = 20; |
| 70 | third\_ticks = 30; |
| 71 |  |
| 72 | k\_reenter = 0; |
| 73 | ticks = 0; |
| 74 |  |

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99 1.}

p\_proc\_ready = proc\_table;

dbg\_disp\_time = 0;

finish\_proc\_num = 0;

disp\_pos = 0;

**for** (i = 0; i < 80\* 25; i++) { disp\_str(" ");

}

disp\_pos = 0; disp\_queue();

/\* 初 始 化 8253PIT \*/ out\_byte(TIMER\_MODE, RATE\_GENERATOR); out\_byte(TIMER0, (u8) (TIMER\_FREQ/HZ) );

out\_byte(TIMER0, (u8) ((TIMER\_FREQ/HZ) »8));

put\_irq\_handler(CLOCK\_IRQ, clock\_handler); /\* 设定时钟中断处理程序 \*/ enable\_irq(CLOCK\_IRQ); /\* 让8259A可以接收时钟中断 \*/

restart();

**while**(1){}

#### \_)

所有准备工作做完后，我们开始利用 cloker\_handler 和 schedule 两个函数来完成多级反馈队列。首先每次产生时钟中断，都要首先看看当前进程的响应时间是否被赋值，如果没有就把当前时刻设为 响应时间。然后 ticks++，k\_reenter 解决中断重入问题。随后

* + 判断当前进程在当前它所处的队列是否还有时间片，如果没有那么就要调用 schedule 函数进行调度。
  + 否则再判断一下当前进程的总时间片是否用完了，如果用完了也要用 schudle 函数进行调度。
  + 如果不满足上面两个情况，那么就让这个进程在当前队列的剩余时间片和总时间片都减 1。

,PUBLIC **void** clock\_handler(**int** irq)

{

**if** (p\_proc\_ready->resp\_ticks == -1) p\_proc\_ready->resp\_ticks = ticks; ticks++;

**if** (k\_reenter != 0) {

**return**;

}

**if** (p\_proc\_ready->ft > 0&& p\_proc\_ready->queue == 1) {

**if** (p\_proc\_ready->ticks > 0) { p\_proc\_ready->ft--; p\_proc\_ready->ticks--; **return**;

}

}

**if** (p\_proc\_ready->st > 0&& p\_proc\_ready->queue == 2) {

**if** (p\_proc\_ready->ticks > 0) { p\_proc\_ready->st--; p\_proc\_ready->ticks--; **return**;

}

}

**if** (p\_proc\_ready->tt > 0&& p\_proc\_ready->queue == 3) {

**if** (p\_proc\_ready->ticks > 0) { p\_proc\_ready->tt--; p\_proc\_ready->ticks--; **return**;

}

}

1 ＼

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

schedule();

1.

}

调用了 schedule 函数，就说明总时间片或者在当前的队列的时间片用完了。于是我们先要处

\_)

35

### 理这

种情况，如果总时间片用完了，那么这个进程就需要被删除，同时记录该进程结束时刻到 out\_ticks 字段中（ﬁnish\_pro\_num 是为了更好打印性能评价信息，记录了当前有几个进程结束了，如果所有进程结束了，那么他就会按顺序打印性能评价信息，在 7-25 行）。如果不是总时间片用完了，那么就是在当

前队列的时间片用完了。在 1 和 2 队列的进程处理方式一致，都是先删除，然后我们再把进程放入下一个队列的末尾，放入末尾后，那个队列的进程数量也要加一。在第三个队列处理方式不同，它要把进程删除后重新放入第 3 个队列，并且还要重新把当前进程在第 3 队列的剩余时间片恢复。

处理完该进程后，我们要选择下一个运行的进程，方式就是从 1 队列到 3 队列、从头往后找，找

到第一个进程，并且把当前运行进程 p\_proc\_ready 指向它。

,

＼

1 PUBLIC **void** schedule()

2 {

1. **if** (p\_proc\_ready->ticks == 0) {
2. delete\_proc();
3. p\_proc\_ready->out\_ticks = get\_ticks();
4. finish\_proc\_num++;
5. **if** (finish\_proc\_num == NR\_TASKS) {
6. disp\_str(" PROC\_NAME | submit | finish | response | waiting |\n");
7. **int** i = 1;
8. **for** (PROCESS\* p = proc\_table; p < proc\_table + NR\_TASKS; p++) {
9. disp\_str("PROCESS ");
10. disp\_int(i);

13 i++;

1. disp\_str("| ");
2. disp\_int(p->in\_ticks);
3. disp\_str(" | ");
4. disp\_int(p->out\_ticks);
5. disp\_str(" | ");
6. disp\_int(p->resp\_ticks);

20 **if** (i == 2) disp\_str(" ");

1. disp\_str(" | ");
2. disp\_int(p->out\_ticks - p->priority);
3. disp\_str(" |\n");

24 }

25 }

1. // add\_proc();
2. } **else if** (p\_proc\_ready->queue == 1) {
3. delete\_proc();
4. p\_proc\_ready->queue = 2;
5. p\_proc\_ready->pos = second\_num;

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71 1.}

second\_num++;

} **else if** (p\_proc\_ready->queue == 2) { delete\_proc();

p\_proc\_ready->queue = 3; p\_proc\_ready->pos = third\_num; third\_num++;

} **else** { // if (p\_proc\_ready->queue == 3) delete\_proc();

p\_proc\_ready->queue = 3; p\_proc\_ready->pos = third\_num; third\_num++;

p\_proc\_ready->tt = third\_ticks;

}

**if** (finish\_proc\_num < NR\_TASKS) { disp\_queue();

}

// find next PROCESS\* p;

**for** (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {

**if** (p->pos == 0&& p->queue == 1) { p\_proc\_ready = p;

**return**;

}

}

**for** (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {

**if** (p->pos == 0&& p->queue == 2) { p\_proc\_ready = p;

**return**;

}

}

**for** (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {

**if** (p->pos == 0&& p->queue == 3) { p\_proc\_ready = p;

**return**;

}

}

#### \_)

上面多次提到了删除进程的函数，其具体实现如下所示。首先当前进程所在队列的进程个数要减

1，然后把当前进程的队列字段改为 0。随后我们还需要遍历所有进程，把和删除进程所在相同队列的

进程的位置都减 1，也就是往前挪 1 位。

,

＼

1. PRIVATE **void** delete\_proc() {
2. **int** queue = p\_proc\_ready->queue;
3. **if** (queue == 1) {
4. first\_num--;
5. } **else if** (queue == 2) {
6. second\_num--;
7. } **else** { // if (queue == 3)
8. third\_num--;

9 }

10 p\_proc\_ready->queue = 0;

11

1. PROCESS\* p;
2. **for** (p = proc\_table; p < proc\_table + NR\_TASKS; p++) {
3. **if** (p->queue == queue) p->pos--;

15 }

16 1.} \_)

## 4 . 2 Part A 任务二

### 任务二主要就是在 command 文件夹中编写，编译修改 makeﬁle 后，编译生成可执行文件并且压缩成 tar 文件。os 启动时会自动解压 tar 文件。

这里我们添加了三个应用程序，第一个和第二个是密码学算法 DES 和 AES，第三个是和进程通信结合起来的模仿 linux 中 ps 的实现。

DES 代码过长（主要很多都是做好的表），这里只展示 DES 核心的加解密的实现。首先利用 keyGen 生成轮密钥。DES\_PT 进行初始置换。在 16 轮的加解密过程中，每一次通过 row 和 col 找到 S-box 中的值作为输出，随后左右部分进行交换。由于 DES 是对合的，加解密只是利用轮密钥的顺序不同，因

此 32-38 行判断是加密还是解密。

,

＼

1 /\*

1. \* The DES function
2. \* plaintext: 64bits message
3. \* key: 64 bits key for encryption/decryption
4. \* mode: ’e’ = encryption, ’d’ = decryption

6 \*/

1. u64 DES(u64 plaintext, u64 key, **char** mode) {
2. KEY\_PD(key, keyPD)

9 u64 sub\_key[16] = {0};

10 keyGen(keyPD, sub\_key);

11

12 DES\_PT(plaintext, IP, init\_perm\_res)

|  |  |  |
| --- | --- | --- |
| 13  14  15  16  17  18  19 | /\* Decompose T64 into L and R parts \*/ u32 L = 0;  u32 R = 0;  L = (u32)(init\_perm\_res »32) & L64\_MASK; R = (u32)(init\_perm\_res)&L64\_MASK;  **for** (**int** i = 0; i < 16; i++) { | |
| 20 | /\* f(R,k) function \*/ |  |
| 21 | /\* expansion 32bits R -> 48bits s\_input \*/ |  |
| 22 | u64 s\_input = 0; |  |
| 23 | **for** (**int** j = 0; j < 48; j++) { |  |
| 24 | s\_input «= 1; |  |
| 25 | s\_input |= (u64)((R »(32 - EDB[j])) & LB32\_MASK); |  |
| 26 | } |  |
| 27 |  |  |
| 28 | /\* |  |
| 29 | \* Encryption/Decryption |  |
| 30 | \* XORing expanded Ri with Ki |  |
| 31 | \*/ |  |
| 32 | **if** (mode == ’d’) { |  |
| 33 | // decryption |  |
| 34 | s\_input = s\_input ^ sub\_key[15 - i]; |  |
| 35 | } **else** { |  |
| 36 | // encryption |  |
| 37 | s\_input = s\_input ^ sub\_key[i]; |  |
| 38 | } |  |
| 39 |  |  |
| 40 | u8 row, col; |  |
| 41 | u32 s\_output = 0; |  |
| 42 | /\* S-Box Tables \*/ |  |
| 43 | **for** (**int** j = 0; j < 8; j++) { |  |
| 44 | row = (**char**)((s\_input & (0x0000840000000000 »6\* j)) | »(42 - 6\* j)); |
| 45 | row = (row »4) | row & 0x01; |  |
| 46 | col = (**char**)((s\_input & (0x0000780000000000 »6\* j)) | »(43 - 6\* j)); |
| 47 |  |  |
| 48 | s\_output «= 4; |  |
| 49 | s\_output |= (u32)(SB[j][16 \* row + col] & 0x0f); |  |
| 50 | } |  |
| 51 |  |  |
| 52 | /\* post S-box permutation \*/ |  |
| 53 | u32 f\_function\_res = 0; |  |
| 54 | **for** (**int** j = 0; j < 32; j++) { |  |
| 55 | f\_function\_res «= 1; |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 56 |  | f\_function\_res |= (s\_output | »(32 - SBP[j])) & LB32\_MASK; |
| 57 |  | } |  |
| 58 |  |  |  |
| 59 |  | /\* Xor \*/ |  |
| 60 |  | L ^= f\_function\_res; |  |
| 61 |  |  |  |
| 62 |  | /\* exchange \*/ |  |
| 63 |  | L ^= R; |  |
| 64 |  | R ^= L; |  |
| 65 |  | L ^= R; |  |
| 66 | } |  |  |
| 67 |  |  |  |

68

u64 pre\_output = (((u64)R) «32) | (u64)L; DES\_PT(pre\_output, PI, ct)

**return** ct;

1.

}

AES 的代码也比较长，在这里我们考虑了 AES 的速度必须得够快，这样在后面作为校验码

69

70

\_)

71

### 的时

候，才能快速计算校验和。在网上也有很多做四个表的实现原理的介绍，具体可以参考[该博客](https://zhuanlan.zhihu.com/p/42264499)。我们通过仔细的构造也把 AES 加解密做成了伪对合的，加解密都在同一个函数，具体实现如下。简单的来说，就是加解密用的四张表是不一样的，那么我们判断完 mode 后就可以直接用指针可以指向加解密不同的表。并且轮密钥使用顺序也不一样，在 18 和 28 行有所体现。后面每一轮加解密都是进行查表操作，查表操作中 j、pn、tot 等变量都是控制加解密顺序的，讲解较为费劲，这里不再赘述。在编译器为 clang version 13.0.0，Target 为 arm64-apple-darwin21.1.0 的情况下，该 AES 速度达到了 400Mb/s，

为后续能够快速计算校验和和快速检验并且运行程序打下了基础。

,

＼

1. **static void** \_aes(u8\* out, u8\* in, AesKeySched\_t rk, **char** mode) {
2. **int** pn, tot, d;
3. **const** u32 \*AES\_TB0, \*AES\_TB1, \*AES\_TB2, \*AES\_TB3;
4. **const** u8\* AES\_SB;

5

|  |  |  |
| --- | --- | --- |
| 6 | u8 | state[Nk \* Nb]; |
| 7 | \_copy(state, **sizeof**(state), in, **sizeof**(state)); | |
| 8 |  |  |
| 9 | **if** | (mode == ’e’) { |
| 10 |  | pn = 1; |
| 11 |  | tot = 0; |
| 12 |  | d = 0; |
| 13 |  | AES\_TB0 = FT0; |
| 14 |  | AES\_TB1 = FT1; |
| 15 |  | AES\_TB2 = FT2; |
| 16 |  | AES\_TB3 = FT3; |
| 17 |  | AES\_SB = FSb; |

1. add\_round\_key(state, rk->words + 0);
2. } **else if** (mode == ’d’) {

20 pn = -1;

