**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](#_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 分工情况](#_bookmark4) 3

[二、实验环境](#_bookmark5) 3

[三、实验方案设计](#_bookmark6) 3

[3 . 1 Part A 任 务二](#_bookmark7) 3

1. [. 2 Part A 任 务三](#_bookmark8) 3

[3 . 2 . 1 wait 和 exit](#_bookmark9) 6

[四、实验过程分析](#_bookmark10) 7

1. [. 1 Part A 任 务二](#_bookmark11) 7

[4 . 2 Part A 任 务三](#_bookmark12) 12

[五、实验结果总结](#_bookmark13) 16

[5 . 1 准备工作](#_bookmark14) 16

[5 . 2 实验结果](#_bookmark15) 17

[5 . 3 实验体会](#_bookmark16) 19

[六、指导教师评语及成绩](#_bookmark17) 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 . 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

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

，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");

了两

个

＼

1

2

3

4

5

6

7

8

9

10

/\* 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 \*/

printf("[child is running, pid:%d]\n", getpid()); close(fd\_stdin);

close(fd\_stdout);

shabby\_shell(tty\_list[i]); assert(0);

}

}

**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

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 \_)

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); |  |
| 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], O\_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

}

}

close(1);

close(0);

1.

}

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：

55

56

57

58

59 \_)

* 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。

## . 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

## . 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 /\*

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  13  14  15  16  17  18  19 | DES\_PT(plaintext, IP, init\_perm\_res)  /\* 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; |

1. AES\_TB3 = FT3;
2. AES\_SB = FSb;
3. add\_round\_key(state, rk->words + 0);
4. } **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 \_copy(out, **sizeof**(state), t, **sizeof**(state));

53 \_)

1.

}

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 . 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);

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 \*/

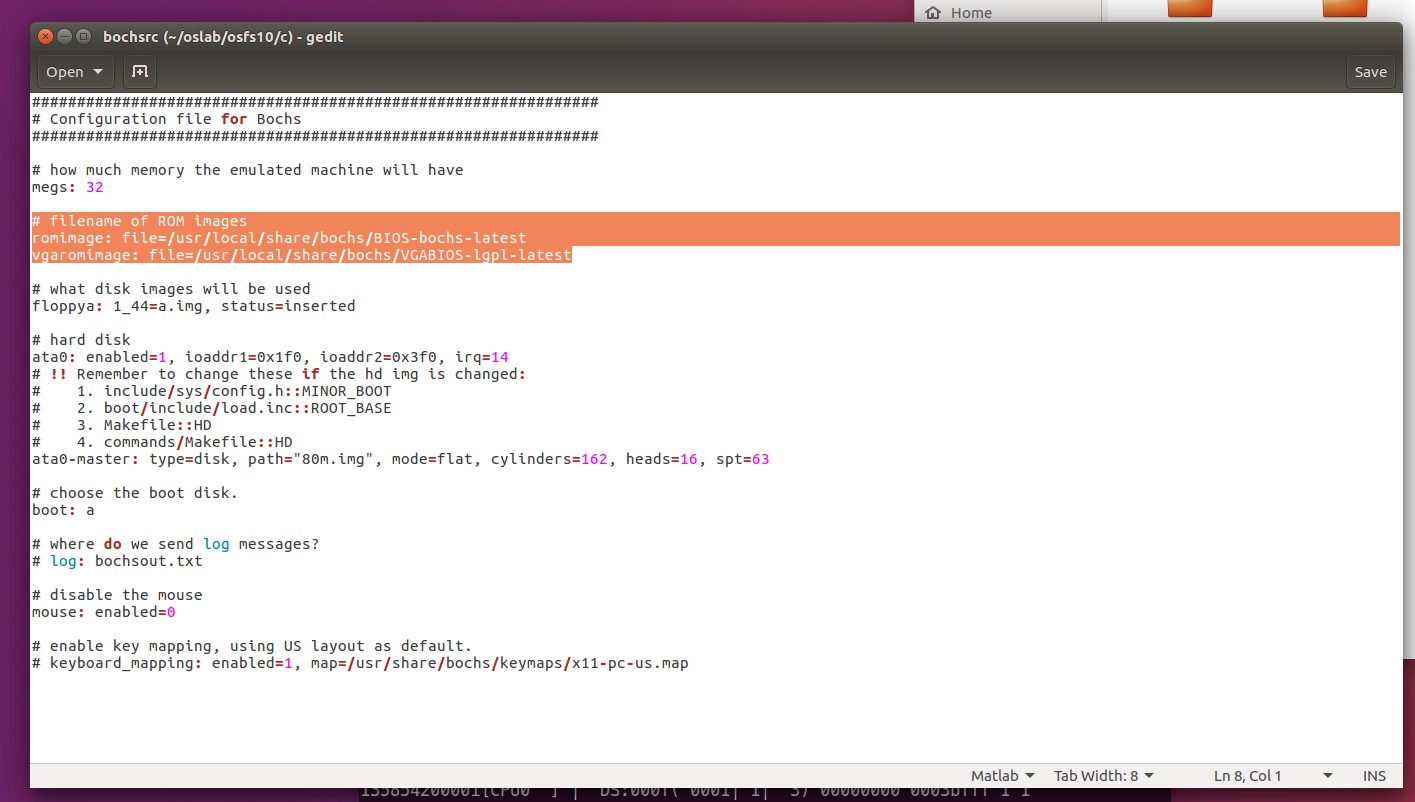
48 **int** num\_proc = 1;

