|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 武汉大学国家网络安全学院教学实验报告  目录  [1.1 实验目的 2](#_Toc182835666)  [1.2 实验内容 2](#_Toc182835667)  [2.1 实验环境 4](#_Toc182835669)  [2.2 实验步骤 4](#_Toc182835670)  [2.2.1 读取键盘输入并显示，创建键盘缓冲区、键盘中断程序将键盘输入写入缓冲区、tty 任务读取缓冲区并根据扫描码解析，显示结果。 4](#_Toc182835671)  [2.2.2 显示器输出，通过写CRTController Data 寄存器组指定光标位置、设置显示地址、向上向下翻页 5](#_Toc182835672)  [2.2.3 搭建TTY基本架构 8](#_Toc182835673)  [2.2.4 设置不同程序处于不同特权级 11](#_Toc182835674)  [2.2.5 实现printf() 11](#_Toc182835675)  [2.2.6 解释 Scancode、MakeCode、BreakCode 的区别 16](#_Toc182835676)  [2.2.7 为什么在/c/的 kliba.asm，代码 7.12 中，需要 disable\_int, enable\_int 17](#_Toc182835677)  [2.2.8 解释一下，重新设置显示开始地址的原理 17](#_Toc182835678)  [2.2.9 动手做 1：请添加一个你自己个性化的TTY，在这个TTY上，你可以根据键 盘输入或者移动光标的某种规律，运行一个你的彩蛋程序，而在其他 TTY 中 不会有这个效果 17](#_Toc182835679)  [2.2.10 动手做 2：尝试扩展一下printf，让它支持%s，想想目前的 printf 实现是否有什么安全漏洞？可以怎么解决 18](#_Toc182835680)  [参考实验题中的实现printf 的“%x”功能，进入 vsprintf.c 函数，填充 case‘s’部分。 18](#_Toc182835681)  [函数vsprintf 虽然它只识别%x 这一种格式，但其他格式的原理也是一样的，即根 据%后的格式字符就能判断下一个参数的类型，从而知道从堆栈中取出什么。 18](#_Toc182835682)  [va\_arg 功能：获取当前指针所指的可变参数并将并将指针移向下一可变参数。 18](#_Toc182835683)  [自定义insert\_str() 函数：将参数的内容传到p中。 18](#_Toc182835684)  [3.1 王亚鹏： 21](#_Toc182835686)  [3.2 杨依磊： 22](#_Toc182835687)  [键盘输入的处理: 通过实验，我了解了Scancode 的分类（MakeCode、BreakCode）及它们的不同作用，我分析了实验中 disable\_int 和 enable\_int 的使用，以及中断处理在保证数据一致性中的作用。 显示器输出的实现: 我利用重新设置显示开始地址的原理，进行了显存的使用和光标位置的控制。理解了文本模式字符输出的底层原理，尤其是操作系统如何与硬件协调完成显示任务。 TTY 终端的并发机制: 我通过多个 TTY 并发处理输入输出的流程，学习了任务调度和资源隔离机制。并与杜洪波讨论并共同动手创建个性化 TTY，添加Ctrl + Enter 触发显示“hello,randi”字符串。 printf 的扩展: 我通过讨论，理解了 printf 的底层实现，尝试扩展其功能以支持 %s 格式符，并发现了当前实现中潜在的缓冲区溢出问题。提出了通过边界检查并动态分配缓冲区来提升安全性的改进方案。 22](#_Toc182835688)  [3.3杜泓波： 22](#_Toc182835689)  [3.3 侯名扬： 23](#_Toc182835690) | | | | | |
| 课程名称 | 操作系统设计与实践 | | 实验日期 | | 2024.11.14 |
| 实验名称 | I/O子系统 | | 实验周次 | | 11 |
| 姓名 | 学号 | | 专业 | | 班级 |
| 王亚鹏 | 2022302181161 | | 信息安全 | | 5班 |
| 杨依磊 | 2022302181159 | | 信息安全 | | 5班 |
| 杜泓波 | 2022302181162 | | 信息安全 | | 5班 |
| 侯名扬 | 2022302181165 | | 信息安全 | | 5班 |
| 1. 实验目的及实验内容   （要求掌握的知识；实验内容；原理分析）   * 1. 实验目的   1. 熟悉键盘输入的基本方法  2. 熟悉显示器输出的基本方法  3. 掌握TTY(TeleTYpe) 任务子系统的基本功能和方法  4. 合理配置不同程序处于不同特权级  5. 掌握Printf 函数的实现方法   * 1. 实验内容  1. 验证并解释键盘输入的处理程序 2. 解释Scancode、Makecode、BreakCode的区别 3. 解释键盘输入缓冲区的作用以及处理过程 4. 解释在/c/的kliba.asm，代码7.12中，需要disable\_int，enable\_int 5. 解释重新设置显示开始地址的原理 6. 解释多个tty能够并发处理输入输出的原理并给出流程图 7. 动手做1：请添加一个你自己个性化的TTY，在这个TTY上，你可以根据键盘输入或者移动光标的某种规律，运行一个你的彩蛋程序，而在其他TTY中不会有这个效果 8. 动手做2：尝试扩展一下printf，让它支持%s，想想目前的printf实现是否有什么 安全漏洞？可以怎么解决 | | | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体实验步骤） | | | | | |