21 tot = Nb \* (Nr + 1);

22 d = 1;

1. AES\_TB0 = RT0;
2. AES\_TB1 = RT1;
3. AES\_TB2 = RT2;
4. AES\_TB3 = RT3;
5. AES\_SB = RSb;
6. add\_round\_key(state, rk->words + 40);

29 }

30

31 u8 t[Nk \* Nb] = {0};

32 **for** (**int** i = 0; i < Nr - 1; i++) {

1. **for** (**int** j = 0; j < Nb; j++) {
2. u32 temp = AES\_TB0[state[0 + ((j + pn \* c0 + Nb) % Nb) \* 4]] ^ AES\_TB1[state[1 + ((j + pn \* c1 + Nb) % Nb) \* 4]] ^

35 AES\_TB2[state[2 + ((j + pn \* c2 + Nb) % Nb) \* 4]] ^ AES\_TB3[state[3 + ((j + pn \* c3 + Nb) % Nb) \* 4]] ^

36 rk->words[tot + pn \* (i + 1+ d) \* 4+ j];

37

38 temp = ((temp & 0xFFFF0000) »16) | ((temp & 0x0000FFFF) «16);

39 temp = ((temp & 0xFF00FF00) »8) | ((temp & 0x00FF00FF) «8);

40

41 \*(u32\*)(t + j \* 4) = temp;

42 }

43 \_copy(state, **sizeof**(state), t, **sizeof**(state));

44 }

45

46 **for** (**int** j = 0; j < Nb; j++) {

47 t[4 \* j + 0] = AES\_SB[state[0 + ((j + pn \* c0 + Nb) % Nb) \* 4]] ^ ((u8)(rk->words[tot + pn \* (Nb \* (Nr + d)) + j] »24));

48 t[4 \* j + 1] = AES\_SB[state[1 + ((j + pn \* c1 + Nb) % Nb) \* 4]] ^ ((u8)(rk->words[tot + pn \* (Nb \* (Nr + d)) + j] »16));

49 t[4 \* j + 2] = AES\_SB[state[2 + ((j + pn \* c2 + Nb) % Nb) \* 4]] ^ ((u8)(rk->words[tot + pn \* (Nb \* (Nr + d)) + j] »8));

50 t[4 \* j + 3] = AES\_SB[state[3 + ((j + pn \* c3 + Nb) % Nb) \* 4]] ^ ((u8)(rk->words[tot + pn \* (Nb \* (Nr + d)) + j] »0));

51

52

53 1.}

}

\_copy(out, **sizeof**(state), t, **sizeof**(state));

#### \_)

最后一个程序模仿了 linux 的 ps 命令，在理解了 3.4.1（也就是 IPC 机制）后，理解起来也比较简单。首先我们创建了一个消息，消息 type 为 GET\_PROC\_INFO，这个是我们新建的一个消息类型，

然后向 TASK\_SYS 发送和接收消息（BOTH）。接收到进程信息后输出进程信息。

,

＼

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

22 }

23 }

24 }

25 **return** 0;

26 1.} \_)

### 在 task\_sys 中，我们新增了该消息类型（同时要在 const.h 中新增加该枚举类型），做的事情就是

传入一个待放置进程体的指针和对应的 pid，将 proc\_table[pid] 对应的进程地址复制过去。

,

＼

1 PUBLIC **void** task\_sys()

2 {

1. MESSAGE msg;
2. **struct** time t;

5

1. **while** (1) {
2. send\_recv(RECEIVE, ANY, &msg);
3. **int** src = msg.source;

9

10 **switch** (msg.type) {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | |  | **case** GET\_TICKS: |  |
| 12 | |  | msg.RETVAL = ticks; |  |
| 13 | |  | send\_recv(SEND, src, &msg); |  |
| 14 | |  | **break**; |  |
| 15 | |  | **case** GET\_PID: |  |
| 16 | |  | msg.type = SYSCALL\_RET; |  |
| 17 | |  | msg.PID = src; |  |
| 18 | |  | send\_recv(SEND, src, &msg); |  |
| 19 | |  | **break**; |  |
| 20 | |  | **case** GET\_RTC\_TIME: |  |
| 21 | |  | msg.type = SYSCALL\_RET; |  |
| 22 | |  | get\_rtc\_time(&t); |  |
| 23 | |  | phys\_copy(va2la(src, msg.BUF), |  |
| 24 | |  | va2la(TASK\_SYS, &t), |  |
| 25 | |  | **sizeof**(t)); |  |
| 26 | |  | send\_recv(SEND, src, &msg); |  |
| 27 | |  | **break**; |  |
| 28 | |  | **case** GET\_PROC\_INFO: |  |
| 29 | |  | msg.type = SYSCALL\_RET; |  |
| 30 | |  | phys\_copy(va2la(src, msg.BUF), |  |
| 31 | |  | va2la(TASK\_SYS, &proc\_table[msg.PID]), |  |
| 32 | |  | **sizeof**(**struct** proc)); |  |
| 33 | |  | send\_recv(SEND, src, &msg); |  |
| 34 | |  | **break**; |  |
| 35 | |  | **default**: |  |
| 36 | |  | panic("unknown msg type"); |  |
| 37 | |  | **break**; |  |
| 38 | |  | } |  |
| 39 | | } |  |  |
| 40 | 1.} | \_) | | |

## 4 . 3 Part A 任务三

### 这一部分要支持多任务运行，最朴素的想法就是 fork 多个子进程，然后子进程去运行那些命令。这

里我们直接看代码注释（/\* \*/中间的注释）说话。

,

＼

1. **#define** MAX\_SHELL\_PROC 4
2. **#define** MAX\_SHELL\_PROC\_STACK 128
3. **void** shabby\_shell(**const char**\* tty\_name) {
4. **int** fd\_stdin = open(tty\_name, O\_RDWR);
5. assert(fd\_stdin == 0);
6. **int** fd\_stdout = open(tty\_name, O\_RDWR);
7. assert(fd\_stdout == 1);

8

9 **char** rdbuf[128];

10

11 **while** (1) {

12 write(1, "$ ", 2);

1. **int** r = read(0, rdbuf, 70);
2. rdbuf[r] = 0;

15

1. **int** argc = 0;
2. **char**\* argv[PROC\_ORIGIN\_STACK];
3. **char**\* p = rdbuf;
4. **char**\* s;
5. **int** word = 0;
6. **char** ch;
7. **do** {
8. ch = \*p;

24 **if** (\*p != ’ ’ && \*p != 0&& !word) {

1. s = p;
2. word = 1;

27 }

28 **if** ((\*p == ’ ’ || \*p == 0) && word) {

1. word = 0;
2. argv[argc++] = s;

31 \*p = 0;

32 }

33 p++;

1. } **while** (ch);
2. argv[argc] = 0;

36 /\* 上面这一部分和作者还是一样的，0用来标志命令结束

37 \* 定义多个命令用&分开后，argv数组可能是这样的

38 \* {echo, hello, world, &, pwd, &, aes, -e, -m, 0x1234, -k, 0x5678, 0}

39 \*/

40

41 /\* 我们利用multi\_argv保存二维字符串数组

42 \* multi\_argv = {{echo, hello, world, 0}

43 \* {pwd, 0}

44 \* {aes, -e, -m, 0x1234, -k, 0x5678, 0}}

45 \*/

46 **char**\* multi\_argv[MAX\_SHELL\_PROC][MAX\_SHELL\_PROC\_STACK];

47 /\* num\_proc表示有多少个命令 \*/

48 **int** num\_proc = 1;

49 /\* sec\_count和上面argc的作用类似 \*/

50 **int** sec\_count = 0;

51 /\* 标记命令是否出错了 \*/

52 **int** error = 0;

53 /\* 开始顺序扫描argv数组 \*/

1. **for** (**int** i = 0; i < argc; i++) {
2. **if** (strcmp(argv[i], "&")) {

56 /\* 如果遇到的不是&，那么把字符串放入数组 \*/

1. multi\_argv[num\_proc - 1][sec\_count++] = argv[i];
2. } **else** {

59 /\* 并且还要用0表示该命令结束 \*/

60 multi\_argv[num\_proc - 1][sec\_count] = 0;

61 /\* 任务数量+1，并且要让sec\_count重新指向0 \*/

1. num\_proc++;
2. sec\_count = 0;
3. **if** (num\_proc > MAX\_SHELL\_PROC) {

65 /\* 如果任务数量大于定义的最大的任务数，那么error置1 \*/

1. error = 1;
2. printf("Too many commands!\n");

68 }

69 }

70 }

71

72 /\* 没有错误才会执行，出错直接跳过 \*/

73 **if** (!error) {

74 /\* 这个是父进程保留的子进程的pid数组，为了保证同步

75 \* 继续往下看可以理解其含义

76 \*/

77 **int** pres\_pid[num\_proc];

78

79 **int** pid = -1;

80

81 // FINISHED: 命令出错处理

82 /\* 这一段代码是为了判断哪些命令是否都有效，只要有一个无效就不会执行 \*/

1. **int** err\_cmd = 0;
2. **for** (**int** i = 0; i < num\_proc; i++) {
3. **int** fd = open(multi\_argv[i][0], O\_RDWR);

86 **if** (fd == -1) {

1. err\_cmd = 1;
2. **break**;

89 }

90 close(fd);

91 }

92

93 **int** i;

|  |  |
| --- | --- |
| 94  95  96  97  98  99  100  101  102  103  104  105  106  107  108 | **if** (err\_cmd) { write(1, "{", 1); write(1, rdbuf, r); write(1, "}\n", 2);  } **else** {  /\* 这一段代码要做到，所有进程都是由一个父进程fork出来的 \*/  **for** (i = 0; i < num\_proc; i++) {  /\* 父进程循环fork子进程 \*/ pid = fork();  /\* 如果是子进程就退出循环，子进程不要进行fork \*/  **if** (pid == 0) **break**; // child exit for  /\* 父进程保留子进程的pid \*/ pres\_pid[i] = pid;  /\* 随后父进程再次进入for循环fork子进程 \*/  } |
| 109 | } |
| 110 | // FINISHED: 一个同步机制 |
| 111 | /\* 但是上面代码不做处理还会出现问题 |
| 112 | \* 由于调度机制，子进程可能抢占了父进程导致 |
| 113 | \* 运气好不会出什么事，但在我们多次执行过程 |
| 114 | \* 出现了死锁的情况 |
| 115 | \* echo hello & pwd后，出现了先输出hello，然后输出$/的情况 |
| 116 | \* 正常来说应该是 |
| 117 | \* hello |
| 118 | \* / （或者hello和/反过来） |
| 119 | \* $ （这里继续输入命令） |
| 120 | \*/ |
| 121 | **if** (pid != 0&& !err\_cmd) { /\* parent \*/ |
| 122 | /\* 父进程运行到这里就说明fork子进程那一步完成了 |
| 123 | \* 那么就要遇历保留的子进程pid数组，将他们解除阻塞 |
| 124  125 | * 但由于子进程可能没来得及自我阻塞，所以用while循环进行同步 * 也就是子进程必须阻塞后，父进程才能解除阻塞，否则又会出现非预期结果 |
| 126 | \*/ |
| 127 | **for** (**int** i = 0; i < num\_proc; i++) { |
| 128 | **while**((&FIRST\_PROC + pres\_pid[i])->p\_flags != 1) {}; |
| 129 | (&FIRST\_PROC + pres\_pid[i])->p\_flags = 0; |
| 130 | unblock(&FIRST\_PROC + pres\_pid[i]); |
| 131 | } |
| 132 |  |
| 133 | /\* 解除完所有子进程的阻塞状态后，就开始wait |
| 134 | \* 每个子进程都应该wait一次，不然无法释放完 |
| 135 | \*/ |
| 136 | **for** (**int** i = 0; i < num\_proc; i++) { |

137

**int** s; wait(&s);

}

} **else if** (pid == 0) { /\* child \*/

/\* 因此fork出来后的子进程应该主动把自己阻塞

\* 等待父进程的解除阻塞

\*/

p\_proc\_ready->p\_flags = 1; block(p\_proc\_ready);

/\* 这一部分是Part B 任务二部分，随后再解释 \*/

**int** position = find\_position(check\_table, multi\_argv[i][0]); u32 real\_checkNum = check\_table[position].checkNum;

u32 now\_checkNum = check(multi\_argv[i][0], check\_table[position].byteCount);

// u32 now\_checkNum = real\_checkNum;

**if** (real\_checkNum == now\_checkNum) {

/\* 子进程解除阻塞后就用execv执行命令

* 如此才会出现多进程同时运行的效果
* 而不是一个命令运行完再运行下一个命令

\*/

execv(multi\_argv[i][0], multi\_argv[i]);

} **else** {

printf("This file has been changed!\n");

}

}

}

}

close(1);

close(0);

1.}

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168 \_)

## 4 . 4 Part A 任务四

### 该部分并非独立完成的，我到现在也没有一个比较完整的动态分配内存的方案，也就是动态分配内存。刚开始读代码后，发现并没有用分页。但是我记得第六章的时候是用过分页的，然后去比对代码注意到，oranges 在第十章把线性地址转物理地址的宏定义给注释掉了。大致有了如何开启分页的思路， 也就是在 3.4.3 描述的思路。但是自己尝试许久后，bug 连连，无法找到跑通。

