

操作系统原理实验报告

实验项目四：中断机制的建立，用户程序与内核的解耦，以及用户程序的时间片轮转

院(系)名称： 数据科学与计算机学院

专业名称： 计算机科学与技术

学生姓名： 王永锋

学生学号： 16337237

指导教师： 凌应标

二〇一八年四月十七日

目 录

1	实验目的及要求	1
1.1	实验目的	1
1.2	实验要求	1
2	实验方案	2
2.1	实验工具和环境	2
2.2	程序功能说明及大致思路阐述	2
2.2.1	程序大致思路阐述	2
2.2.2	实验四主要工作说明	3
2.2.3	实验结果不足，未来展望	3
2.3	代码框架的设计	4
2.3.1	用户静态库的设计	4
2.3.1.1	部分函数的函数调用图	5
2.3.1.2	静态链接库	6
2.3.2	进程控制块的设计	6
2.3.3	自定义时钟中断，实现切换进程	7
3	实验难点及亮点	10
3.1	用户程序与内核解耦合	10
3.2	进程切换	10
4	曾经遇到的问题	11
4.1	内存改写的 bug	11
4.2	gcc 中结构体的自动对齐	13
4.3	一条机器相关的指令	14
5	实验结果	16
5.1	进程切换的测试结果	16
6	实验总结	17

附录 A 文件的组织	19
------------------	----

1 实验目的及要求

1.1 实验目的

1. 操作系统中断机制的建立，并学会如何使用中断实现系统调用。
2. 使用 C 程序编写用户程序，并通过编写 C 运行时库，让用户程序与系统内核解耦合
3. 尝试通过修改时钟中断，实现用户进程的时间片轮转运行。

1.2 实验要求

本次实验，要求需要完成以下目标：

- 实验四必须在实验三基础上进行，保留或扩展原有功能，实现部分新增功能。
- 内核中，对 33 号、34 号、35 号和 36 号中断编写中断服务程序，分别在屏幕 1/4 区域内显示一些个性化信息。
- 再编写一个用户程序，利用 int 33、int 34、int 35 和 int 36 产生中断调用你这 4 个服务程序。
- 扩充系统调用，实现三项以上新的功能，并编写一个测试所有系统调用功能的用户程序。
- 编写键盘中断响应程序，原有的你设计的用户程序运行时，键盘事件会做出有事反应：当键盘有按键时，屏幕适当位置显示” OUCH! OUCH!”。
- 制作包含引导程序，监控程序和若干可加载并执行的用户程序组成的 1.44M 软盘映像。
- 在指定时间内，提交所有相关源程序文件和软盘映像文件，操作使用说明和实验报告。
- 实验报告格式不变，实验方案、实验过程或心得体会中主要描述个人工作，必须有展示技术性的过程细节截图和说明。

2 实验方案

2.1 实验工具和环境

本次实验环境与之前变化不大，唯一的改变是因为需要生成静态链接库，使用了 linux 系统中提供的 ar 工具。以下是本次实验的工具链。表 2.1

表 2.1 本实验所使用的工具链

软件名称	用途
bash	一个命令行终端，可提供 linux 的一些命令与执行 shell 脚本
nasm	将 x86 汇编文件编译成 .bin 二进制文件
gcc	编译工具，将 c 编译成二进制文件
ar	使用多个 .o 文件，生成 .a 静态链接库
ld	gcc 套件中包含的连接器，用于将多个可执行文件连接起来
make	gcc 套件中的工具，用于执行 makefile 文件
dd	将二进制文件的内容写进软盘镜像中
objdump	对可执行文件或二进制文件进行反编译
hexdump	以十六进制形式查看软盘镜像文件
bochs	虚拟机，用于加载装有自定义引导程序的软盘，使用软件模拟，速度不稳定
bochsdbg	调试工具，用于给装有自定义引导程序的软盘文件进行调试
qemu	虚拟机，用于加载装有自定义程序的软盘，使用硬件模拟，速度稳定且较快