| 1. 1. 实验环境   VMware Workstation pro + Ubuntu 16.04.1 + bochs 2.7   * 1. 实验步骤      1. 读取键盘输入并显示，创建键盘缓冲区、键盘中断程序将键盘输入写入缓冲区、tty 任务读取缓冲区并根据扫描码解析，显示结果。   首先添加一个简单的中断处理程序，创建文件keyboard.c,结果是每按一次按键打印一个‘\*’，像是在输入密码。我们注释其他进程的输处，在proto.h中声明init\_keyboard()并调用，修改Makefile，make并运行。  发现只出现一个‘\*’就不再响应。  其次打印收到的扫描码，进一步修改keyboard\_handler，在运行时按下‘h’和‘enter’。出现字符‘h’和‘enter’的MakeCode和BreakCode：  0x23,0xA3,0x1C,0x9C。  我们建立数组，以扫描码为下标，对应的元素就是相应的字符。建立一个缓冲区，用于放置中断例程中的扫描码。接着对缓冲区进行添加操作，对Kb\_in的成员进行初始化，代码在init\_keyboard()中。为了保持kernel\_main() 的整洁，我们把时钟中断的设定和开启也放到单独的函数 init\_clock() 中。添加解析扫描码功能，我们由简到繁，先挑能打印者打印一下。运行结果如下：    成功输入了‘randi2022302181165“ENTER”’。   * + 1. 显示器输出，通过写CRTController Data 寄存器组指定光标位置、设置显示地址、向上向下翻页        1. 基本概念   开机看到的默认模式为80x25文本模式。在这种模式下，显存大小为32KB，范围为0xB8000～0xBFFFF。每2字节代表一个字符，其中低字节表示字符的ASCII 码，高字节表示字符的属性。一个屏幕总共可以显示25行，每行80个字符，每一个字符对应的2字节的定义如图所示：   * + - 1. 相关寄存器   以VGA系统为例，有6组寄存器，分别为GeneralRegisters、Sequencer Registers、 CRT Controller Registers、Graphics Controller Registers、Attribute Controller Registers、 Video DAC Palette Registers  这里我们以CRT Controller Registers 寄存器组为例，来实现指定光标的位置、设 置显示地址以及向上向下翻页的功能。该寄存器组的数据寄存器有25个寄存器，如下图    每一个寄存器都对应一个索引值，当想要访问其中一个时，只需要先向 Address Register 写对应的索引值（通过端口 0x3D4），然后再通过端口 0x3D5 进行的操作就是针对索引值对应的寄存器了   * + - 1. 光标跟随字符   Cursor Location High Register 和 Cursor Location Low Register 是用来设置光标位置的，索引号分别是0Eh和0Fh。这里我们修改Cursor Location High Register 和 Cursor Location Low Register 来改变光标位置。   1. disable\_int () ; 2. out\_byte(CRTC\_ADDR\_REG, CURSOR\_H); 3. out\_byte(CRTC\_DATA\_REG, ((disp\_pos/2)>>8)&0xFF) ; 4. out\_byte(CRTC\_ADDR\_REG, CURSOR\_L); 5. out\_byte(CRTC\_DATA\_REG, (disp\_pos/2)&0xFF) ; 6. enable\_int() ;   make运行后发现光标跟随字符。   * + - 1. 设置开始显示地址   Start Address High Register 和 Start Address Low Register 是用来设置显示开始地址的。这里我们修改Start Address High Register 和 Start Address Low Register 来显示地址，从而实现滚屏的功能。我们在代码中增加按住 shift+↑ 能够更改显示地址的操作。修改寄存器的方法与上面是一致的。   1. PUBLIC **void** in\_process(u32 key) 2. { 3. **char** output[2] = {'\0', '\0'}; 5. **if** (!(key & FLAG\_EXT)) { 6. . . . 