后来偶然和学长吃了顿饭聊起 os，发现他们也只完成了开启分页，完成静态内存分配，然而他们的当时虽然能跑通，但是还是有一些 bug 存在，会出现一些错误。本章节是在 18 级谭雨奇、王萌和石

可人的帮助下完成。

**在进程表中添加页表** 在 3.4.3 中我们提到，我们需要让页表成为进程表中的一部分并且随着上下文的切换进行页表的切换。直接将整个页表放在进程表中显然不现实，它太大了，因此我们可以选择将页目录的基址，即 cr3 寄存器的值，保存到进程表中。首先我们需要修改 proc\_table 这个结构体的

stackframe，添加我们的页表项。

,

＼

1. **struct** stackframe { /\* proc\_ptr points here Low \*/
2. u32 pde;

3 u32 gs; /\* \*/

4 u32 fs; /\* \*/

5 u32 es; /\* \*/

6 u32 ds; /\* \*/

7 u32 edi; /\* \*/

8 u32 esi; /\* pushed by save() \*/

9 u32 ebp; /\* \*/

10 u32 kernel\_esp; /\* <- ’popad’ will ignore it \*/

11 u32 ebx; /\* 栈从高地址往低地址增长\*/

12 u32 edx; /\* \*/

13 u32 ecx; /\* \*/

14 u32 eax; /\* \*/

15 u32 retaddr; /\* return address for assembly code save() \*/

16 u32 eip; /\* \*/

17 u32 cs; /\* \*/

18 u32 eflags; /\* these are pushed by CPU during interrupt \*/

19 u32 esp; /\* \*/

20 u32 ss; /\* High \*/

21 1.}; \_)

### 我们选择将页目录基址 pde 放在进程表的第一项，即 stackframe 中，和保存的其他寄存器放在一起，这样做的原因是我们希望尽可能不改变原来的上下文切换方式，我们只需要把我们添加的 cr3 寄存器放在进程的栈帧的顶部，这样当进行上下文切换时，esp 就会从 gs 变成 pde，这样我们就只需要在保存上下文的位置加上保存和恢复 cr3 的逻辑就可以了。在 kernel.asm 中的修改有两处，分别对应

上下文的保存和恢复：

,

＼

1. save:
2. pushad ; ‘.
3. push ds ; |
4. push es ; | 保存原寄存器值
5. push fs ; |
6. push gs ; /

7

1. mov eax, cr3
2. push eax

10

mov eax, PAGE\_DIR\_BASE

mov cr3, eax

mov eax, [esp + EAXREG - P\_STACKBASE]

;...

restart\_reenter:

dec dword [k\_reenter]

pop eax

mov cr3, eax

1.

pop gs pop fs pop es pop ds popad

add esp, 4

iretd

寄存器的恢复比较简单，我们需要注意的是保存上下文的时候，我们用了很多条指令，不仅包括

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28 \_)

从

### 栈中弹出寄存器的值并赋给 cr3，并且又从栈里面找到 eax 的值并赋给 eax，这样做的原因是因为 save 的时候 eax 里的值是有意义的，它保存着系统调用的参数，如果我们用 cr3 的值替换了它，会使得下一步的参数丢失了，因此我们需要相应的做修改。

另外一个需要注意的点是，我们用到了 EAXREG 这个常量，它表示的是 eax 在栈帧中的偏移量，

我们修改了栈帧的定义，因此需要相应的修改 sconst.inc 中对栈帧的相关定义如下：

,

＼

1. P\_STACKBASE equ 0
2. PDEREG equ P\_STACKBASE
3. GSREG equ P\_STACKBASE + 4
4. FSREG equ GSREG + 4
5. ESREG equ FSREG + 4

6 1.;... \_)

### 添加了定义，还需要给它赋初值，因为每次上下文切换时都会从栈帧中取出他的值并赋给 cr3 寄存器，因此我们需要给每个进程表项都附上一个初值。显然，只要给他们都赋值为 0x100000 就可以了， 继续使用操作系统初始化好的页表肯定是没错的，当我们给进程创建好了自己的页表再替换这个值就 好。在 kernel/main.c 中添加上对 pde 成员的赋值。

**在 exec 时填充页表** 到这一步为止，我就不知道如何继续进行下去了。后来是参考他们代码，发现也是静态进行分配，也就是刚开始就已经分配好，做了一个伪虚拟内存。

为了简化过程，我们并不准备维护一个新的数据结构来维护进程的地址映射，换言之，我们希望尽可能静态的分配进程的内存空间，那么，最简单的实现办法就是在 exec 的时候分配进程的内存，并初

始化好进程的页表，原因很简单，我们并不允许进程动态的申请内存，那么进程只会在 exec 的时候加载内存映像，我们可以在加载 elf 文件的时候给每个 segment 创建好对应的页表项，在这一过程中我们就完成了地址的重新映射，最后替换进程表中 cr3 寄存器的值，我们的任务就完成了。在 do\_exec.c 中修改部分如下，这段代码完成 elf 中各个段的加载，在这里我们引入了新的分页机制来完成。注释掉的

是原始代码，可以与新的代码进行比较。

,

＼

1. /\* overwrite the current proc image with the new one \*/
2. Elf32\_Ehdr\* elf\_hdr = (Elf32\_Ehdr\*)(mmbuf);
3. **int** i;
4. **for** (i = 0; i < elf\_hdr->e\_phnum; i++) {
5. Elf32\_Phdr\* prog\_hdr = (Elf32\_Phdr\*)(mmbuf + elf\_hdr->e\_phoff +
6. (i \* elf\_hdr->e\_phentsize));
7. **if** (prog\_hdr->p\_type == PT\_LOAD) {
8. // assert(prog\_hdr->p\_vaddr + prog\_hdr->p\_memsz <
9. // PROC\_IMAGE\_SIZE\_DEFAULT);
10. // phys\_copy((void\*)va2la(src, (void\*)prog\_hdr->p\_vaddr),
11. // (void\*)va2la(TASK\_MM,
12. // mmbuf + prog\_hdr->p\_offset),
13. // prog\_hdr->p\_filesz);
14. **int** size = ((prog\_hdr->p\_vaddr + prog\_hdr->p\_memsz - 1) » PG\_PAGE\_OFFSET) - (prog\_hdr->p\_vaddr »PG\_PAGE\_OFFSET) + 1;
15. Elf32\_Addr alloc\_addr = addMapping(src, p, (Elf32\_Addr)va2la(src,

(**void**\*)prog\_hdr->p\_vaddr), size);

1. phys\_copy((**void**\*)va2la(src, (**void**\*)(alloc\_addr | (prog\_hdr->p\_vaddr & PG\_PAGE\_OFFSET))),
2. (**void**\*)va2la(TASK\_MM, mmbuf + prog\_hdr->p\_offset),
3. prog\_hdr->p\_filesz);

19 }

20 1. } \_)

### 可以发现，不同于原来直接拷贝到对应地址，我们在拷贝之前通过 addMapping 这一函数分配了一个地址，然后才进行拷贝。addMapping 的定义如下。这一分配的作用，等于是我们根据待分配的segment 的大小，从地址范围里分配一些页面映射给这个 segment，并通过页表使得可以通过 segment 的虚拟地址寻址到分配的页面，页面可以从 1MB 的地址范围里任意选择，当然为了简单，我们选择从低地址到高地址连续分配，毕竟这样分配的唯一作用就是减少内存空洞。比如我们在 elf 里面的地址空间是不连续的，而是位于不同地址的很多小段，通过这次映射，它们在物理页面上的排布就是连续的，

因此我们就可以使用更大的地址空间。

,

＼

1 /\*

2 \* size为分配的页数

3 \* addr为起始地址---物理地址

4 \* 假设为页对齐,且不会重复分配同一段内存

5 \*/

1. Elf32\_Addr addMapping(**int** src, ptr\_pt p, Elf32\_Addr addr, **int** size) {
2. assert(size <= 1024);
3. assert(((addr & PG\_DIR\_MASK) »PG\_DIR\_OFFSET) == 3);
4. **if** (p->pde\_alloc[(addr & PG\_DIR\_MASK) »PG\_DIR\_OFFSET] == 0) {

10 // 该pde未分配页表项,分配一个新的页表项

1. p->alloc\_size++;
2. assert(p->alloc\_size <= 15);
3. p->pde\_alloc[(addr & PG\_DIR\_MASK) »PG\_DIR\_OFFSET] = p->alloc\_size;
4. p->pde[(addr & PG\_DIR\_MASK) »PG\_DIR\_OFFSET] = (u32)\_va2la(src, (**void**\*)(p->alloc\_size \* PT\_SIZE\_DEFAULT)) | PG\_P | PG\_USU | PG\_RWW;

15 }

1. **int** idx = p->pde\_alloc[(addr & PG\_DIR\_MASK) »PG\_DIR\_OFFSET] - 1;
2. Elf32\_Addr ret = p->alloc\_addr;
3. **int** pte\_start = (addr & PG\_PAGE\_MASK) »PG\_PAGE\_OFFSET;
4. **for** (**int** i = 0; i < size; i++) {
5. p->pte[i + idx \* 1024+ pte\_start] = (u32)\_va2la(src, (**void**\*)p->alloc\_addr) | PG\_P | PG\_USU | PG\_RWW;
6. p->alloc\_addr += 4096;
7. assert(p->alloc\_addr < PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_STACK\_SIZE\_DEFAULT);

23

24

25 1.};

}

**return** ret;

#### \_)

**分配栈空间** 添加了分页机制后，除了 elf 中的段以外，还有栈段等着我们分配。因为当我们创建了新的页表之后，它初始状态下是全空的，因此我们要想访问一段地址，首先得添加映射。原来是我们提前确定了栈将使用的位置，接下来只要把栈拷贝到这里，让 esp 和 eax（参数，指向栈）指向这里就好了，

但引入了分页之后我们显然不能这么简单的这么做，因为我们首先要添加映射。为了简单，我们没有改变栈的地址，仍然使用原来的地址（但此时栈的物理地址已经不是原来的地址了），由于栈的物理地址改变了（和线性地址不同），因此我们在给寄存器赋值的时候，要赋值成虚拟地址而不是物理地址（上

面的三个地址都是物理地址而不是线性地址）。

,

＼

1. /\* setup the arg stack \*/
2. **int** orig\_stack\_len = mm\_msg.BUF\_LEN;
3. **char** stackcopy[PROC\_ORIGIN\_STACK];
4. phys\_copy((**void**\*)va2la(TASK\_MM, stackcopy),
5. (**void**\*)va2la(src, mm\_msg.BUF),
6. orig\_stack\_len);

7

1. Elf32\_Addr stack\_addr = addMapping(src, p, (Elf32\_Addr)va2la(src, (**void**\*)(PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_STACK\_SIZE\_DEFAULT)), 16);
2. u8\* orig\_stack = (u8\*)(stack\_addr + PROC\_STACK\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK);

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36 1.

**int** delta = PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK - (**int**)mm\_msg.BUF;

**int** argc = 0;

**if** (orig\_stack\_len) { /\* has args \*/

**char** \*\*q = (**char**\*\*)stackcopy;

**for** (; \*q != 0; q++,argc++)

\*q += delta;

}

phys\_copy((**void**\*)va2la(src, orig\_stack), (**void**\*)va2la(TASK\_MM, stackcopy), orig\_stack\_len);

proc\_table[src].regs.ecx = argc; /\* argc \*/

proc\_table[src].regs.eax = (u32)PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK;

/\* setup eip & esp \*/

proc\_table[src].regs.eip = elf\_hdr->e\_entry; /\* @see \_start.asm \*/ proc\_table[src].regs.esp = PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK;

proc\_table[src].regs.pde = (u32)va2la(src, 0); phys\_copy((**void**\*)va2la(src, 0),

(**void**\*)va2la(TASK\_MM, p), PROC\_SIZE\_PT);

strcpy(proc\_table[src].name, pathname); \_)

### **放置页表，修改进程表** 最后一个问题是如何让 cr3 指向我们新创建的页表。最简单的方法就是把他放在进程空间里的一个固定位置。下面这段代码做的事很简单，我们让进程表中的 cr3 指针直接指向进程的最低地址，并且把我们新创建的页表拷贝到这个位置，并且我们在刚才创建 mapping 的时候，同

样按照这个位置进行地址的计算（因为页目录项中需要有页表的地址，我们提前假设它在这个地址，这

样当它被拷贝过来之后还是正确的）。

,

＼

1. proc\_table[src].regs.pde = (u32)va2la(src, 0);
2. phys\_copy((**void**\*)va2la(src,0),
3. (**void**\*)va2la(TASK\_MM, p),
4. 1. PROC\_SIZE\_PT); \_)

**最后的一些问题** 按照如上思路写完后，仍然无法正确运行。

### 首先考虑内核页表，刚才的实现看起来很正确，但它运行不了，因为进程的页表缺少了一些东西， 考虑下面这段代码。这段代码显然工作在内核中，而第四行我们将页表切换回了进程的页表，接下来，

当下一条 pop 指令运行时，问题出现了：现在的 esp 指向的是内核地址（proc\_table），而这段地址在进程页表中并没有映射，因此 cpu 认为地址无效，因此系统卡住了。解决办法当然是把这些缺少的映射添上，为了简单，我们不关心内核里除了 proc\_table 还有没有别的需要映射的地址，我们直接将内

