

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	庞紫萱	学号	2019301040155	专 业	信息安全	年级班级	19 级 8 班

目录

一、实验内容	1
1.1 实验选题内容	1
1.1.1 Part A 综合装配	1
1.1.2 Part B 安全分析与防御	2
1.2 分工情况	3
二、实验环境	3
三、实验方案设计	3
3.1 Part A 任务二	3
3.2 Part A 任务三	3
3.2.1 wait 和 exit	6
四、实验过程分析	7
4.1 Part A 任务二	7
4.2 Part A 任务三	12
五、实验结果总结	16
5.1 准备工作	16
5.2 实验结果	17
5.3 实验体会	19
六、指导教师评语及成绩	20

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

1.1.1.2 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？

1.2 分工情况

我写的是 Part A 的任务二和任务三

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。

3.2 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 进程打开了两个 shell，分别运行在 TTY1 和 TTY2 上。

```
1 void Init()  
2 {  
3     int fd_stdin = open("/dev_tty0", O_RDWR);  
4     assert(fd_stdin == 0);  
5     int fd_stdout = open("/dev_tty0", O_RDWR);  
6     assert(fd_stdout == 1);  
7  
8     printf("Init() is running ...\n");  
9 }
```

```
10  /* extract 'cmd.tar' */
11  untar("/cmd.tar");
12
13  char * tty_list[] = {"/dev_tty1", "/dev_tty2"};
14
15  int i;
16  for (i = 0; i < sizeof(tty_list) / sizeof(tty_list[0]); i++) {
17      int pid = fork();
18      if (pid != 0) { /* parent process */
19          printf("[parent is running, child pid:%d]\n", pid);
20      }
21      else { /* child process */
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) {
32      int s;
33      int child = wait(&s);
34      printf("child (%d) exited with status: %d.\n", child, s);
35  }
36  assert(0);
37 }
```

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_shell 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  {
3      int fd_stdin = open(tty_name, O_RDWR);
4      assert(fd_stdin == 0);
5      int fd_stdout = open(tty_name, O_RDWR);
6      assert(fd_stdout == 1);
7
8      char rdbuf[128];
9
10     while (1) {
```

```
11     write(1, "$ ", 2);
12     int r = read(0, rdbuf, 70);
13     rdbuf[r] = 0;
14
15     int argc = 0;
16     char * argv[PROC_ORIGIN_STACK];
17     char * p = rdbuf;
18     char * s;
19     int word = 0;
20     char ch;
21     do {
22         ch = *p;
23         if (*p != ' ' && *p != 0 && !word) {
24             s = p;
25             word = 1;
26         }
27         if ((*p == ' ' || *p == 0) && word) {
28             word = 0;
29             argv[argc++] = s;
30             *p = 0;
31         }
32         p++;
33     } while(ch);
34     argv[argc] = 0;
35
36     int fd = open(argv[0], 0_RDWR);
37     if (fd == -1) {
38         if (rdbuf[0]) {
39             write(1, "{", 1);
40             write(1, rdbuf, r);
41             write(1, "}\n", 2);
42         }
43     }
44     else {
45         close(fd);
46         int pid = fork();
47         if (pid != 0) { /* parent */
48             int s;
49             wait(&s);
50         }
51         else { /* child */
52             execv(argv[0], argv);
53         }
54     }
```

```
54     }  
55 }  
56  
57 close(1);  
58 close(0);  
59 }
```

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。

3.2.1 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）。

四、 实验过程分析

4.1 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  /*
2   * The DES function
3   * plaintext: 64bits message
4   * key:      64bits key for encryption/decryption
5   * mode:    'e' = encryption, 'd' = decryption
6   */
7  u64 DES(u64 plaintext, u64 key, char mode) {
8      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 /* Decompose T64 into L and R parts */
14 u32 L = 0;
15 u32 R = 0;
16 L = (u32)(init_perm_res >>32) & L64_MASK;
17 R = (u32)(init_perm_res)&L64_MASK;
18
19 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;
69 DES_PT(pre_output, PI, ct)
70 return ct;
71 }
```