2.2 程序功能说明及大致思路阐述

这里会对程序的大致运行流程进行一个粗略的描述，同时罗列了当前系统内核支持的功能。

2.2.1 程序大致思路阐述

1. 一开机，处于软盘第一个扇区的引导程序加载位于第 40-45 个扇区的加载器到内存 0x8000 处，并跳转到加载器中。

2. 加载器（loader）使用文件系统的接口，加载位于第 40-86 个扇区的系统内核到内存 0x10000 处，并将控制权转交给系统内核。
3. 系统内核刚开始运行，先进行初始化工作，（如初始化自定义中断，初始化文件系统数据，初始化进程控制块表）
4. 调用文件系统接口，将指定文件加载到指定内存地址处
5. 安装 8 号时钟中断，并且调用 `restart` 函数开始第一个进程。

2.2.2 实验四主要工作说明

在上一个实验中，我已经实现了终端，自定义中断以及时钟中断的修改，同时还实现了文件系统。在这一次实验中，我主要做了以下工作：

- 因为操作系统内核越来越大，18 个扇区远远不够用，因此在引导扇区与内核之间增加了加载器，加载器能够使用文件系统的接口，读取内核所在扇区，如此便可摆脱 18 个扇区的限制。
- 将用户 C 库中最底层的四个函数包装成软中断，从而实现与操作系统解耦合。
- 在保证用户 C 库不依赖操作系统的库文件后，整理 C 库文件，使用 `ar` 生成静态链接库，从而能够直接通过链接 C 库的方式直接生成用户程序，而不需要让用户程序与内核一起编译。
- 保证用户程序能够正常编译运行后，考虑到要让用户程序与内核的段分开（原来是在一个段上的），因此考虑使用进程切换的方式，实现切换段运行用户程序。
- 设计编写了进程控制块，并使用进程控制块与自定义时钟中断，实现多个用户程序的并发执行。

2.2.3 实验结果不足，未来展望

本次实验还有很多想完成的，碍于时间问题，先完成这一部分，下面是我觉得这一次做的不够好的地方，这也是之后我工作的方向。

- 原先写的 `tty.c` 命令行终端，是依附于内核的，它必须使用内核才能够访问的全局变量，这导致了我无法将用户程序的并发执行与终端的运行放到一起。因此在这一次实验结果的镜像中，并没有看到上一次实验的终端。
- C 库与内核的解耦合。操作系统的很多服务，我都需要使用中断的方式，包装成系统调用，从而能够让 C 库中通过调用中断，避免了与内核数据的直接交互，以此，实现与操作系统的解耦合。本次实验我将一部分函数从内核中使用中断抽象出接口，从而实现了解耦合，但是还有一些操作系统服务没能包含在用户 C 库中（如文件系统相关操作），这也是我没能将终端从内核中分离出

来的主要原因。下一步，就是要继续完善系统调用，构建功能更加丰富的用户 C 库。

- 在创建进程的时候我是采用手动分配内存的方式，这样并不优美的做法，让我萌生了编写内存分配的想法，有了内存分配就可以进一步编写创建进程映像的系统调用，这样子我的操作系统就更加完满了。

2.3 代码框架的设计

2.3.1 用户静态库的设计

为了能够在 C 中更有效的操作硬件，同时保证更有效率的开发，我实现了以下库函数作为内核需要调用的头文件。对于一些无法使用 C 语言实现的功能，如端口的读写，我使用汇编语言实现了一些函数，将端口读写封装成一个个 C 语言直接可用的函数，而对 bios 中断的使用也是类似，在 `basic.asm` 中实现了对一些 bios 中断的直接调用，那么在 C 中，就可以对这些硬件操作进行进一步的封装（主要在 `stdio.c` 中实现），实现一些对开发者友好的接口，从而加快开发操作系统的效率。