核的所有地址都映射一遍，这样就不会出现问题了。

,

＼

1. restart\_reenter:
2. dec dword [k\_reenter]

3

1. pop eax
2. mov cr3, eax

6

1. pop gs
2. pop fs
3. pop es
4. pop ds
5. popad
6. add esp, 4
7. 1. iretd \_)

### 这段代码非常简单粗暴，它直接分配了三个页表，把最低的 12mb 内存映射了一遍——这包含了内核的全部地址空间，甚至还多出来了一部分，这不重要，我们只要保证它映射了所有内核地址就可以

了，这样即使我们切换到了进程的页表，它至少还可以在内核里继续工作而不是直接报错。

,

＼

1 **void** InitKernelMapping(**int** src, ptr\_pt p) {

2 **for**(**int** i = 0; i < 3; i++) {

1. p->pde[i] = (u32)\_va2la(src, (**void** \*)((i + 1) \* PT\_SIZE\_DEFAULT)) | PG\_P

| PG\_USU | PG\_RWW;

1. p->pde\_alloc[i] = i + 1;
2. u32 addr = (u32)i «PG\_DIR\_OFFSET;

6 **for**(**int** j = 0; j < 1024; j++) {

7 p->pte[j + i \* 1024] = addr | PG\_P | PG\_USU | PG\_RWW;

8 addr += 4096;

9 }

10 }

11 p->alloc\_size = 3;

12 1.} \_)

### 添加分页机制很简单，但它会影响到操作系统原本的一些机制——例如 IPC。我们添加了分页机制之后，进程运行的很顺利，上下文切换也好像没有什么问题。但即使我们的分页机制“好像没有什么 问题”之后，它仍然没法完全正常工作，在系统调用的时候系统会崩溃。原因是因为，我们通过在进程 页表中添加内核地址映射，解决了进程上下文中访问内核地址的问题，但当每个进程的地址映射彼此独立的时候，我们没法解决的是内核上下文中如何访问进程地址，而这恰恰是基于消息队列的系统调用正在做的事，在进程空间中创建消息，然后把地址传给内核空间。显然，因为内核使用自己的平坦页表，这个地址对于内核来说是错误的，因此原本的 IPC 机制就完全没法正常工作了。在我们多次尝试

下，还是无法解决学长学姐他们遗留下来的这一问题，但至少分页看似运行好像还比较正常。

## 4 . 5 Part B 任务一

,

我们编写了一个非常简单的 c 程序，就是两条 printf，打印 hello 和我们小组成员名字。

**#include** "stdio.h"

**int** main(**int** argc, **char**\* argv[]) { printf("hello\n");

printf("pya pzx whx thm\n");

1.

}

objdump 指令对 hello 文件进行反汇编后，可以查看对应的汇编指令：

1 ＼

2

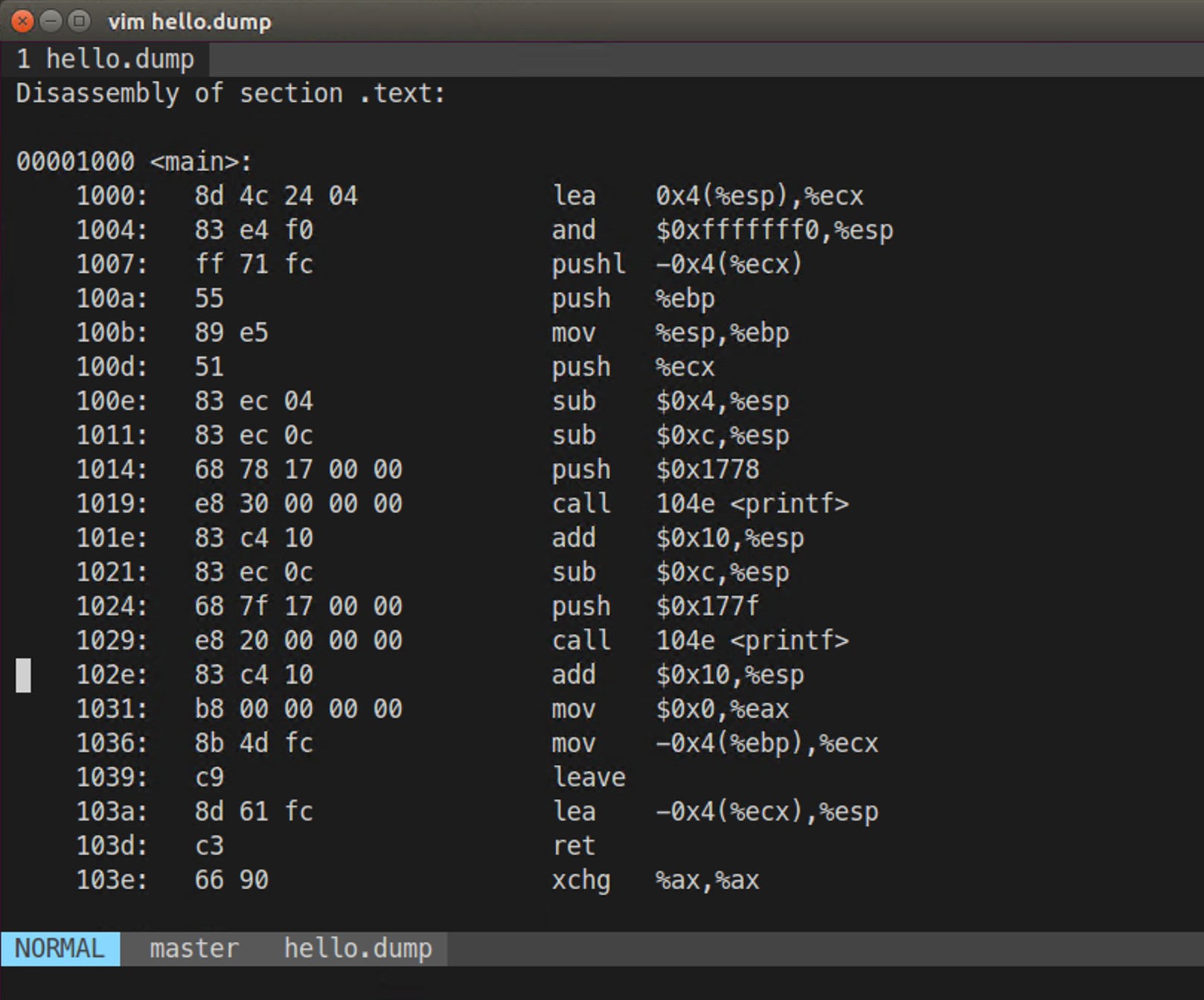
3

4

5

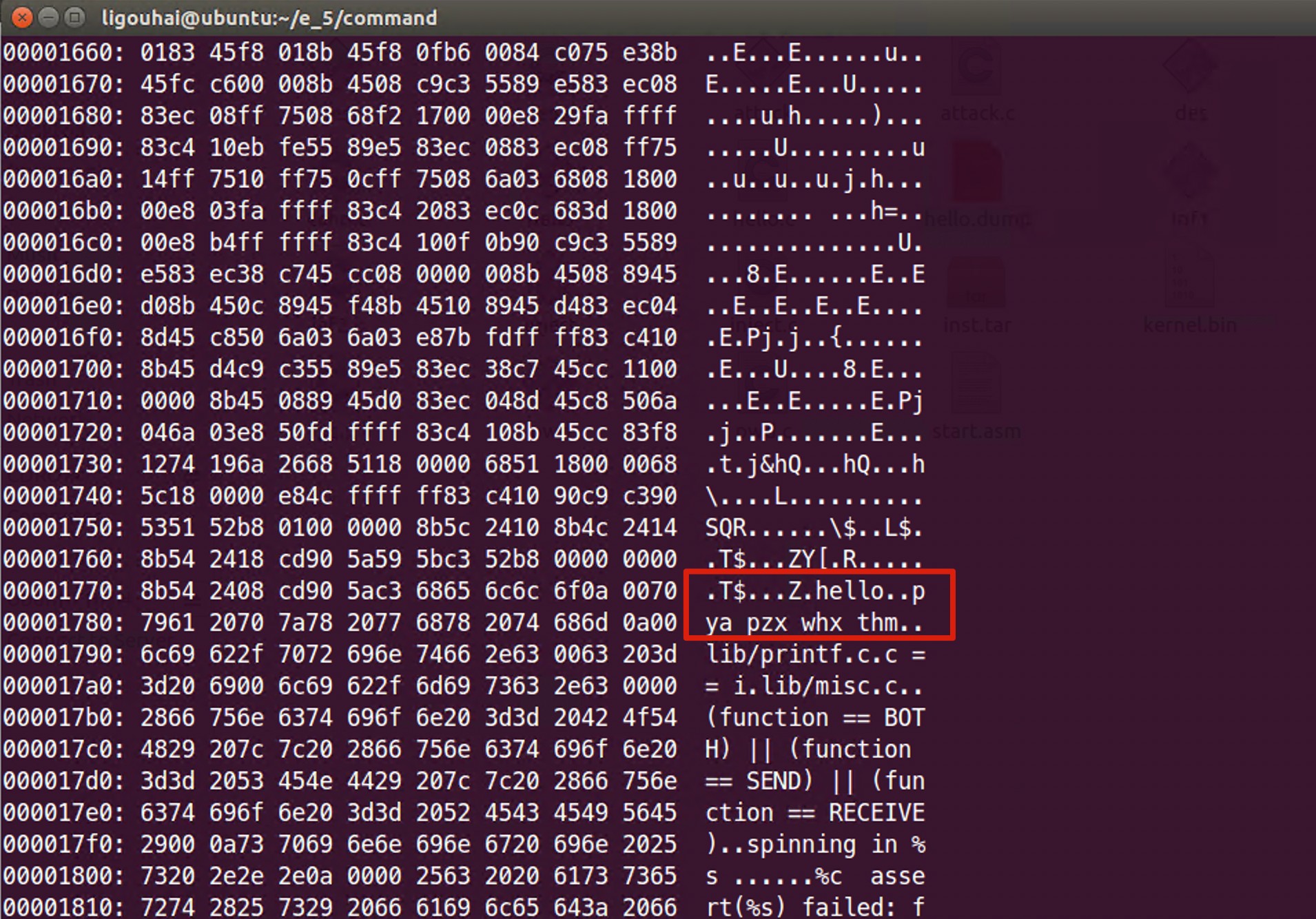
6 \_)

用

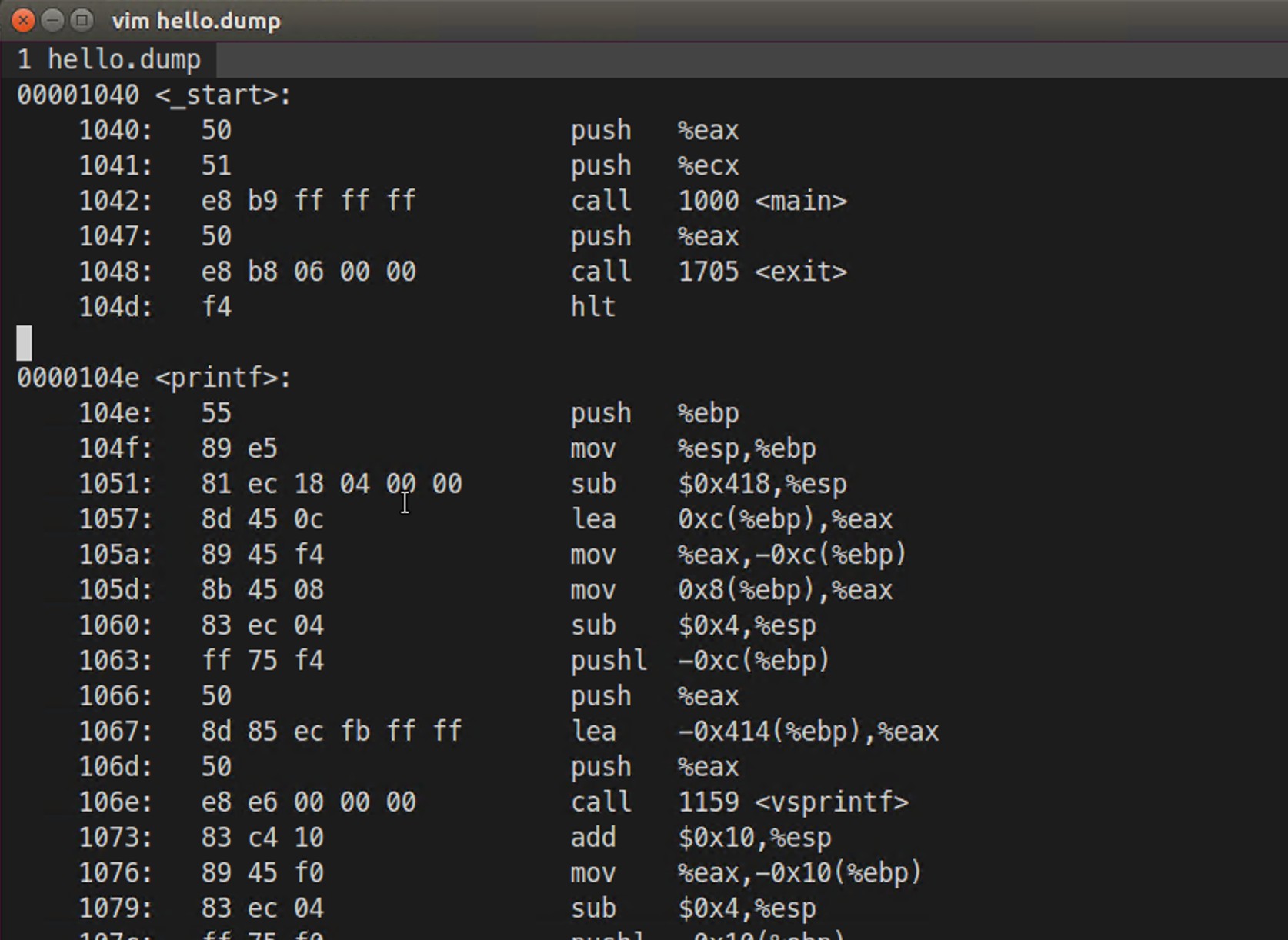


### 尝试查找文件中是否存在 code cave，也就是找到 nop 最多的位置，发现没有。于是我们考虑用新增节的方式实现，一开始，我们想通过 int 80 中断的调用添加字符串的打印，但是在 bochs 运行中无法调用很多的系统中断，所以考虑从原程序中含有的内容入手。我们干脆覆盖掉一些其他的代码进行注入。