8. **else** { 9. **int** raw\_code = key &MASK\_RAW; 10. **switch**(raw\_code) { 11. **case** UP: 12. **if** ((key & FLAG\_SHIFT\_L) || (key & FLAG\_SHIFT\_R)) { 13. disable\_int () ; 14. out\_byte(CRTC\_ADDR\_REG, START\_ADDR\_H); 15. out\_byte(CRTC\_DATA\_REG, ((80∗15) >> 8) & 0xFF); 16. out\_byte(CRTC\_ADDR\_REG, START\_ADDR\_L); 17. out\_byte(CRTC\_DATA\_REG, (80∗15) & 0xFF); 18. enable\_int() ; 19. } 20. **break**; 21. **case** DOWN: 22. **if** ((key & FLAG\_SHIFT\_L) || (key & FLAG\_SHIFT\_R)) { 23. /∗ Shift+Down, **do** nothing ∗/ 24. } 25. **break**; 26. **default** : 27. **break**; 28. } 29. } 30. }   Make image运行发现实现更改显示地址的功能。   * + 1. 搭建TTY基本架构   TTY 和Console 的结体如下：  整体流程图如下：  1. task\_tty( ) 中，通过循环来处理每一个 TTY 的读和写操作  2. 读写操作全都放在tty\_do\_read(),tty\_do\_write()两个函数中，这样就让task\_tty( ) 很简洁，而且逻辑清晰  3. 读操作会调用keyboard\_read( ) ，此时已经多了一个参数  4. 写操作会调用out\_char( ) ，它会将字符写入指定的CONSOLE  32KB 的显存可以允许同时存在 3个控制台，所以声明3个TTY和对应的3个 CONSOLE。  task\_tty( ) 框架实现如下：  其中nr\_current\_console 是一个定义在const.h的全局变量EXTERN int nr\_current\_console，它可以用来记录当前的控制台是哪一个。只有当某个TTY对应的控制台是当前控制台时，它才可以读取键盘缓冲区。在这个主循环中，先做了初始化工作。为其中的缓冲区设置初值和指定对应的CONSOLE  tty\_do\_read( ) 要判断 nr\_current\_console 的值，对控制台进行切换时也要改变 nr\_current\_console：  key\_board( ) 改变了，要对其函数体进行修改。同时in\_process( ) 也要增加参数p\_tty 。往TTY缓冲区写入数据的代码很简单，只把输出字符写入缓冲区，读操作就结束了；如果遇到Alt+Fn切换控制台的操作，也要在in\_process 中处理掉  看一下写操作，它从TTY缓冲区中中取出值，类似get\_byte\_from\_kbuf( ) ，然后用out\_char( ) 显示在 CONSOLE 中  out\_char( ) 实现如下：V\_MEM\_BASE=0xB8000定义在const.h中，V\_MEM\_BASE + disp\_pos 等同于当前显示位置的地址，我们直接把字符写入特定地址   * + 1. 设置不同程序处于不同特权级   我们有了4个进程——TTY,A,B,C，后三者可有可无，它们是“用户进程”，而TTY是“任务”。具体实现上，让用户进程运行在ring3，任务运行在ring1，ring0 运行的是进程调度,图示如下  增加对NR\_PROCS的定义，同时修改NR\_TASKS=1；其次，在所有用到NR\_TASKS的地方进行修改。新声明一个数组user\_proc\_table[ ]；然后，修改初始化进程表的代码。这里改变了用户进程的特权级，而且通过eflags 的改变，取消了用户进程所有的IO权限；另外，protect.c 中初始化 GDT 中 LDT 描述符的代码也进行了修改；最后，区分任务和用户进程后的proc.c进程调度代码。  Make image运行结果如下：   * + 1. 实现printf()        1. 为进程指定TTY   当某个进程调用printf( ) 时，操作系统必须知道往哪个控制台输出才 行。而当系统调用发生，ring3跳入ring0时，系统只能知道当前系统调用是由哪个进程 触发的。所以，我们必须为每个进程指定一个与之相对应的TTY，这可以通过在进程 表中增加一个成员intnr\_tty 来实现。  所有进程的nr\_tty被初始化成0，这样，所有进程默认与第0个TTY绑定。不过 在后面，B和C两个进程与第1个TTY绑定。这意味着，将来B和C的输出将同时 出现在控制台1，而A的输出出现在控制台0。   * + - 1. 实现printf   下面的printf只支持“%x”一种格式，在动手做中我们会实现第二种格式“%s”。   