以下是我用户 C 库的文件组织。

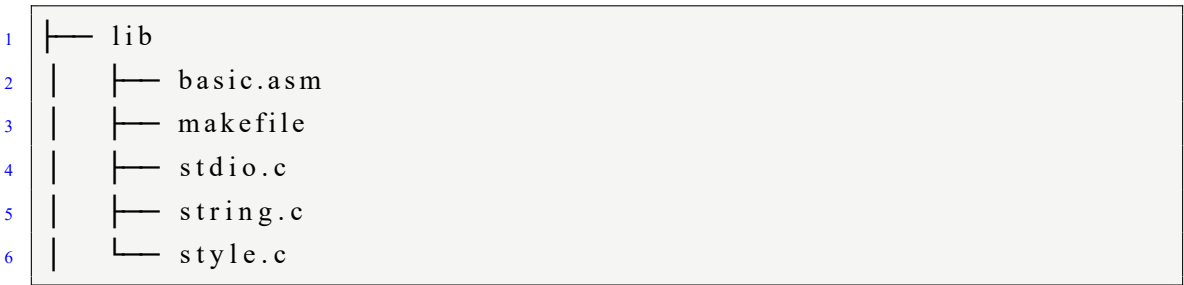


表 2.2 库文件说明

头文件名	功能说明
<code>basic.asm</code>	实现一些只能通过汇编实现的接口函数
<code>stdio.c</code>	存放处理 I/O 的函数（如 <code>printf</code> ）
<code>string.c</code>	用于字符串的处理，包括比较 <code>strcmp</code> 和复制 <code>strcpy</code> 等函数
<code>style.c</code>	编写了一个用于移动一行显示内容的函数，以后可以考虑做图形接口相关函数

2.3.1.1 部分函数的函数调用图

库中对用户的每一个接口,其底层都是依赖于系统内核已经编写好的 33,34,35,36 号中断及一些 bios 中断提供的功能。下面举两例: `putc` 函数与 `printf` 函数。

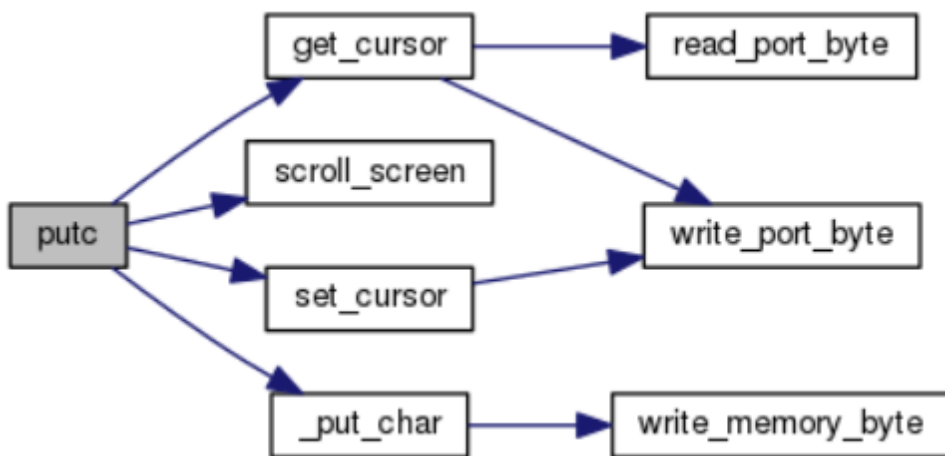


图 2.1 `putc` 函数的函数调用图

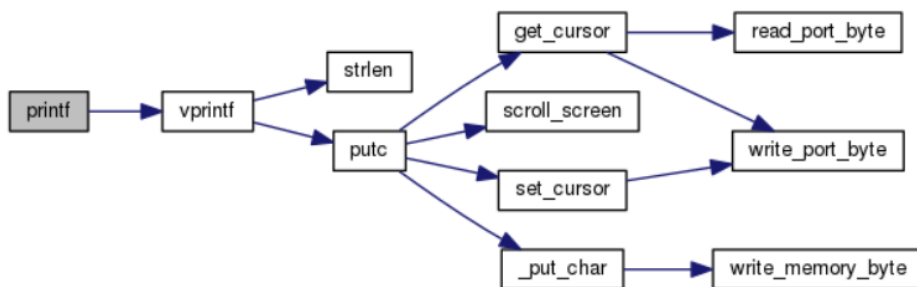


图 2.2 `printf` 函数的函数调用图

对于 `putc` 函数的执行,以下放出简短的源代码方便理解该函数如何与内核解耦合。本函数没有直接调用 `biox` 的中断,而是通过端口交互的方式,修改光标,然后调用底层的 `_put_char()` 函数来将指定的字母写到显存中,这样做是为了以后可以不依赖 `bios` 中断,从而有可能能够尝试保护模式。