可以看到在 0x1778 处有字符串 hello



在 0x104e 处有 printf



1 ＼

,

我们可以复用这个输出，编写 shellcode 如下：

**char** inject\_code[] = {

0 68, 078, 0x17, 000, 000,

2

3

0xe8, 0x20, 000, 000, 000

1.};

\_)

4

### 将文件指针指向对应的位置后，即可注入 shellcode。inject.c 代码如下：

,

＼

1. **#include** "stdio.h"
2. **#include** "string.h"
3. **#include** "elf.h"

4

5 **#define** PAGESIZE 4096

6

1. **void** cal\_addr(**int** entry, **int** addr[]) {
2. **int** temp = entry;
3. **int** i;

10 **for** (i = 0; i < 4; i++) {

11 addr[i] = temp % 256;

12 temp /= 256;

13 }

14 }

15

1. **void** inject(**char**\* elf\_file) {
2. printf("start to inject...\n");

18

1. **int** old\_entry;
2. Elf32\_Ehdr elf\_ehdr;
3. Elf32\_Phdr elf\_phdr;

22

1. **int** old\_file = open(elf\_file, O\_RDWR);
2. read(old\_file, &elf\_ehdr, **sizeof**(Elf32\_Ehdr));
3. old\_entry = elf\_ehdr.e\_entry;
4. printf("old\_entry: %x\n", old\_entry);

27

1. // Modify e\_shoff of ELF Header，adding PAGESIZE
2. // elf\_ehdr.e\_shoff += PAGESIZE;

30

31 **int** i = 0;

32

33 printf("Modifying the program header table...\n");

34 // 读取并修改程序头部表

1. close(old\_file);
2. old\_file = open(elf\_file, O\_RDWR);
3. **char** buffer[20000];
4. read(old\_file, buffer, elf\_ehdr.e\_phoff);

39

40

41

42

43

44

45 }

46

read(old\_file, &elf\_phdr, **sizeof**(elf\_phdr));

printf("Inserting the injector...\n");

// 插入注入程序

close(old\_file);

insert(elf\_ehdr, elf\_file, old\_entry);

47 **void** insert(Elf32\_Ehdr elf\_ehdr, **char**\* elf\_file, **int** old\_entry) {

48 // 程序的原始入口地址

1. **int** old\_entry\_addr[4];
2. cal\_addr(old\_entry, old\_entry\_addr);

51

52 // printf("old\_entry = 0x%x%x%x%x\n", old\_entry\_addr[3],old\_entry\_addr[2],old\_entry\_addr[1],old\_entry\_addr[0]);

53 // 每一行对应一条汇编代码

54 **char** inject\_code[] = {

55 0x68, 0x78, 0x17, 0x00, 0x00,

56 0xe8, 0x20, 0x00, 0x00, 0x00,

57 };

58 **int** inject\_size = **sizeof**(inject\_code);

59

60 // 防止注入代码太大

1. **if** (inject\_size > PAGESIZE) {
2. printf("Injecting code is too big!\n");
3. exit(0);

64 }

65 **int** old\_file = open(elf\_file, O\_RDWR);

66 u8 buffer[20000];

1. read(old\_file, buffer, 0x1024);
2. write(old\_file, inject\_code, inject\_size);
3. close(old\_file);
4. old\_file = open(elf\_file, O\_RDWR);
5. read(old\_file, buffer, 0x1024);
6. read(old\_file, buffer, 10);

73 **for** (**int** i =0;i <10;i++) {

74 printf("%x\n",buffer[i]);

75 }

76

77 printf("Finished!\n");

78 }

79

80 **int** main(**int** argc, **char**\*\* argv) {

81

**if** (argc != 2) {

printf("inject <elf\_filepath>\n"); exit(0);

}

inject(argv[1]);

**return** 0;

1.}

82

83

84

85

86

87 \_)

,

为实现对缓冲区的溢出攻击，编写函数如下。

**#include** <stdio.h>

**int** i;

**int** \*addr;

**void** test() { printf("pya, whx, thm, pzx"); }

**void** testat() {

**char** buff[72] = {0};

**for** (i = 0; i < 72; i++) { buff[i] = 0;

}

**for** (; i < 72; i++) { buff[i] = 0;

}

addr = &buff[72];

**for** (i = 0; i < 3; i++) { addr[i] = 0x1000;

}

}

**void** main(**int** argc, **char** \*argv[]) { testat();

1.

}

调用 testat 函数后，栈底有返回地址，正常来说应该返回 main 函数。但是我们通过 objdump

1 ＼

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

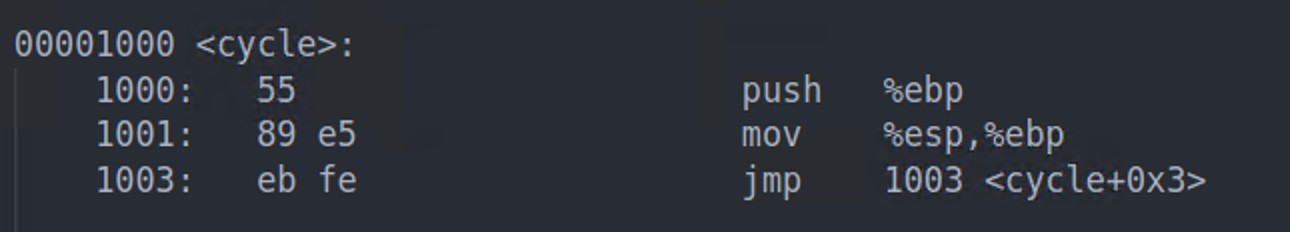
25 \_)

得

### 到 test() 函数的返回地址为 0x1000，于是我们在向高地址填充地址 0x1000，直到覆盖返回地址。那么会跳转到 test() 函数中输出我们想要的信息。

这种方式的栈溢出模拟了代码在代码段中的情况，但更多时候是通过产生 shellcode 将代码放入堆栈段中并跳转到堆栈运行，下面进行第二种方式的攻击演示：

* 首先将代码通过 objdump 进行反汇编，由于 printf 在堆栈段中的地址需要重定位，因此这里使用简单的 while 循环来展示。

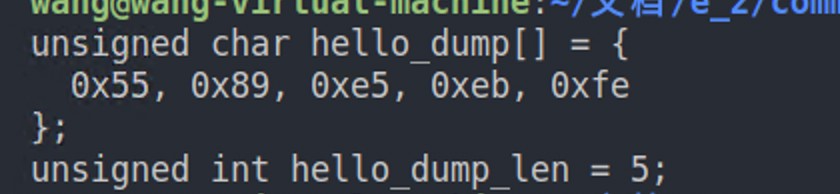


* 然后,进入 gdb 将这段循环的代码转换为二进制文件 ＼

1 1.dump memory xxx.dump cycle cycle+5 \_)

### 然后,使用 xxd 过滤出 shellcode, 得到 shellcode 如下 ＼

1 1.xxd -i xxx.dump \_)



1 ＼

2

,

将这段代码放入 shellcode\_dump[] 数组中存放起来，然后使用下列代码：

**#include** <stdio.h>

**int** i;

**int**\* addr;

**unsigned char** shellcode\_dump[] = { 0x55, 0x89, 0xe5, 0xeb, 0xfe

};

**void** testat() {

**char** buff[72] = {0};

**for** (i = 0; i < 72; i++) {

**if** (0 == shellcode\_dump[i])

**break**;

buff[i] = shellcode\_dump[i];

}

**for** (; i < 72; i++) { buff[i] = 0;

}

addr = &buff[72];

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

**for** (i = 0; i < 6; i++) { addr[i] = buff;

}

}

**void** main(**int** argc, **char**\* argv[]) { testat();

1.

}

这段代码会将原本函数返回地址覆盖为 buff 的初始地址，而 buff 已经被填入了 shellcode，

24

25

26

27

28

29

\_)

30

### 因此

会不断执行在栈区循环。

## 4 . 6 Part B 任务二

在对可执行文件进行完整性验证这一部分，我们采用密码学算法 AES 进行可信防护。正如之前提到的，操作系统会先将指定的编译链接后的可执行文件打包成一个 tar 包 inst.tar，再用工具 dd 写入磁盘的某段特定扇区。

在启动系统时，mkfs 会在文件系统中建立一个新文件 cmd.tar，并将其 inode 中的 i\_start\_sect 字段设置为 inst.tar 所在的扇区号。在初始化的时候会将 cmd.tar 解包，把其中包含的文件都存入文件系统。从理论上分析，我们应当在可执行文件打包成 inst.tar 之前，对可执行文件计算得到校验值，存放在操作系统的全局变量中。但因为在实验中，文件压缩操作是在 Makeﬁle 中完成的，而且整个过程没有与外界的交互，即文件在被解压之前都是处于原始安全的状态，为了方便起见，我们选择在解包的时候对文件进行第一次的校验值计算，获得标志信息。

当某个子进程想执行某个文件时，通过文件名在文件系统中找到该文件，获得文件的校验值，并与存放的校验值进行对比，如果不同说明文件已经被篡改，不能执行；否则表示文件没有被篡改并继续打开执行。

我们首先把 AES 算法放入到 lib 文件夹，作为一个库函数。因为校验值在系统中需要有存放的位置，我们在头文件 proc.h 中定义了一个结构体 check\_t，其中包含了三个成员：ﬁlename、byteCount

和 checkNum，分别代表了文件名、文件大小（文件字节数）和校验值。

,

＼

1. **struct** check\_t {
2. **char** filename[32];
3. **int** byteCount;
4. u32 checkNum;

5 1.}; \_)

### ,同时，在头文件 global.h 中定义了 check 结构体数组 check\_table，用来存放所有文件的相关信＼息。

1. 1.PUBLIC **struct** check\_t check\_table[NR\_CHECKFILES]; \_)

### 在 kenel/main.c 中，Init 函数调用了 untar 函数，我们将校验值计算的过程放入 untar 中。代码

含义见注释。

,

＼

1 // 记录了check\_table的下标

1. **int** check\_count = 0;
2. **while** (1) {
3. read(fd, buf, SECTOR\_SIZE);

5 **if** (buf[0] == 0)

6 **break**;

7

8 **struct** posix\_tar\_header\* phdr = (**struct** posix\_tar\_header\*)buf;

9

1. /\* calculate the file size \*/
2. **char**\* p = phdr->size;
3. **int** f\_len = 0;
4. **while** (\*p)

14 f\_len = (f\_len \* 8) + (\*p++ - ’0’); /\* octal \*/

15

16 // FINISHED: untar first, than compute checksum!

17 // bytes\_left记录了还剽多少字节没有读取

18 **int** bytes\_left = f\_len;

19 // fout是与入到oranges中的文件句柄

1. **int** fdout = open(phdr->name, O\_CREAT | O\_RDWR);
2. **if** (fdout == -1) {
3. printf(" failed to extract file: %s\n", phdr->name);
4. printf(" aborted]");
5. **return**;

25 }

26

1. printf(" %s (%d bytes)", phdr->name, f\_len);
2. **char** temp\_name[32];
3. strcpy(temp\_name, phdr->name);
4. **while** (bytes\_left) {

31 // 一块一块读取文件内容到buf

1. **int** iobytes = min(chunk, bytes\_left);
2. read(fd, buf,
3. ((iobytes - 1) / SECTOR\_SIZE + 1) \* SECTOR\_SIZE);
4. write(fdout, buf, iobytes);
5. bytes\_left -= iobytes;

37 }

38 close(fdout);

39

40 **if** (strcmp(temp\_name, "kernel.bin") != 0) {

41 // 把文件名字复制到check\_table中

42 strcpy((check\_table + check\_count)->filename, temp\_name);

43 // 把文件大小也放入check\_table中

44 check\_table[check\_count].byteCount = f\_len;

45 // 通过check函数计算校验值放入checkNum中

46

check\_table[check\_count].checkNum = check((check\_table + check\_count)->filename, f\_len);

printf(" (checkNum = %d)\n", check\_table[check\_count].checkNum); check\_count = check\_count + 1;

} **else** {

printf("\n");

}

1.