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](#)。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;
18     add_round_key(state, rk->words + 0);
19 } else if (mode == 'd') {
20     pn = -1;
21     tot = Nb * (Nr + 1);
22     d = 1;
23     AES_TB0 = RT0;
24     AES_TB1 = RT1;
25     AES_TB2 = RT2;
26     AES_TB3 = RT3;
27     AES_SB = RSb;
28     add_round_key(state, rk->words + 40);
29 }
30
31 u8 t[Nk * Nb] = {0};
32 for (int i = 0; i < Nr - 1; i++) {
33     for (int j = 0; j < Nb; j++) {
34         u32 temp = AES_TB0[state[0 + ((j + pn * c0 + Nb) % Nb) * 4]] ^
35             AES_TB1[state[1 + ((j + pn * c1 + Nb) % Nb) * 4]] ^
36             AES_TB2[state[2 + ((j + pn * c2 + Nb) % Nb) * 4]] ^
37             AES_TB3[state[3 + ((j + pn * c3 + Nb) % Nb) * 4]] ^
38             rk->words[tot + pn * (i + 1 + d) * 4 + j];
39
40         temp = ((temp & 0xFFFF0000) >>16) | ((temp & 0x0000FFFF) <<16);
41         temp = ((temp & 0xFF00FF00) >>8) | ((temp & 0x00FF00FF) <<8);
42
43         *(u32*)(t + j * 4) = temp;
44     }
45     _copy(state, sizeof(state), t, sizeof(state));
46 }
47
48 for (int j = 0; j < Nb; j++) {
49     t[4 * j + 0] = AES_SB[state[0 + ((j + pn * c0 + Nb) % Nb) * 4]] ^
50         ((u8)(rk->words[tot + pn * (Nb * (Nr + d)) + j] >>24));
51     t[4 * j + 1] = AES_SB[state[1 + ((j + pn * c1 + Nb) % Nb) * 4]] ^
52         ((u8)(rk->words[tot + pn * (Nb * (Nr + d)) + j] >>16));
53     t[4 * j + 2] = AES_SB[state[2 + ((j + pn * c2 + Nb) % Nb) * 4]] ^
54         ((u8)(rk->words[tot + pn * (Nb * (Nr + d)) + j] >>8));
55     t[4 * j + 3] = AES_SB[state[3 + ((j + pn * c3 + Nb) % Nb) * 4]] ^
56         ((u8)(rk->words[tot + pn * (Nb * (Nr + d)) + j] >>0));
57 }
58 _copy(out, sizeof(state), t, sizeof(state));
```

53 }

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 }
```

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 {
3     MESSAGE msg;
4     struct time t;
5
6     while (1) {
7         send_recv(RECEIVE, ANY, &msg);
8         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 }
```

4.2 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);
13     int r = read(0, rdbuf, 70);
14     rdbuf[r] = 0;
15
16     int argc = 0;
17     char* argv[PROC_ORIGIN_STACK];
18     char* p = rdbuf;
19     char* s;
20     int word = 0;
21     char ch;
22     do {
23         ch = *p;
24         if (*p != ' ' && *p != 0 && !word) {
25             s = p;
26             word = 1;
27         }
28         if ((*p == ' ' || *p == 0) && word) {
29             word = 0;
30             argv[argc++] = s;
31             *p = 0;
32         }
33         p++;
34     } while (ch);
35     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 */
54      for (int i = 0; i < argc; i++) {
55          if (strcmp(argv[i], "&")) {
56              /* 如果遇到的不是&, 那么把字符串放入数组 */
57              multi_argv[num_proc - 1][sec_count++] = argv[i];
58          } 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 */
62              num_proc++;
63              sec_count = 0;
64              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 */
66                  error = 1;
67                  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 */
83          int err_cmd = 0;
84          for (int i = 0; i < num_proc; i++) {
85              int fd = open(multi_argv[i][0], O_RDWR);
86              if (fd == -1) {
87                  err_cmd = 1;
88                  break;
89              }
90              close(fd);
91          }
```