最底层的三个函数, `read_port_byte()`, `write_port_byte`, `write_memory_byte`, 使用了中断的方式,调用了操作系统提供的功能,来对端口进行读写,以及进行写内存。这里要用中断实现写内存的考虑主要是想在内核中判断该地址是否合法,可能用户想写一些内核的关键区域,这时候是应该要阻止的。

```
1 void putc(char c){
2     u16 cursor_index = get_cursor();
3     u16 row = cursor_index / 80;
```



```

4      ul6 col = cursor_index % 80;
5      if (cursor_index >= 1920){
6          scroll_screen();
7          cursor_index = 1840;
8      }
9      switch (c) {
10         case '\n':
11             set_cursor((row+1)*80); // 回车，移到下一行
12             break;
13         case '\r':
14             set_cursor(row*80);      // 移到本行开头处
15             break;
16         default:
17             _put_char(c, cursor_index);
18             set_cursor(cursor_index+1);
19             break;
20     }
21     return ;
22 }

```

2.3.1.2 静态链接库

在将 C 库与操作系统解耦合之后，我们可以讲 C 库连接成一个静态链接库，这样子多个用户程序都可以链接同一个 C 运行时库，不用重新编译。我是用的是下面的指令：

```

1      ar rcs ../include/c_run_time.a stdio.o  basic.o  string.o
        style.o

```

此后需要编译用户程序的时候，只需要这样写即可：

```

1      ld $(LINK_FLAGS) -o test_a.bin test_a.o ../include/c_run_time.a
2      // LINK_FLAGS = -Ttext 0x00000 -m elf_i386 -T t.lds --oformat
        binary

```

2.3.2 进程控制块的设计

对于进程控制块，我使用了如下代码所示的结构体来表示 (代码见 include/type.h):

```

1  typedef struct proc_register {
2      u16 sp; // 用户栈指针
3      u16 ss; // 用户所处栈段
4      u16 es;
5      u16 ds;
6      // popad
7      u32 edi;
8      u32 esi;
9      u32 ebp;
10     u32 placeholder;
11     u32 ebx;
12     u32 edx;
13     u32 ecx;
14     u32 eax;
15     // pushad
16     u16 ip;
17     u16 cs;
18     u16 flags;
19 } procRegister;
20
21 typedef struct controlProcessBlock {
22     procRegister regs;
23     u32 pid;
24     char p_name[16];
25 } PCB_t;

```

其中第一个结构体：proc_register 存放的是用户程序用到的各种寄存器，第二个结构体，在包含了第一个结构体的基础上，增加了一些进程的信息。

2.3.3 自定义时钟中断，实现切换进程

为了实现进程切换，我对时钟中断进行了修改，为了方便调试，进程切换的程序安装在了 0x8 号和 0x41 号两个中断处，以便我可以在程序中显式调用“int 0x41”进行该程序的调试。

时钟中断的代码分为两部分，第一部分是将用户程序的运行信息保存到用户栈中，然后将信息复制到进程表中。保存信息后，将栈转为内核栈，并且调用内核的调度程序更改当前进程指针。该部分代码如下所示。

```

1  new_int41h:
2      ; 保存所有信息到用户栈中

```

```

3      ; cli
4      pushad
5      push ds
6      push es
7      push ss
8      ;push sp 这样的写法是不好的，这个是先push再减2
9      ;push sp ; 这个sp的值是不确定的
10     sub sp, 2
11     mov bp, sp
12     mov [ss:bp], sp
13
14     ; ds 已经是用户段了，并且和ss相同
15     mov si, sp
16     mov ax, 0x1000 ; 内核段
17     mov es, ax ;
18     mov di, [es:cur_process]
19     mov cx, 46 ;
20     cld
21     rep movsb
22
23     mov ax, 0x1000
24     mov es, ax
25     mov ds, ax
26     mov ss, ax
27     mov sp, 0x5000
28     ; 进入到内核段中
29     call dword schedule_process
30     jmp int4lh_restart ;