}

47

48

49

50

51

\_)

52

### 接下来我们看看 checkh 函数的具体实现：

,

＼

1. PUBLIC u32 check(**char**\* filename, **int** byteCount) {
2. **int** hFile = open(filename, O\_RDWR);
3. **if** (hFile < 0) {
4. **return** -1;

5 }

6

1. u8 plaintext[16] = {0};
2. u8 ciphertext[16] = {0};

9 u32 key[4] = {0x12345678, 0x20193021, 0x40023201, 0x93021400};

10 u32 current\_checkSum = 0;

11

12 // FINISHED: 一块一块读，加快check速度

1. u8 buffer[byteCount];
2. read(hFile, buffer, byteCount);
3. **for** (**int** i = 0; i < byteCount; i++) {
4. plaintext[i % 16] = buffer[i];

17 **if** ((i % 16== 0&& i != 0) || (i == byteCount - 1)) {

18 // 每读取完16字节后进行一次加密

19 aes\_enc(ciphertext, plaintext, key);

20 **for** (**int** j = 0; j < 4; j++) {

21 // 我们用32bit作为一个checkSum，密文有128位

22 // 故密钥要循环四次计算checkSum

23 u32 temp = ((u32)ciphertext[4 \* j + 0] «24) |

24 ((u32)ciphertext[4 \* j + 1] «16) |

25 ((u32)ciphertext[4 \* j + 2] «8) |

26 ((u32)ciphertext[4 \* j + 3] «0);

27 current\_checkSum ^= temp;

28 }

29 }

30 }

1. close(hFile);
2. **return** current\_checkSum;

33

1.

}

完成上述步骤后，系统已经获得了各可执行文件的原始校验值，接下来就是在程序运行前对

\_)

### 其进

行再次的校验。获得了用户的键入后，会调用 execv 运行程序，但在运行之前需要验证校验值，从而判

断是否能执行此文件。

,

＼

1. **else if** (pid == 0) { /\* child \*/
2. p\_proc\_ready->p\_flags = 1;
3. block(p\_proc\_ready);

4

5 /\* 利用find\_position函数找到该程序在check\_table的位置 \*/

6 **int** position = find\_position(check\_table, multi\_argv[i][0]);

7 /\* 得到原始校验值 \*/

8 u32 real\_checkNum = check\_table[position].checkNum;

9 /\* 对现在运行的程序进行校验值计算 \*/

10 u32 now\_checkNum = check(multi\_argv[i][0], check\_table[position].byteCount);

11 /\* 把这一句话取消注释就是关闭完整性验证

12 \* 因为取消注释后now\_checkNum 始钦等于 real\_checkNum

13 \*/

14 // u32 now\_checkNum = real\_checkNum;

15

16 /\* 如果校验值相等则可以运行，否则打印文件被修改 \*/

1. **if** (real\_checkNum == now\_checkNum) {
2. execv(multi\_argv[i][0], multi\_argv[i]);
3. } **else** {
4. printf("This file has been changed!\n");

21 }

22 1. } \_)

,

上面提到的 ﬁnd\_position 也非常简单，就是遍历寻找位置：

PUBLIC **int** find\_position(**struct** check\_t check\_table[], **char**\* filename) {

**for** (**int** i = 0; i < NR\_CHECKFILES; i++) {

**int** flag = strcmp((check\_table + i)->filename, filename);

**if** (flag == 0) {

**return** i;

}

}

**return** -1;

1.

}

除此，在本次实验中，实现的校验是在子进程创建后、执行前进行的，相当于为执行函数 execv

1 ＼

2

3

4

5

6

7

8

9 \_)

加

### 了一个保护壳。但这只能局部有效，如果有其他地方调用 execv 函数，这时候是缺少验证的。因此，改

进的方法就是直接把校验的过程嵌入到 execv 函数体里，可以提高通用性。

## 4 . 7 Part B 任务三

在这个任务里，我们编写了一个软件中断，这个中断每过一定时间间隔触发，可以对进程表内的所有进程进行检查是否产生栈溢出，检查栈溢出的原理是根据内存的分布情况。

对于用户定义的程序，由于原本的 OrangeOs 中没有定义栈的区域，只给出了栈的上限，因此我们假定一个数据 0xF0000 作为栈的上限，栈位于 0xF0000 以上的位置，因此我们根据这条原则判断 proc 表中的用户进程的 eip 是否大于 0xF0000 来判断是否产生了栈溢出。

对于系统内核初始化的程序，由于其栈的区域为 [task\_stack, task\_stack + STACK\_SIZE\_TOTAL]， 因此可以直接判定 eip 是否在栈区。

* 下面是在 proc.c 中定义一个用来判断栈溢出的函数，这个函数对所有运行的进程进行检查
* 首先检查

,

＼

1 PUBLIC **int** sys\_checkstack()

2 {

3 **int** i;

4 /\* 遇历进程表 \*/

1. PROCESS \*P = proc\_table;
2. **for**(i = 0; i < NR\_TASKS + NR\_PROCS; i++, P++)

7 {

1. **if**(P->p\_flags == FREE\_SLOT) //进程未使用
2. **continue**;

10

11 /\* 对于在内核中初始化的程序检查其eip是否大于task\_stack的地址 \*/

1. **else if**(i < NR\_TASKS + 4){
2. **if**(P->regs.eip > (u32)task\_stack){
3. printf("detect stack overflow\n");
4. assert(0);

16 }

17 }

18

19 /\* 对于用尸的运行程序检查其eip是否大于0xF0000 \*/

20 **else if**(P->regs.eip > 0xF0000)

21 {

1. //printf("CS=%x and SS=%x\n", P->regs.cs, P->regs.ss);
2. printf("pid:%d, name:%s, detect stack overflow\n", i, P->name);
3. P->regs.eip = exit;

25 }

26 //printf("name:: %s CS:IP = %x : %x SS:esp = %x::%x\n", P->name, P->regs.cs, P->regs.eip ,P->regs.ss, P->regs.esp);

27 }

28 **return** 1;

29 \_)

1.}

### 先在 proto.h 加入此函数的声明，并在 global.c 中将此函数添加到系统调用表中：



* 在

,

syscall.asm 中加入此系统调用：

1.

......

\_NR\_checkstack equ 2

......

global checkstack

......

checkstack:

mov eax, \_NR\_checkstack

**int** INT\_VECTOR\_SYS\_CALL

ret

＼

1

2

3

4

5

6

7

8

9 \_)

### 最后在 kernel/main.c 的 TextA 进程中每隔一定时间间隔调用检查函数，在检查前需要延迟一段时间先让 INIT 函数完成初始化：

,

＼

1. **void** TestA() {
2. **int** k, i, j;
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

8 **int** check;

9 **for**(k=0;k<100;k++){

10 **for**(i=0;i<10;i++){

11 **for**(j=0;j<10000;j++){}

12 }

13 }

14 **while**(1){

15 **if**(ticks % 200== 0) // 等效于按照定时中断触发

16 check = sys\_checkstack();

17 }

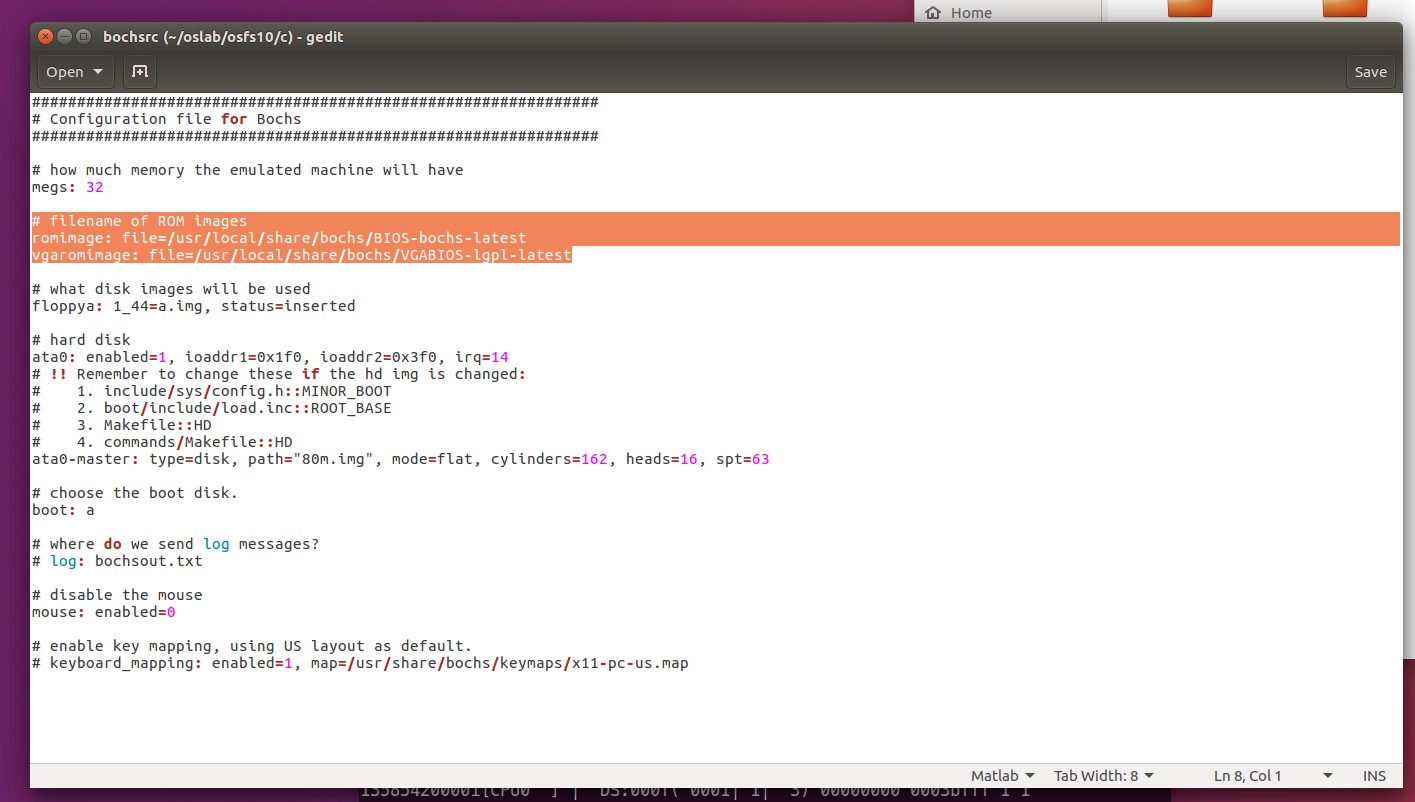
18 1.} \_)

### 在这种检测方式下，我们认为只要 eip 不大于 0xF0000 就算合法，这是基于栈区往往在高地址而设置的。这种防御手段仅能防御住部分将代码放在栈区执行的缓冲区溢出攻击。对于我们在任务一实现的栈溢出攻击而言，第一种情况，也就是通过栈溢出跳转到代码段中的函数入口执行，这种防御手段无法防御，这种攻击手段常见于绕过保护验证；但对于第二种攻击手段，也就是构造 shellcode 放入栈区等到溢出后执行的攻击可以进行防御。

# 五、 实验结果总结

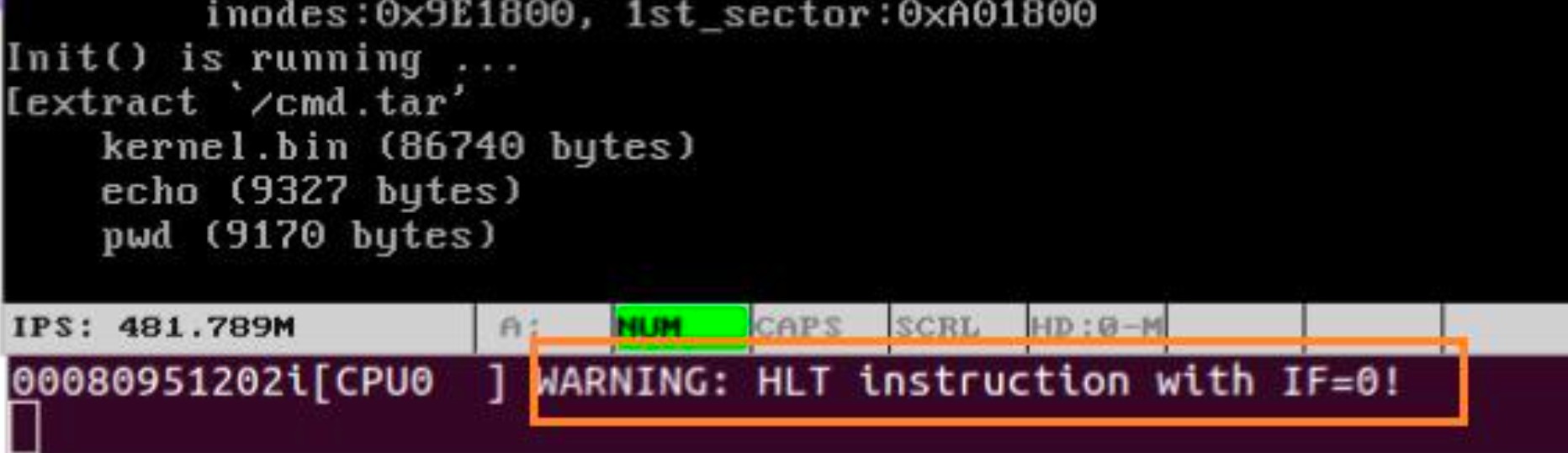
## 5 . 1 准备工作

首先每次都需要改一改 bochsrc，需要把 romimage 和 vgaromimage 的位置改成自己电脑安装的位置：



makeﬁle 编译的时候要加上-fno-stack-protector，这个问题在之前的实验也出现过。

上面的修改好之后，第一次运行的时候卡住了，报错为 HLT IF=0。在 kernel/proc.c 中，应该在进入 msg\_receive 函数后先关中断，在退出函数时（该函数有两个 return 退出）重新开中断。这是为了防止进程冲突，保护该过程不被中途停止。