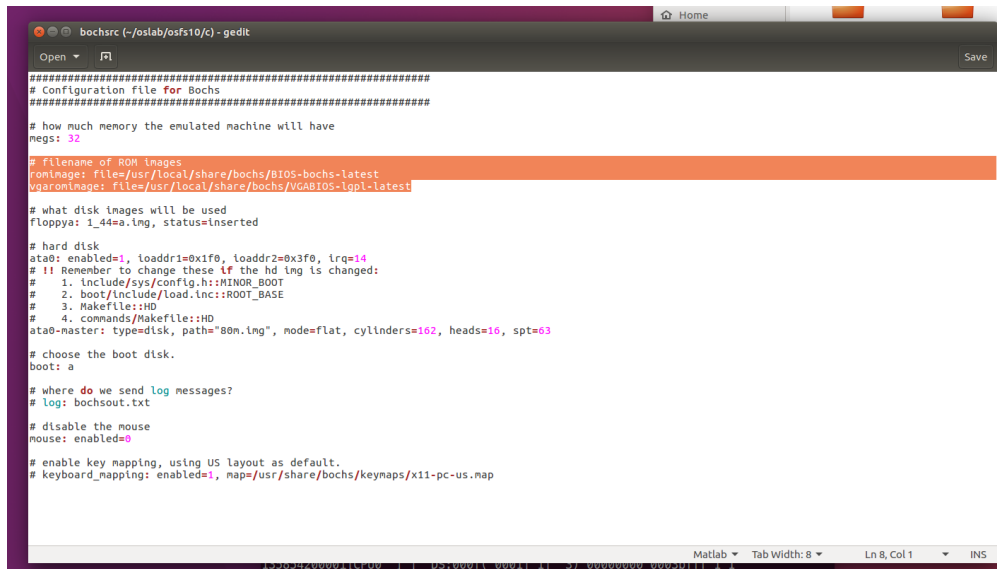
```
92
93     int i;
94     if (err_cmd) {
95         write(1, "{", 1);
96         write(1, rdbuf, r);
97         write(1, "}\n", 2);
98     } else {
99         /* 这一段代码要做到，所有进程都是由一个父进程fork出来的 */
100        for (i = 0; i < num_proc; i++) {
101            /* 父进程循环fork子进程 */
102            pid = fork();
103            /* 如果是子进程就退出循环，子进程不要进行fork */
104            if (pid == 0) break; // child exit for
105            /* 父进程保留子进程的pid */
106            pres_pid[i] = pid;
107            /* 随后父进程再次进入for循环fork子进程 */
108        }
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         * 那么就要遍历保留的子进程pid数组，将他们解除阻塞
124         * 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
125         * 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;
138         wait(&s);
139     }
140 } else if (pid == 0) { /* child */
141     /* 因此fork出来后的子进程应该主动把自己阻塞
142      * 等待父进程的解除阻塞
143      */
144     p_proc_ready->p_flags = 1;
145     block(p_proc_ready);
146
147     /* 这一部分是Part B 任务二部分，随后再解释 */
148     int position = find_position(check_table, multi_argv[i][0]);
149     u32 real_checkNum = check_table[position].checkNum;
150     u32 now_checkNum = check(multi_argv[i][0],
151                             check_table[position].byteCount);
152     // u32 now_checkNum = real_checkNum;
153
154     if (real_checkNum == now_checkNum) {
155         /* 子进程解除阻塞后就用execv执行命令
156          * 如此才会出现多进程同时运行的效果
157          * 而不是一个命令运行完再运行下一个命令
158          */
159         execv(multi_argv[i][0], multi_argv[i]);
160     } else {
161         printf("This file has been changed!\n");
162     }
163 }
164 }
165
166 close(1);
167 close(0);
168 }
```

五、实验结果总结

5.1 准备工作

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



```
bochsrc (~/.oslab/osfs10/c) - gedit
#####
# 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
vgaromimage: file=/usr/local/share/bochs/VGABIOS-lqpl-latest

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

# hard disk
ata0: enabled=1, loadaddr1=0x1f0, loadaddr2=0x3f0, lirq=14
# !! Remember to change these if the hd linq is changed:
# 1. include/sys/config.h: MINOR_BOOT
# 2. boot/include/load.inc: ROOT_BASE
# 3. Makefile: HD
# 4. commands/Makefile: HD
ata0-master: type=disk, path="80m.img", mode=flat, cylinders=162, heads=16, spt=63

# choose the boot disk.
boot: a

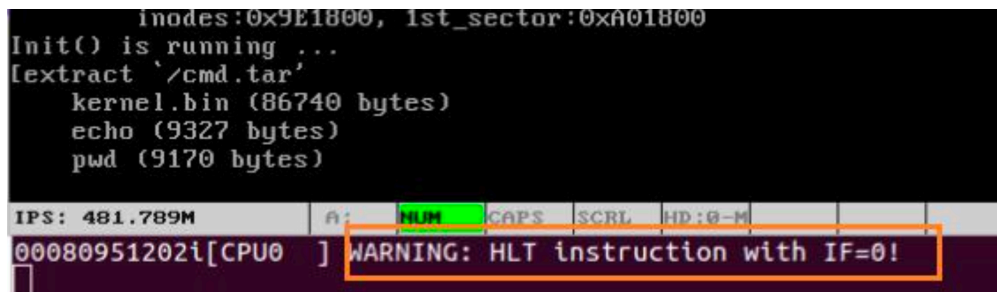
# where do we send log messages?
# log: bochsout.txt

# disable the mouse
mouse: enabled=0

# enable key mapping, using US layout as default.
# keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map
```