|  |  |  |  |
| --- | --- | --- | --- |
| 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 \*/ |  | |
| 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  124  125  126 | * 那么就要遍历保留的子进程pid数组，将他们解除阻塞 * 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   \*/ | |
| 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 | |

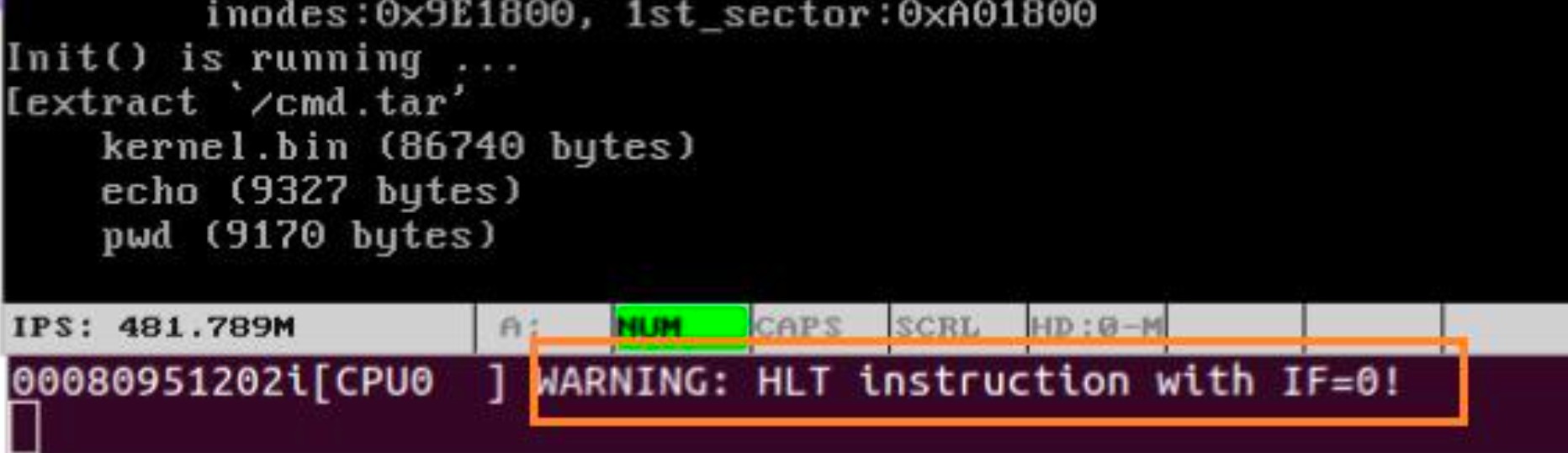
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 135 |  | | | | \*/ | |  |
| 136 |  | | | | **for** (**int** i = 0; i < num\_proc; i++) { | |
| 137 |  | | | | **int** s; | |
| 138 |  | | | | wait(&s); | |
| 139 |  | | | | } | |
| 140 | } | | | | **else if** (pid == 0) { /\* 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  145  146  147  148  149  150  151  152 | \*/  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; | | | | | |
| 153 | **if** | | | | | (real\_checkNum == now\_checkNum) { |
| 154 |  | | | | | /\* 子进程解除阻塞后就用execv执行命令 |
| 155 |  | | | | | \* 如此才会出现多进程同时运行的效果 |
| 156 |  | | | | | \* 而不是一个命令运行完再运行下一个命令 |
| 157  158 | \*/  execv(multi\_argv[i][0], multi\_argv[i]); | | | | | |
| 159 |  | |  |  | } | **else** { |
| 160 |  | |  |  |  | printf("This file has been changed!\n"); |
| 161 |  | |  |  | } |  |
| 162 |  | |  | } |  |  |
| 163 |  | | } |  |  |  |
| 164 | } | |  |  |  |  |
| 165 |  | |  |  |  |  |
| 166 |  | close(1); | | | |  | |
| 167 |  | close(0); | | | |  | |
| 168 | 1.} |  | | | | \_) | |
| **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。