最后需要在 kernel/tty.c 中，将切换 shell 的功能键 ALT 改为 CTRL。这是因为 ALT 在 ubuntu

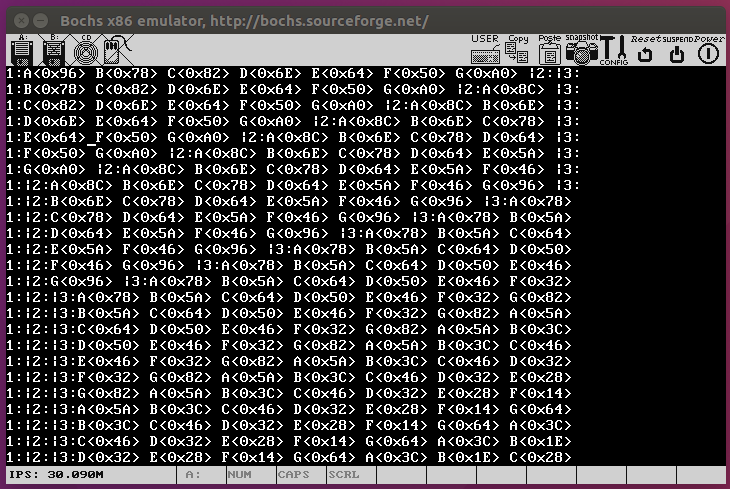
中是特殊的功能键，所以我们用 CTRL 替换了该键。

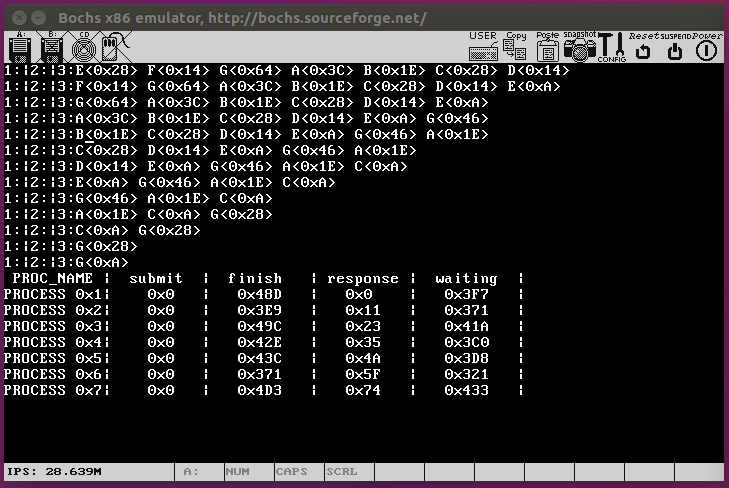
## 5 . 2 实验结果

## 5 . 2 . 1 Part A

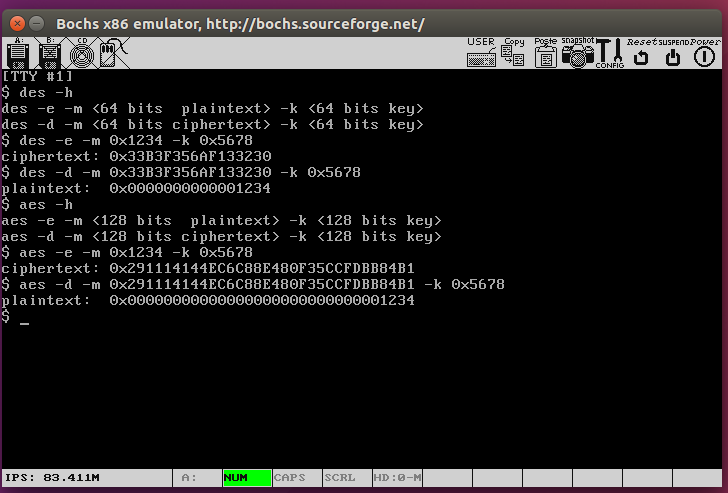
Part A 任务一实现了多级反馈调度算法，这里打印出了进程在三个队列中的排队情况，并且 <> 内包含了该进程剩余时间片。可以看到 A-G 进程首先都在 1 队列，然后 A 在第一队列运行了 10 个时间片后，剩余时间片从 0x96 变成 0x8C，并且进入到第二个队列...

由于在第六章的代码中，所有进程都必须手工创建，也就是只能在 0 时刻的时候手动添加，所以最后输出的性能评价信息中提交时刻都是 0。并且后面还输出了结束时间、响应时间和等待时间。通过理论计算验证，实际输出的时间和理论略有一点偏差。这是因为实际过程中，除了这些函数要消耗时钟， 其他函数比如调度过程、切换过程等等都需要消耗少量时钟。

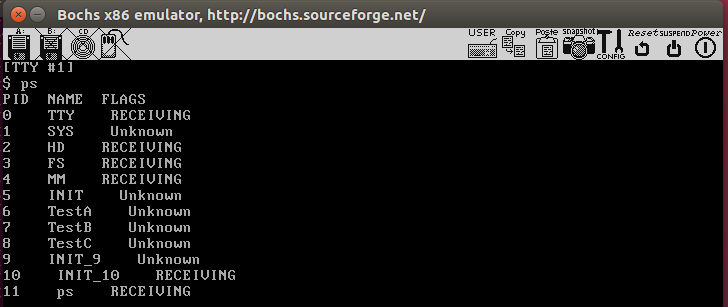




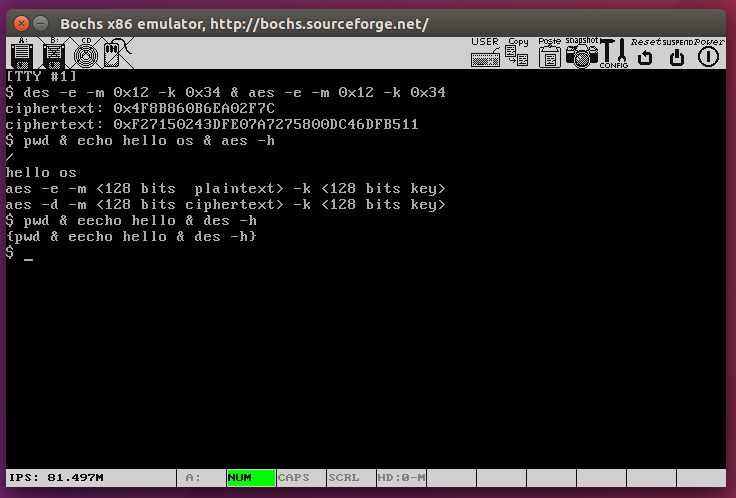
Part A 任务二实现 DES 和 AES 算法，输入 des -h 和 aes -h 可以获得帮助信息。加解密如下图所示，可以看见我们成功实现了加解密。

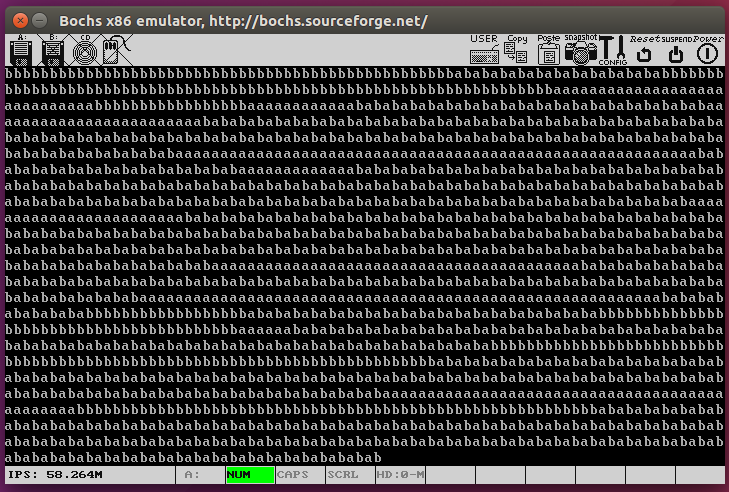


Part A 任务二我们还实现了类似 linux 的 ps 命令，可以看到我们成功利用消息传递机制输出了进程信息：



Part A 任务三实现了 shell 多任务执行的支持。从下图可以看出，不管是两个任务还是三个任务， 都可以成功运行。但是如果命令出错，无论在哪一个位置，都不会继续执行。同时为了展示我们所做的确实成功实现了多任务执行，也就是能够正常调度，我们编写了两个函数 inf1 和 inf2，inf1 无限循环输出 a，inf2 无限循环输出 b，可以看到输出结果 ab 交叉输出，说明我们确实完成了多任务的支持。

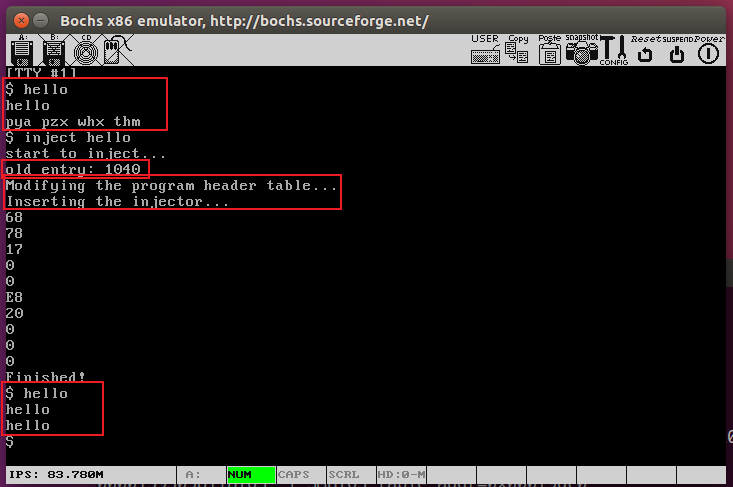




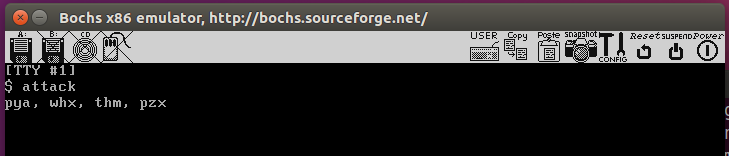
Part A 任务四我们认为思路是这样做没有问题，但还是出现了许多 bug 并没有调通。

## 5 . 2 . 2 Part B

Part B 任务一第一个小任务是进行 elf 文件注入，由于我们完成了任务二的完整性检验，所以在进行该实验的时候我们需要去/kernel/main.c 中 341 行取消注释，取消注释就是关闭完整性检验，注释这一行就是开启完整性检验。注入的结果如下，首先输出 hello 和我们小组成员名字；在注入时，保存old entry，修改 program header table，然后进行注入，下面的数字是为了编写时调试方便输出的；注入完成后运行 hello，可以发现这会会输出两个 hello 而不输出小组成员名字了，说明成功注入了代码。