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。



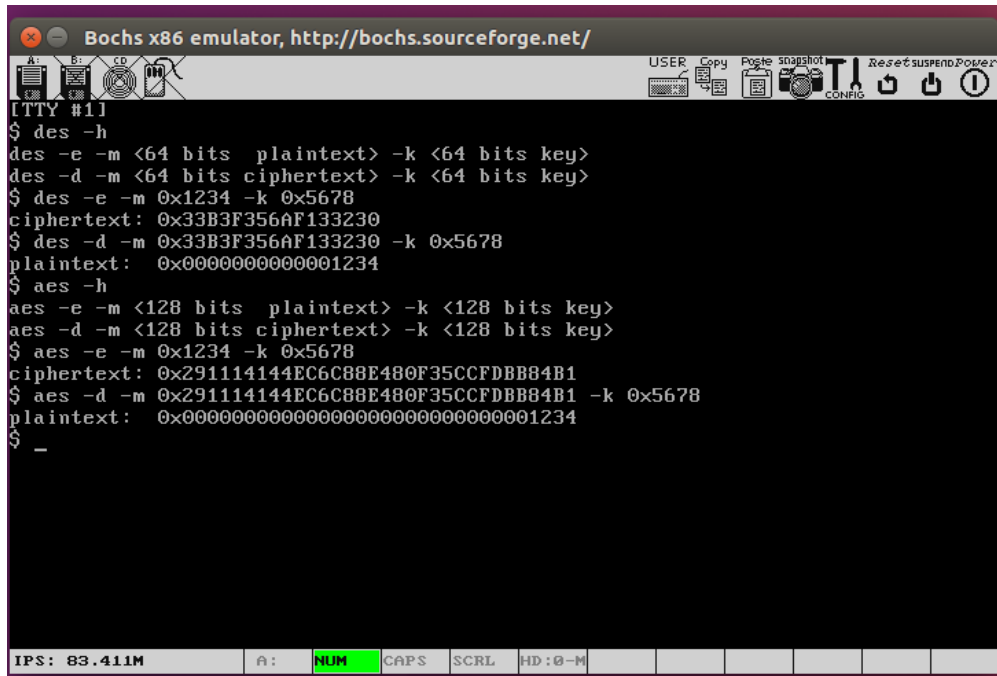
```
Inodes:0x9E1800, 1st_sector:0xA01800
Init() is running ...
[extract '/cmd.tar'
  kernel.bin (86740 bytes)
  echo (9327 bytes)
  pwd (9170 bytes)

IPS: 481.789M  A: NUM CAPS SCRL HD:0-M
00080951202i[CPU0 ] WARNING: HLT instruction with IF=0!
```