```

第二部分，是根据当前进程指针指向的进程表项，启动当前进程。启动的过程是这样的，先使用进程表的信息，得到用户栈的地址，然后将进程表中进程运行所需要的信息复制到用户栈中，最后将栈指针和栈段移到用户栈，并通过 `pop` 指令，恢复现场，`iret` 回到新的用户程序。

```

1      ; 启动一个进程，根据当前进程来启动
2      int4lh_restart:
3          mov bp, [cur_process]
4          mov si, bp
5
6          ; 取得进程表部分需要复制的内容
7          mov es, [ds:bp+2]; 取得当前进程的栈段

```

```

8      mov di, [ds:bp]; 取得当前进程的栈指针
9      mov cx, 46
10     cld
11     rep movsb
12
13     mov al, 20h                ; AL = EOI
14     out 20h, al                ; 发送EOI到主8529A
15     out 0A0h, al              ; 发送EOI到从8529A, 注释掉好像也行, 为啥?
16
17     sub di, 46; 取得第一个进程的栈指针, 移动之后di的值已经变了, 所以要减回来
18     mov ax, es; 取得第一个进程的栈段
19     mov ss, ax
20     mov sp, di
21     pop cx
22     pop cx ; 这两个是push到cx, 清掉栈中原有的东西
23     pop es
24     pop ds
25     popad
26 int4lh_reture:
27     iret

```

3 实验难点及亮点

本次实验主要有以下的难点。

- 用户程序与内核的解耦
- 进程切换

3.1 用户程序与内核解耦合

以前一直很想让用户程序与内核运行在不用的段中，让用户程序不需要用类似于“org 0x20000”的方式显式指定运行所在地址。相反，用户程序不需要知道自己运行所在地址，内存的分配工作应该交给内核来完成。

为了实现这一点，在前面实验方案中提到几点

- 用户静态 C 运行时库与内核的解耦合，与静态链接库的制作
- resetart 函数的编写，使内核能够从表中读取用户程序段地址和偏移地址，并跳转过去运行用户程序

主要通过完成这两点，实现了用户程序与内核的解耦合。

3.2 进程切换

当用户程序运行在与内核不同的段中的时候，此时用户程序是无法直接访问内核的数据的，对用户而言，需要通过中断，回到内核态然后再在内核中完成内核的任务。在实现这个切换的过程时，想到实现这个就快实现进程切换了，就尽力把进程切换也实现了。关于该项的实现可见前面实验方案中的进程切换的实现。

4 曾经遇到的问题

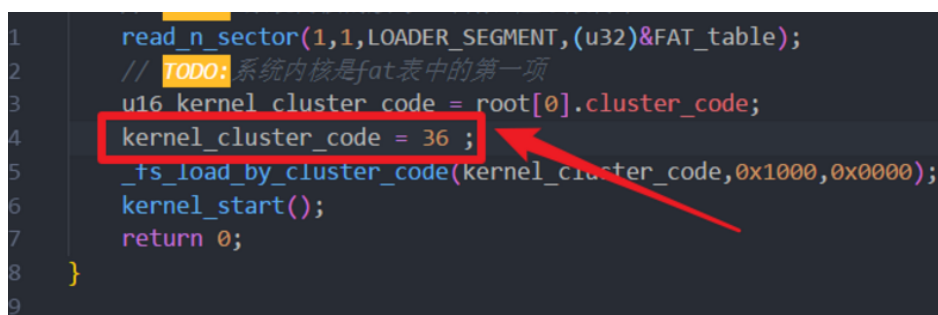
4.1 内存改写的 bug

这个 bug 在上一个实验写文件系统的时候就存在了，那时候通过增大 fat 表和 root 表的大小就可以解决，但是这一此解决这个 bug 的迫切性大大提高。

我遇到了这样的问题，下面的代码无法取得对应的值：

```
1 u16 kernel_cluster_code = root[0].cluster_code;
```

但是当我换一种写法的时候，直接给 kernel_cluster_code 赋值就没问题？图 4.1



```
1 read_n_sector(1,1,LOADER_SEGMENT,(u32)&FAT_table);
2 // TODO: 系统内核是fat表中的第一项
3 u16 kernel_cluster_code = root[0].cluster_code;
4 kernel_cluster_code = 36;
5 _ts_load_by_cluster_code(kernel_cluster_code,0x1000,0x0000);
6 kernel_start();
7 return 0;
8 }
9
```