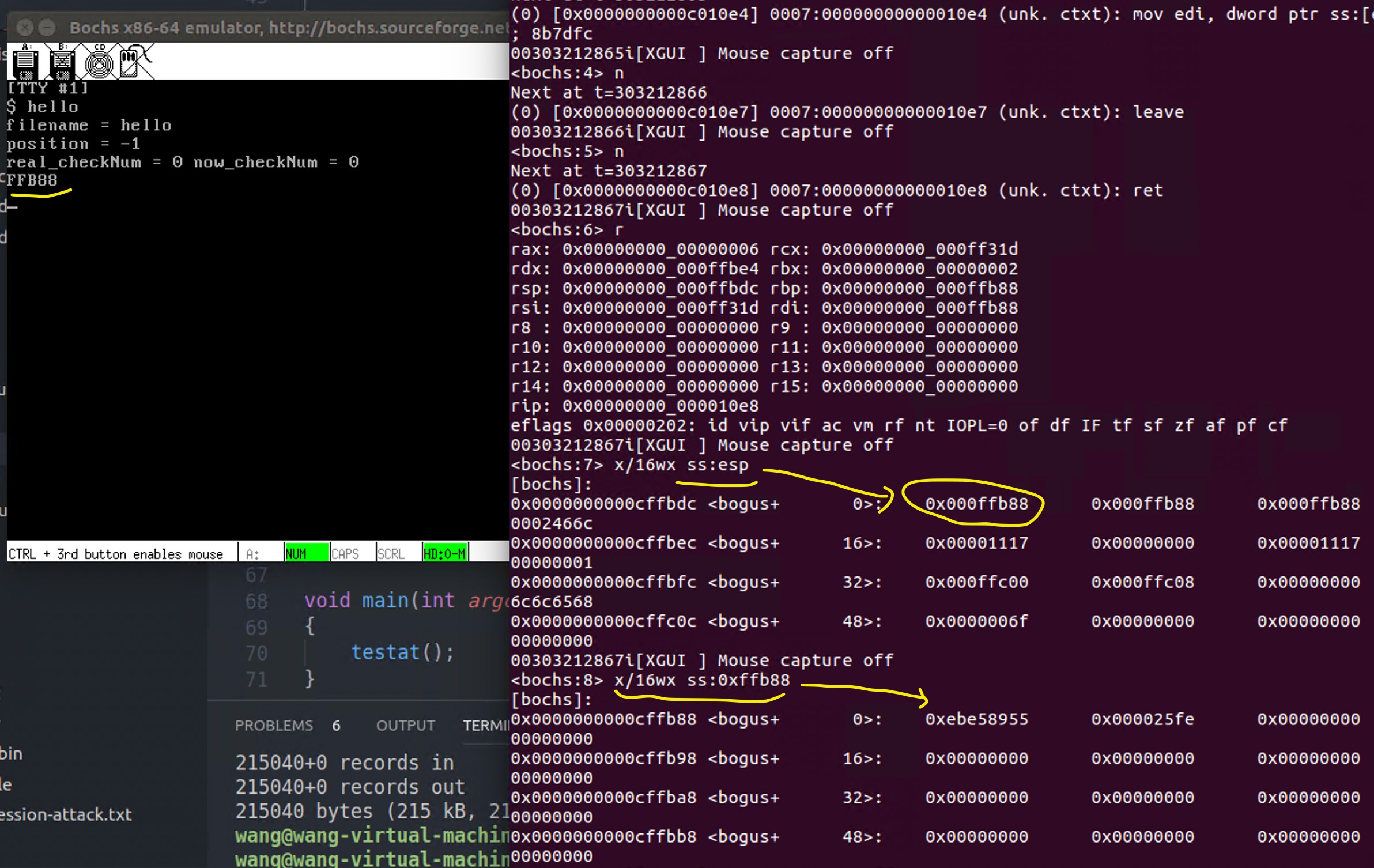


Part B 任务一的缓冲区溢出部分我们有两个版本的代码。第一个是在代码段的缓冲区溢出，结果如下，我们可以看到成功返回到 test() 函数然后执行了其中的 printf 代码：

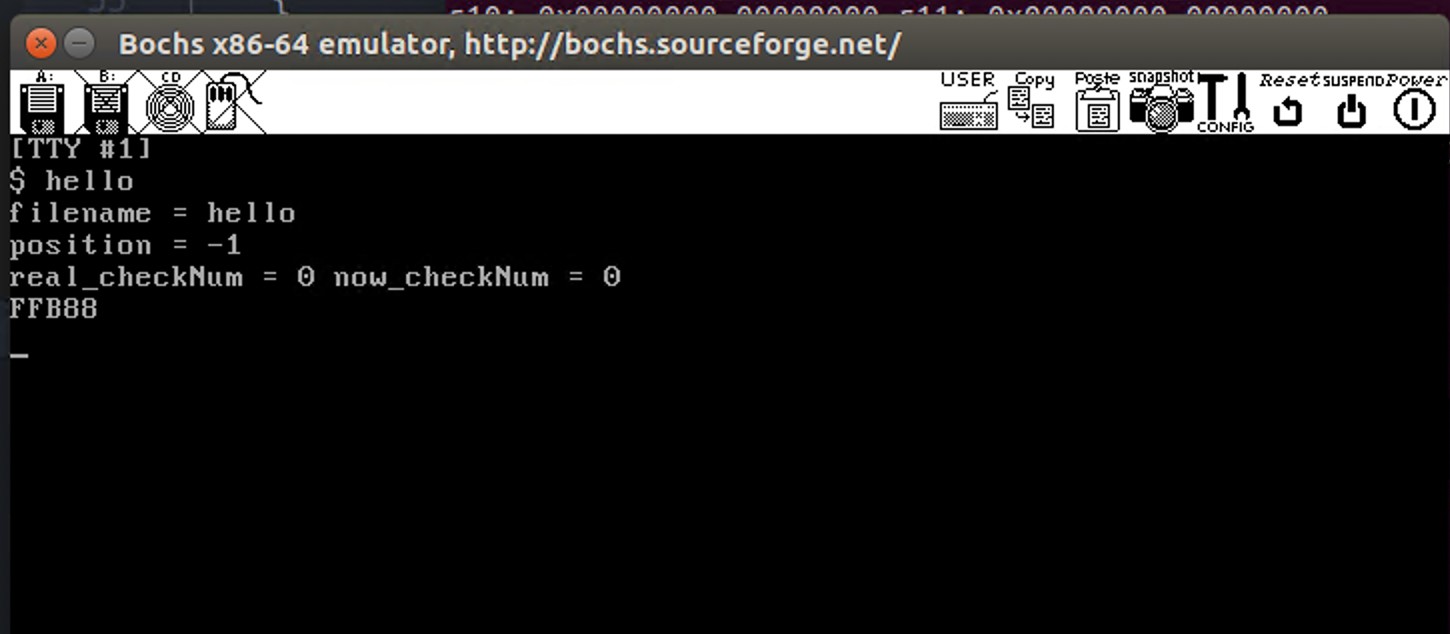


第二个时堆栈段的缓冲区溢出，这段代码会将原本函数返回地址覆盖为 buff 的初始地址，而 buff

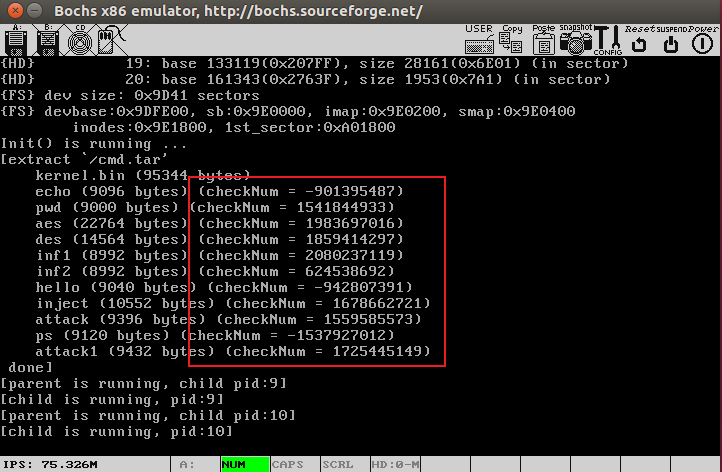
已经被填入了 shellcode，因此会不断执行在栈区循环。下面是实验结果：

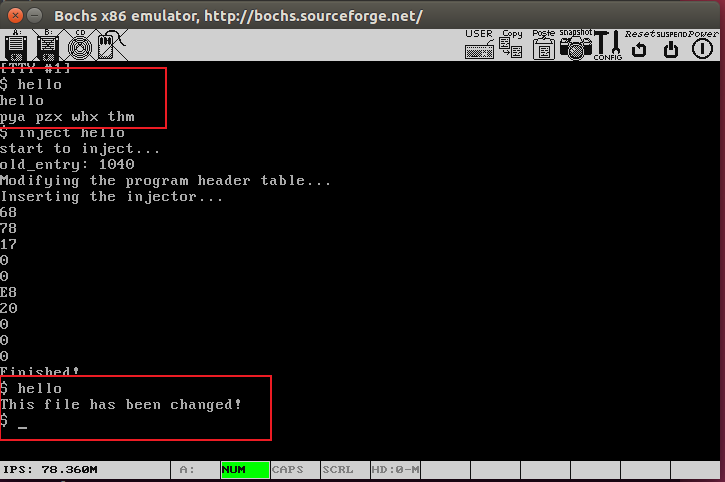


* 在返回点下断点查看堆栈的返回情况，可以看见在返回点处 ss:esp 与程序输出的数组 buff 的地址相等，这说明程序下一步将会返回到 buff 的地址处。查看 ss:esp 处的内容，发现已经是我们填入的 while 循环的机器码，因此程序下一步进入死循环。

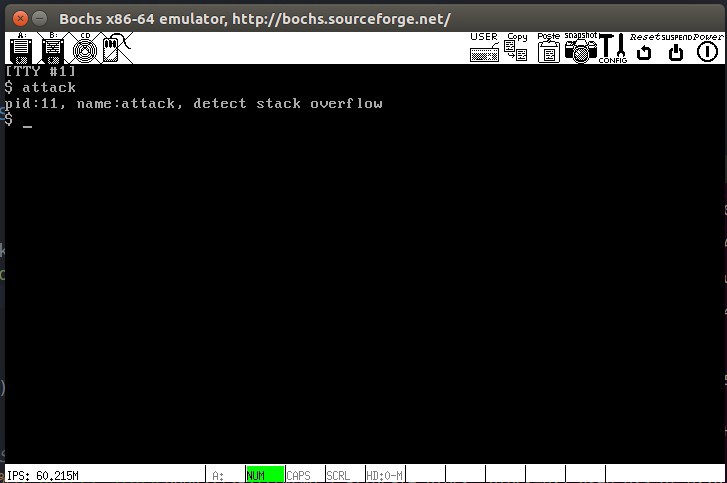


Part B 任务二要在运行命令前对可执行文件做一次完整性检验，我们注释掉/kernel/main.c 中 341 行即可开启完整性检验。首先可以看到 os 刚启动的过程会计算校验码。随后运行 hello 命令可以成功运行，也就是此时校验通过。但是当注入后再次运行，会提示该文件已被修改，也就是此时运行在此计算校验码和 os 刚启动时保存的校验码不一致。





Part B 任务三要求解析堆栈结构，检查堆栈返回地址是否合法。这里我们运行 attack 进行缓冲区溢出，实验结果如下。在这种检测方式下，我们认为只要 eip 不大于 0xF0000 就算合法，这是基于栈区往往在高地址而设置的。这种防御手段仅能防御住部分将代码放在栈区执行的缓冲区溢出攻击。对于我们在任务一实现的栈溢出攻击而言，第一种情况，也就是通过栈溢出跳转到代码段中的函数入口执行，这种防御手段无法防御，这种攻击手段常见于绕过保护验证；但对于第二种攻击手段，也就是构造 shellcode 放入栈区等到溢出后执行的攻击可以进行防御。



## 5 . 3 个人心得体会

本次参与完成了所有实验，感觉越做越还挺有意思的。写多级反馈队列的刚开始本想造一个队列， 然后用队列的操作去完成多级反馈调度，然后唐浩淼同学说既然是一个比较小型的多级反馈队列算法， 就不用搞那么麻烦，提出要不直接在进程数据结构中加入字段也可以同样实现，随后我和王浩翔和唐 浩淼三人完成了调度核心算法和打印性能评价。任务二加入自定义的功能，是我和庞紫萱同学在写密码学作业的时候，想起我们并没有调用太多 c 的库（除了 printf），于是我们就想到把 AES 和 DES 算法移植到 orangeos 中。但其实移植过程也那么轻松，有许多需要转化的地方，比如数字转字符串、字符串转数字，orangeos 的 printf 不支持十六进制输出我们又需要改动一些地方，工作量也不小。这一部分最后由我和庞紫萱和王浩翔三人共同完成。任务三 shell 多进程是我们在图书馆四人一起讨论出来的，这一部分我们改动的逻辑还是比较多的，我们考虑了如何全让父进程 fork、如何 fork 结束后子进程能够进行多任务调度。写这一部分代码很容易一运行就不会再回显 $ 输入下一条指令，也就是很容易卡在某个地方，我们小组是花了一个上午四人一起不断调试最终成功的。任务四分页刚开始在图书 馆讨论的时候放弃掉了，后来自己一个人回去想了下，感觉开启分页不难，就开始尝试了，不过就像实验报告所说，尝试到开启分页后，就不知道如何实现动态分配内存。偶然机会再上一届学长学姐的帮助下又进一步完成了一些，但是还没有完全完成。Part B 部分都是我和王浩翔两人一起完成，其实我们选了 linux 的课程，对缓冲区溢出早早做过实验，所以任务一对于我们并不难写。任务二也是非常简单， 我们很幸运在之前就移植了 AES 算法，所以我们在这里用了 AES 做了校验和，效果会比奇偶校验更好。任务三我们俩也对题目做了简化，我们刚开始看有点不太理解合法到底是什么意思，最后也是做了简化简单处理完成了任务三。

# 六、 指导教师评语及成绩

【评语】

成 绩： 指导老师签名： 批阅日期：