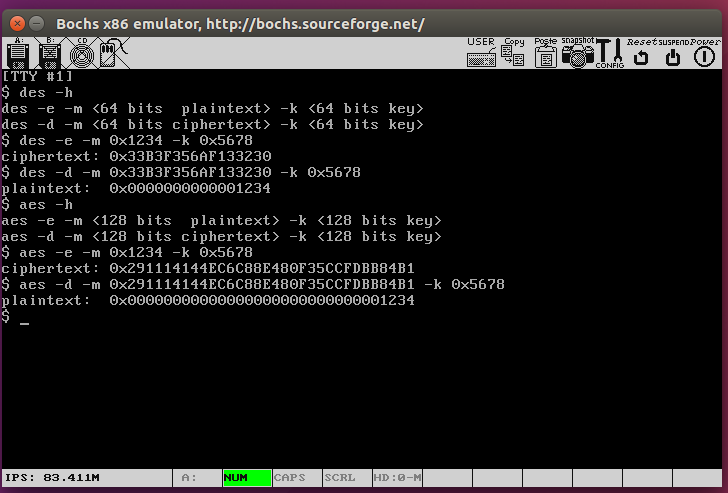


最后需要在 kernel/tty.c 中，将切换 shell 的功能键 ALT 改为 CTRL。这是因为 ALT 在 ubuntu

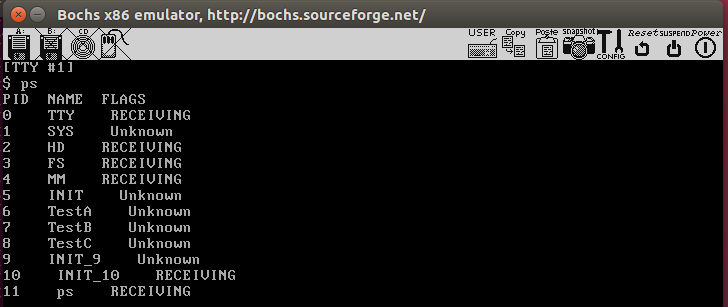
中是特殊的功能键，所以我们用 CTRL 替换了该键。

## 5 . 2 实验结果

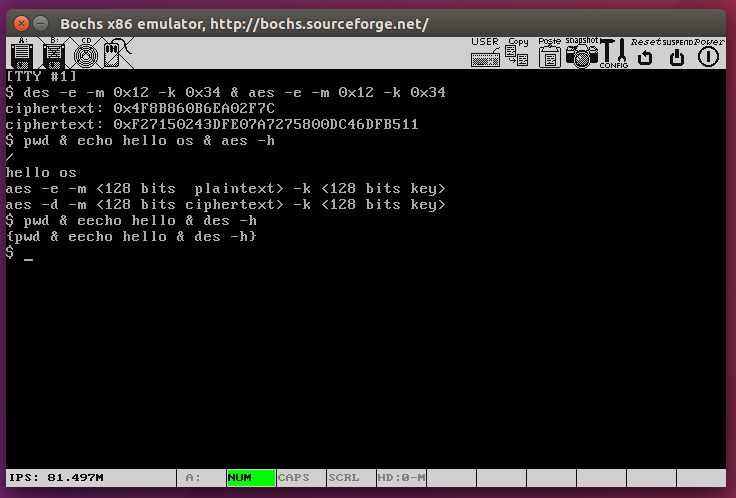
Part A 任务二实现 DES 和 AES 算法，输入 des -h 和 aes -h 可以获得帮助信息。加解密如下图所示，可以看见我们成功实现了加解密。

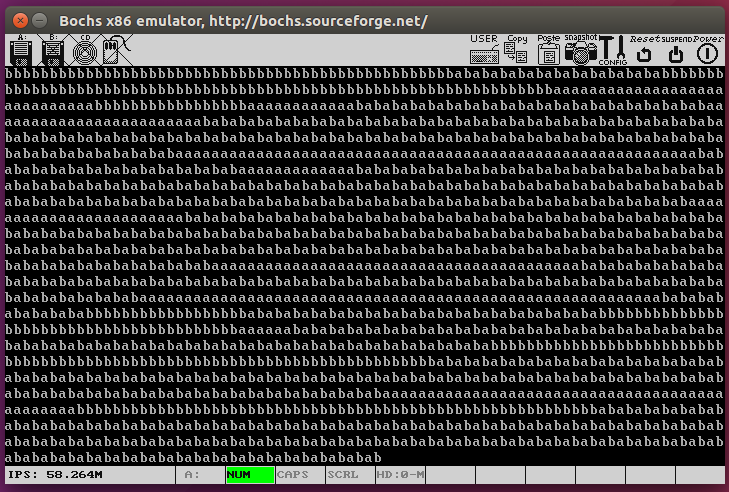


Part A 任务二我们还实现了类似 linux 的 ps 命令，可以看到我们成功利用消息传递机制输出了进程信息：



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。





## 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好了。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ˆˆ。

# 六、 指导教师评语及成绩

【评语】

成 绩： 指导老师签名： 批阅日期：