图 4.1 代码示意图

通过查看汇编，我找到了 root 表的地址，在 bochs 调试的时候，发现我的 root 表的内容并不对！0x8560 是我的根目录区的内容，但是直到 0x8620，才有完整的内容。如图图 4.2

以此，提醒了我，是否是 root 表的内容被某个操作修改的内存覆盖了呢？

```
1 // 读取root文件表项
2 read_n_sector(root_sector,1, LOADER_SEGMENT,(u32)&root);
3 // 读取 fat 表
4 read_n_sector(1,1,LOADER_SEGMENT,(u32)&FAT_table);
```

事实上，的确是这样子的，我的 FAT_table 的地址是 0x8420，在读取了一个扇区的内容后，由于一个扇区的大小为 0x200 字节，该操作会写的内存范围为 0x8420-0x8620，这就修改了 0x8560 所在的 root 表。

这个问题的根本来源，是因为我创建 FAT_table 数组时，没有创建一个扇区大小的数组。

```

<bochs:10> xp /1000bc 0x8500
[bochs]:
0x0000000000008500 <bogus+ 0>: ? ? \0 \0 \0 \0 \0 \0
0x0000000000008508 <bogus+ 8>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008510 <bogus+ 16>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008518 <bogus+ 24>: \0 \0 \0 \0 \x7F \0 \x80 \0
0x0000000000008520 <bogus+ 32>: \x81 \0 ? \0 ? ? \0 \0
0x0000000000008528 <bogus+ 40>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008530 <bogus+ 48>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008538 <bogus+ 56>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008540 <bogus+ 64>: ' \0 , \0 " \0 " \0 \0
0x0000000000008548 <bogus+ 72>: ? ? \0 \0 \0 \0 \0 \0
0x0000000000008550 <bogus+ 80>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008558 <bogus+ 88>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008560 <bogus+ 96>: \0 \0 \0 \0 £ \0 □ \0 \0
0x0000000000008568 <bogus+ 104>: ¥ \0 | \0 ? ? \0 \0 \0
0x0000000000008570 <bogus+ 112>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008578 <bogus+ 120>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008580 <bogus+ 128>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008588 <bogus+ 136>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008590 <bogus+ 144>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008598 <bogus+ 152>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085a0 <bogus+ 160>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085a8 <bogus+ 168>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085b0 <bogus+ 176>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085b8 <bogus+ 184>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085c0 <bogus+ 192>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085c8 <bogus+ 200>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085d0 <bogus+ 208>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085d8 <bogus+ 216>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085e0 <bogus+ 224>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085e8 <bogus+ 232>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085f0 <bogus+ 240>: \0 \0 \0 \0 \0 \0 \0 \0
0x00000000000085f8 <bogus+ 248>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008600 <bogus+ 256>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008608 <bogus+ 264>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008610 <bogus+ 272>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008618 <bogus+ 280>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008620 <bogus+ 288>: s t o n e 4 . b
0x0000000000008628 <bogus+ 296>: i n \0 \0 \0 \0 \0 \0
0x0000000000008630 <bogus+ 304>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008638 <bogus+ 312>: \0 \0 \x90 \0 \0 \0 \0 \0
0x0000000000008640 <bogus+ 320>: o u c h . b i n
0x0000000000008648 <bogus+ 328>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008650 <bogus+ 336>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008658 <bogus+ 344>: ¢ \0 \0 \0 \0 \0 \0 \0
0x0000000000008660 <bogus+ 352>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008668 <bogus+ 360>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008670 <bogus+ 368>: \0 \0 \0 \0 \0 \0 \0 \0
0x0000000000008678 <bogus+ 376>: \0 \0 \0 \0 \0 \0 \0 \0