1. /∗======================================================================∗ 2. printf 3. ∗=====================================================================/ 4. **int** printf(**const** **char** ∗fmt, ...) 5. { 6. **int** i ; 7. **char** buf[256]; 9. **va\_list** arg=(**va\_list**)((**char**∗)(&fmt)+4); /∗4是参数fmt所占堆栈中的大小∗/ 10. i =vsprintf(buf, fmt, arg); 11. write(buf, i); 13. **return** i ; 14. }   其中，vsprintf()的实现方法如下：   1. **int** vsprintf(**char** ∗buf, **const** **char** ∗fmt, **va\_list** args) 2. { 3. **char**∗ p; 4. **char** tmp[256]; 5. **va\_list** p\_next\_arg=args; 7. **for** (p=buf;∗fmt;fmt++) { 8. **if** (∗fmt !='%') { 9. ∗p++=∗fmt; 10. **continue**; 11. } 13. fmt++; 15. **switch** (∗fmt) { 16. **case** 'x': 17. itoa(tmp, ∗((**int**∗)p\_next\_arg)); 18. strcpy(p, tmp); 19. p\_next\_arg+=4; 20. p+=strlen(tmp); 21. **break**; 22. **case** 's': 23. **break**; 24. **default**: 25. **break**; 26. } 27. **return** (p − buf); 28. }   这里遇到可变参数这个问题，在c语言中都是后面的参数先入栈，并且由调用者清 理堆栈。 C 调用约定的好处在处理可变参数函数时得到了充分体现，因为只有调用者知道此 次调用包含几个参数，于是可以方便地清理堆栈。 printf 使用了它的第一个参数 fmt 作为基准，得到了后面若干参数的开始地址，这 样，其值也就容易得到了  &fmt 表示fmt 的地址，(char\*)(&fmt)+4 则表示紧随 fmt 之后的参数， 即var1 的地址。所以，接下来实际上是将var1的地址传递给了紧接着调用的vsprintf。 va\_list 其实就是 char\*，它的定义在 type.h 中。   * + - 1. 系统调用write()   下面是增加一个系统调用的流程:  我们把这个新增的系统调用取名为write()，把它对应的内核部分取名为sys\_write()， 它们的声明在proto.h 中。  添加write( ) 和 sys\_write( ) 这两个函数体:   1. write: 2. mov     eax , \_NR\_write 3. mov     ebx, [esp + 4] 4. mov     ecx , [esp + 8] 5. **int**        INT\_VECTOR\_SYS\_CAL 6. ret 7. /∗======================================================================∗ 8. tty\_write 9. ∗======================================================================∗/ 10. PUBLIC **void** tty\_write(TTY∗ p\_tty, **char**∗ buf, **int** len) 11. { 13. **char**∗ p = buf; 14. **int** i = len; 15. **while** (i) { 16. out\_char(p\_tty−>p\_console , ∗p++); 17. i−−; 18. } 19. } 21. /∗======================================================================∗ 22. sys\_write 23. ∗======================================================================∗/ 24. PUBLIC **int** sys\_write(**char**∗ buf, **int** len , PROCESS∗ p\_proc) 25. { 26. tty\_write(&tty\_table [p\_proc−>nr\_tty] , buf , len); 27. **return** 0; 28. }     * + 1. 使用printf（）   这样，我们的第二个系统调用printf( ) 就完成了。下面在3个用户进程中调用它。   