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

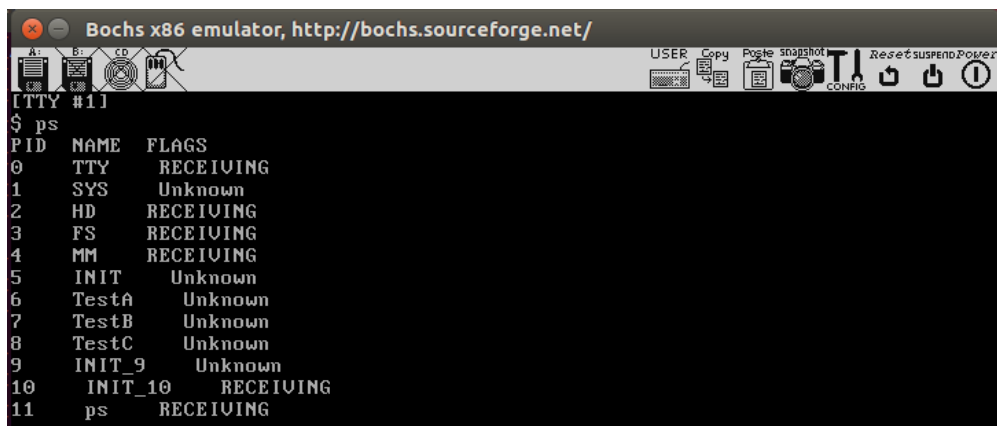
Part A 任务二实现 DES 和 AES 算法，输入 des -h 和 aes -h 可以获得帮助信息。加解密如下图所示，可以看见我们成功实现了加解密。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
[TTY #1]
$ des -h
des -e -m <64 bits plaintext> -k <64 bits key>
des -d -m <64 bits ciphertext> -k <64 bits key>
$ des -e -m 0x1234 -k 0x5678
ciphertext: 0x33B3F356AF133230
$ des -d -m 0x33B3F356AF133230 -k 0x5678
plaintext: 0x00000000000001234
$ aes -h
aes -e -m <128 bits plaintext> -k <128 bits key>
aes -d -m <128 bits ciphertext> -k <128 bits key>
$ aes -e -m 0x1234 -k 0x5678
ciphertext: 0x291114144EC6C88E480F35CCFDBB84B1
$ aes -d -m 0x291114144EC6C88E480F35CCFDBB84B1 -k 0x5678
plaintext: 0x0000000000000000000000000001234
$
-
```

IPS: 83.411M A: NUM CAPS SCRL HD: 0-M

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：



```
Bochs x86 emulator, http://bochs.sourceforge.net/
[TTY #1]
$ ps
PID  NAME  FLAGS
0    TTY   RECEIVING
1    SYS   Unknown
2    HD    RECEIVING
3    FS    RECEIVING
4    MM    RECEIVING
5    INIT  Unknown
6    TestA Unknown
7    TestB Unknown
8    TestC Unknown
9    INIT_9 Unknown
10   INIT_10 RECEIVING
11   ps    RECEIVING
```

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。

The screenshot shows a Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The window contains a terminal window with the following text:

```
[TTY #1]
$ des -e -m 0x12 -k 0x34 & aes -e -m 0x12 -k 0x34
ciphertext: 0x4F8B860B6EA02F7C
ciphertext: 0xF27150243DFE07A727580DC46DFB511
$ pwd & echo hello os & aes -h
/
hello os
aes -e -m <128 bits plaintext> -k <128 bits key>
aes -d -m <128 bits ciphertext> -k <128 bits key>
$ pwd & eecho hello & des -h
{pwd & eecho hello & des -h}
$ -
```

The emulator interface includes a toolbar with icons for USER, Copy, Paste, Snapshot, CONFIG, Reset, suspend, and Power. At the bottom, a status bar shows "IPS: 81.497M", "A:", "NUM" (highlighted in green), "CAPS", "SCRL", "HD: 0-M", and several empty slots.

5.3 实验体会

本学期的实验课程中，每节课我们都根据教材一步步进行操作，结合已经学习过的操作系统的理论知识在 bochs 下实现一个操作系统。刚开始配置 bochs 的时候出现了一些问题，好像是虚拟机的问题，换了一个虚拟机平台之后就解决了。之后的保护模式、页式存储等等每次都是磕磕绊绊但是还算都解决了，对我来说最困难的地方就是中断的部分，汇编代码太多了，理解起来比较困难。

本次大作业里添加应用程序任务将应用程序打包.tar 写入磁盘映像的某段特定扇区，重新 make



image。接着运行 bochs，新添的程序就出现在了列表中。实现多任务支持的任务中，基本思想就是 fork 多个子进程，子进程来运行那些命令，还有一些需要注意的是：用 ° 连接不同指令，在解析时将输入依据 ° 分割，将其连接的指令作为不同的子进程运行；子进程之间需要进行时钟中断调度，因此它们需要由同一个父进程 fork 得到，对于 fork 得到的 pid，父进程执行 wait()，而子进程则分别执行第一个和第二个指令。

通过本学期的学习，对简单的操作系统的搭建有了基本的理解和实践操作，虽然中间坎坷很多，但还是理解了一些最底层的东西，感觉上学期学习的东西都很理论化，在这学期的实际应用中通过复现代码、完成老师布置的小任务等等就感觉没有那么抽象了，非常感谢小组同学们，在遇到困难的时候都能很快想到很好的解决办法，还会一起讨论，让我这个学期的课程学习很有意义^^。

六、 指导教师评语及成绩

【评语】

成 绩：

指导老师签名：

批阅日期：