```

图 4.2 root 表的内容

```

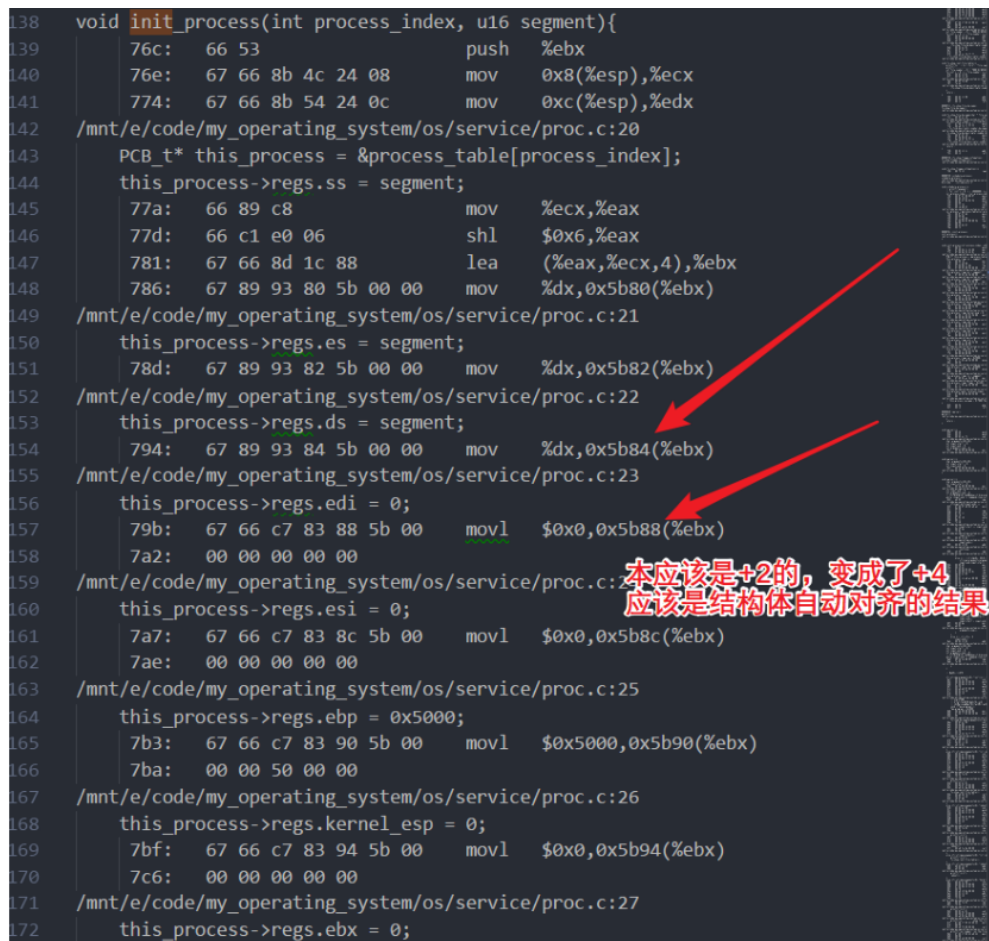
1 u16 FAT_table[256]; // 0x420
2 FileDescriptor root[32]; // 0x620