1. /∗======================================================================∗ 2. TestA 3. ∗======================================================================∗/ 4. **void** TestA() 5. { 6. **int** i = 0; 7. **while** (1) { 8. printf ("<Ticks:%x>" , get\_ticks()) ; 9. milli\_delay(200) ; 10. } 11. } 13. /∗======================================================================∗ 14. TestB 15. ∗======================================================================∗/ 16. **void** TestB() 17. { 18. **int** i = 0x1000; 19. **while**(1){ 20. printf ("B") ; 21. milli\_delay(200) ; 22. } 23. } 25. /∗======================================================================∗ 26. TestC 28. ∗======================================================================∗/ 29. **void** TestC() 30. { 31. **int** i = 0x2000; 32. **while**(1){ 33. printf ("C") ; 34. milli\_delay(200) 35. } 36. }   运行结果如下：  0:  1:  printf 的调用过程如下:   * + 1. 解释 Scancode、MakeCode、BreakCode 的区别   敲击键盘所产生的编码被称作扫描码 Scan Code ，它分为 Make Code 和 Break Code 两类。当一个键被按下或者保持住按下时，将会产生MakeCode；当键弹起时，产 生Break Code。除了 Pause 键之外，每一个按键都对应一个 Make Code 和一个 Break Code 。扫描码总共有三套，叫做Scan code set 1、Scan code set 2、Scan code set 3 。 Scan code set 1 是早期的 XT 键盘使用的，现在的键盘默认都支持Scancode set 2，而 Scan code set 3 很少使用。   * + 1. 为什么在/c/的 kliba.asm，代码 7.12 中，需要 disable\_int, enable\_int   在c文件中，我们需要添加一个任务，它将来不仅会处理键盘操作，还将处理诸如 屏幕输出等内容，这些操作共同组成同一个任务：终端任务。现在，功能为不停地调用 keyboard.c 中的函数 keyboard\_read ，在c文件中，我们需要添加一个任务，它将来不仅会处理键盘操作，还将处理诸如 屏幕输出等内容，这些操作共同组成同一个任务：终端任务。现在，功能为不停地调用 keyboard.c 中的函数 keyboard\_read ，发现其只是实现简单的开中断与关中断请求，但由于避免了调用函数的call指令和 返回时的ret 指令，因此更加节省时间。   * + 1. 解释一下，重新设置显示开始地址的原理   显存有32KB，每个屏幕才占4KB，所以显存中足以存放8个屏幕的数据。改变开 始地址能够显示出不同显存段的字符内容。  Data Registers 中有 Start Address High Register 和 Start Address Low Register 两 个寄存器来表示显示开始的地址。我们分别向指定端口0x3D4写入索引号指向这两个 24 操作系统实验报告 第五组 寄存器，并向指定端口0x3D5写入某一值，如此可以实现开始地址的变化。然后屏幕在 显示时，就能够从新的开始地址来显示内容。   * + 1. 动手做 1：请添加一个你自己个性化的TTY，在这个TTY上，你可以根据键 盘输入或者移动光标的某种规律，运行一个你的彩蛋程序，而在其他 TTY 中 不会有这个效果   个性化TTY：按下键盘组合Ctrl（左）+Enter 后会打印“hello”字符串。同时在 其他的TTY中不会有这个效果。  首先添加一个 TYY：修改 const.h 中的 NR\_CONSOLES 数值，然后在 tty.c 中 的in\_process 部分在 Enter 部分添加新的判断，判断是否按下了左 Ctrl 键，同时判断 p\_tty 是否指向第 3 个 TTY。如果同时满足，则答应出一个字符串，否则只进行正常 的换行操作。   1. **case** ENTER: 2. **if**((key & FLAG\_CRTL\_L) && (p\_tty == TTY\_END - 1)){ 3. print\_s(p\_tty,"hello,randi"); 4. } 5. put\_key(p\_tty,‘\n’); 6. **break**;   并编写一个打印字符串的函数print\_s   1. **void** print\_s(TTY\* p\_tty, **char** a[]){ 2. **int** i = 0; 3. **while**(a[i] != ‘\0’){ 4. put\_key(p\_tty,a[i]); 5. i++; 6. } 7. }   运行结果如下：   * + 1. 动手做 2：尝试扩展一下printf，让它支持%s，想想目前的 printf 实现是否有什么安全漏洞？可以怎么解决   参考实验题中的实现printf 的“%x”功能，进入 vsprintf.c 函数，填充 case‘s’部分。  函数vsprintf 虽然它只识别%x 这一种格式，但其他格式的原理也是一样的，即根 据%后的格式字符就能判断下一个参数的类型，从而知道从堆栈中取出什么。  va\_arg 功能：获取当前指针所指的可变参数并将并将指针移向下一可变参数。  自定义insert\_str() 函数：将参数的内容传到p中。   1. #define va\_arg(list ,mode) ((mode∗)(list+=sizeof(mode)))[−1] 2. **static** **char**∗ insert\_str(**char**∗ buf, **const** **char**∗ str) 3. { 4. **char** ∗p=buf; 5. **while** (∗str) { 6. ∗p++=∗str++; 7. } 8. **return** p; 9. } 11. **int** vsprintf(**char** ∗buf, **const** **char** ∗fmt, **va\_list** args) 12. { 13. **char**∗ p; 14. **char**∗ s; 15. **char** tmp[256]; 16. **va\_list** p\_next\_arg=args; 18. **for** (p=buf;∗fmt;fmt++) { 19. **if** (∗fmt !='%') { 20. ∗p++=∗fmt; 21. **continue**; 22. } 24. fmt++; 26. **switch** (∗fmt) { 27. **case** 'x': 28. itoa(tmp, ∗((**int**∗)p\_next\_arg)); 29. strcpy(p, tmp); 30. p\_next\_arg+=4; 31. p+=strlen(tmp); 32. **break**; 33. **case** 's': 34. s=va\_arg(p\_next\_arg, **char**∗); 35. p= insert\_str(p,s); 36. **break**; 37. **default**: 38. **break**; 39. } 40. } 42. **return** (p−buf); 43. }   在main.c中，其使用%s打印出字符串。并使用两个%s，来测试我们的%s功能是 否适用于可变参数。   1. **void** TestA() 2. { 3. **int** i =0; 4. **const** **char** str[] ={"hello"}; 5. **while**(1) 6. { 7. printf("%s,%s",str ,"test"); 8. milli\_delay(200) ; 9. } 10. }   运行如下:  安全漏洞：  打印的时候，printf 依照字符转换说明符规定的格式从低地址开始提取数据。直到 参数打印完。 比方遇到%f说明符就提取8个字节的数据，遇到%d就提取4个字节。printf（）事 实上不知道参数的个数，它仅仅会依据format 中的打印格式的数目依次打印堆栈中參 数format 后面地址的内容。 这样一来，printf() 事实上存在安全隐患：它会强行读取内存的数据当作正常数据 输出，没有边界检测（非常有可能产生堆溢出）。 | | | | | |
| 1. 各人实验贡献与体会（每人各自撰写） | | | | | |
| 实际上每个人都参与讨论，共同完成了实验   1. 1. 王亚鹏：   **键盘输入与显示器输出**  通过实现键盘缓冲区和中断处理程序，理解了键盘输入的工作原理，包括扫描码的解析及其在 I/O 系统中的重要性  显示器输出部分使我熟悉了显存的使用方式，以及通过硬件寄存器（如 CRT Controller Registers）实现光标定位、翻页等功能。  **TTY 子系统**  学习和搭建 TTY 基本架构，掌握了多任务的调度与实现方法（与之前的实验基本相同的调用方法），尤其是 TTY 与控制台的映射关系，使得多控制台并发操作变得清晰。利用多TTY并发处理输入输出的机制设计了每个TTY任务与对应控制台的绑定关系。通过设计task\_tty()主循环，拆分读写操作，还为TTY添加了特定功能，按下Ctrl+Enter触发显示“hello,randi”字符串，并且显示有限制条件  自定义 TTY 的实现使我进一步体会到操作系统设计的灵活性和扩展性  **系统调用与 printf 实现**  通过扩展 printf 支持 %s 格式符，对可变参数的处理方式（如 va\_arg 的使用）有了更深的理解，并意识到 printf 的实现细节中可能存在的安全隐患（如内存溢出），因此在使用printf时尽量使用格式来限定字符串  系统调用的实现让我对用户态与内核态的交互有了直观认识，尤其是为进程绑定 TTY 的设计，使系统输出变得更有条理   * 1. 杨依磊：   键盘输入的处理:通过实验，我了解了Scancode 的分类（MakeCode、BreakCode）及它们的不同作用，我分析了实验中 disable\_int 和 enable\_int 的使用，以及中断处理在保证数据一致性中的作用。显示器输出的实现:我利用重新设置显示开始地址的原理，进行了显存的使用和光标位置的控制。理解了文本模式字符输出的底层原理，尤其是操作系统如何与硬件协调完成显示任务。TTY 终端的并发机制:我通过多个 TTY 并发处理输入输出的流程，学习了任务调度和资源隔离机制。并与杜洪波讨论并共同动手创建个性化 TTY，添加Ctrl + Enter 触发显示“hello,randi”字符串。printf 的扩展:我通过讨论，理解了 printf 的底层实现，尝试扩展其功能以支持 %s 格式符，并发现了当前实现中潜在的缓冲区溢出问题。提出了通过边界检查并动态分配缓冲区来提升安全性的改进方案。  3.3杜泓波：  在实验过程中有如下心得：验证并解释键盘输入的处理程序在实验中，我实现了键盘缓冲区和中断处理程序，理解了键盘输入通过硬件中断传递至系统并显示在屏幕上的完整流程。显示器输出与滚屏功能我通过操作CRT控制器寄存器，实现了光标移动和屏幕滚动的功能。在滚屏功能中，通过修改显示开始地址以动态调整屏幕显示区域，实现了显存的分段管理和直接访问。搭建TTY基本架构我利用多TTY并发处理输入输出的机制设计了每个TTY任务与对应控制台的绑定关系。通过设计task\_tty()主循环，拆分读写操作，使代码易于理解。我还为TTY添加了特定功能，例如按下Ctrl+Enter触发显示“hello,randi”字符串，并且仅限于特定TTY中显示。设置不同程序处于不同特权级我学会了通过配置GDT和LDT实现任务与用户进程的特权级区分，Ring0负责调度任务，Ring3运行用户进程，从而实现了系统的基本安全策略。 实现printf函数我从零实现了支持%x的printf()函数，扩展功能支持了%s格式，并增加了边界检测和变量参数解析机制以保证函数的安全性。   * 1. 侯名扬：   这次实验我有很大的收获，总的来说，本次的实验比较复杂，有许多的重难点和需 要理解的新知识。在本章需要学习的知识中，也有许多值得慢慢分析和理解的地方。  通过本次实验，我深入分析、理解并且掌握了以下内容：键盘输入的基本方法、显 示器输出的基本方法、TTY(TeleTYpe) 任务子系统的基本功能和方法、如何配置不同 程序处于不同特权级以及printf函数的实现...... 对其中各种相关知识都有了一定程度的 理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代 码编写的知识和能力带来了极大的锻炼和提升。  本次实验中最大的困难点在于动手改部分：添加一个个性化的TTY，在该TTY上， 基于键盘输入或者光标移动规律，运行彩蛋程序以及扩展printf实现%s。虽然需要实现 的功能比较简单，可是我对各种繁杂的代码难以快速掌握和吸收、对本章知识不是完全 理解等原因，导致编写和调试还是花了很长时间，不过最终还是得到了正确的结果，感 觉自己收获颇丰，极大加深了我对I/O子系统的理解！ | | | | | |
| 1. 教师评语   （实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评）   |  |  | | --- | --- | | 85-100 | * 实验内容完整或者有超出课程实验大纲的内容； * 实验步骤详尽，能够体现完整的实验过程； * 实验结果正确且实验数据分析得当； * 实验心得与思考全面并且有自己的独立思考； * 实验报告文档规范、排版整齐。 | | 75-84 | * 实验内容较为完整； * 实验步骤较为详尽，能够体现实验过程； * 实验结果正确且实验数据分析较为得当； * 实验心得与思考全面； * 实验报告文档规范、排版较为整齐。 | | 60-74 | * 实验内容有缺失； * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | 60以下 | * 实验内容严重缺失、实验态度不够端正 * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | | | | | |
|  | | | | | |
| **教师评分（请填写好姓名、学号）** | | | | | |
| 姓名 | | 学号 | | 分数 | |
| 王亚鹏 | | 2022302181161 | |  | |
| 杨依磊 | | 2022302181159 | |  | |
| 杜泓波 | | 2022302181162 | |  | |
| 侯名扬 | | 2022302181165 | |  | |
| 教师签名：  年 月 日 | | | | | |