```

此后，将 FAT_table 和 root 表的大小扩大到一个扇区，问题解决。

4.2 gcc 中结构体的自动对齐

在编写进程控制块的时候，我需要在汇编中访问进程控制块结构体指定偏移量的变量，但是一直都无法读取到。想到可能是偏移量错了，我就想看看 init_process 中是如何访问结构体中的元素的，一看，发现的确是偏移量不对的问题，但是造成这个问题的原因，是 gcc 编译器对结构体做的自动对齐。一处原本应该 +2 就可以访问的变量，变成了 +4。



```

138 void init_process(int process_index, u16 segment){
139     76c: 66 53          push    %ebx
140     76e: 67 66 8b 4c 24 08    mov     0x8(%esp),%ecx
141     774: 67 66 8b 54 24 0c    mov     0xc(%esp),%edx
142 /mnt/e/code/my_operating_system/os/service/proc.c:20
143     PCB_t* this_process = &process_table[process_index];
144     this_process->regs.ss = segment;
145     77a: 66 89 c8        mov     %ecx,%eax
146     77d: 66 c1 e0 06      shl     $0x6,%eax
147     781: 67 66 8d 1c 88    lea     (%eax,%ecx,4),%ebx
148     786: 67 89 93 80 5b 00 00    mov     %dx,0x5b80(%ebx)
149 /mnt/e/code/my_operating_system/os/service/proc.c:21
150     this_process->regs.es = segment;
151     78d: 67 89 93 82 5b 00 00    mov     %dx,0x5b82(%ebx)
152 /mnt/e/code/my_operating_system/os/service/proc.c:22
153     this_process->regs.ds = segment;
154     794: 67 89 93 84 5b 00 00    mov     %dx,0x5b84(%ebx)
155 /mnt/e/code/my_operating_system/os/service/proc.c:23
156     this_process->regs.edi = 0;
157     79b: 67 66 c7 83 88 5b 00    movl    $0x0,0x5b88(%ebx)
158     7a2: 00 00 00 00 00 00
159 /mnt/e/code/my_operating_system/os/service/proc.c:24
160     this_process->regs.esi = 0;
161     7a7: 67 66 c7 83 8c 5b 00    movl    $0x0,0x5b8c(%ebx)
162     7ae: 00 00 00 00 00 00
163 /mnt/e/code/my_operating_system/os/service/proc.c:25
164     this_process->regs.ebp = 0x5000;
165     7b3: 67 66 c7 83 90 5b 00    movl    $0x5000,0x5b90(%ebx)
166     7ba: 00 00 50 00 00 00
167 /mnt/e/code/my_operating_system/os/service/proc.c:26
168     this_process->regs.kernel_esp = 0;
169     7bf: 67 66 c7 83 94 5b 00    movl    $0x0,0x5b94(%ebx)
170     7c6: 00 00 00 00 00 00
171 /mnt/e/code/my_operating_system/os/service/proc.c:27
172     this_process->regs.ebx = 0;

```

本应该是+2的，变成了+4
应该是结构体自动对齐的结果

图 4.3 汇编中查看访问结构图内容的代码

4.3 一条机器相关的指令

在编写进程切换的时钟中断时，遇到了不少的问题，此处摘录一个比较大的问题：运行一段时间之后，本来两个进程并发执行得还挺好，但是运行了一段时间后，会卡住，如图 4.4 所示。

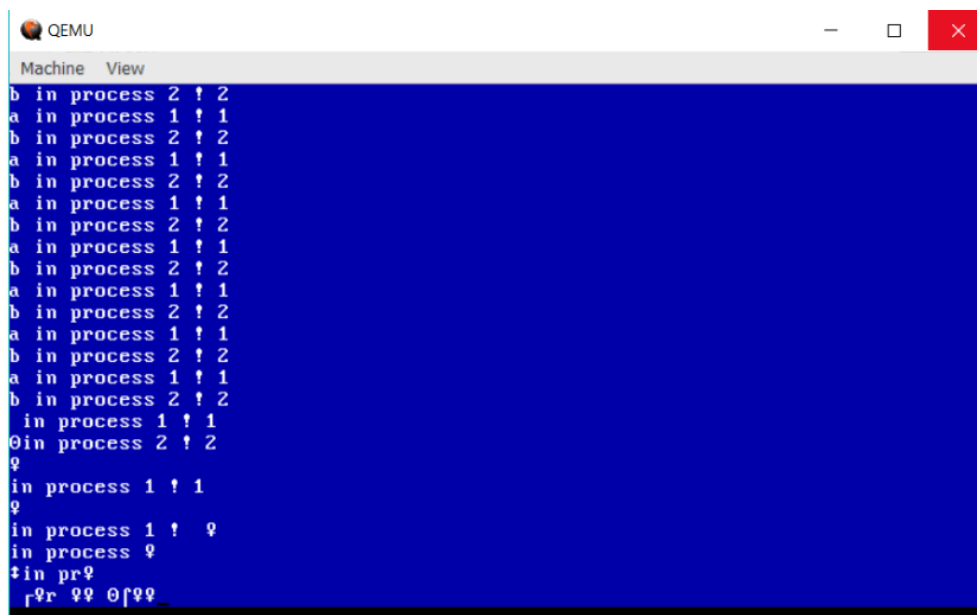


图 4.4 进程切换运行结果

在调试的时候，发现我的进程 1 每运行一次，栈都会向后退两个字节！如??所示。

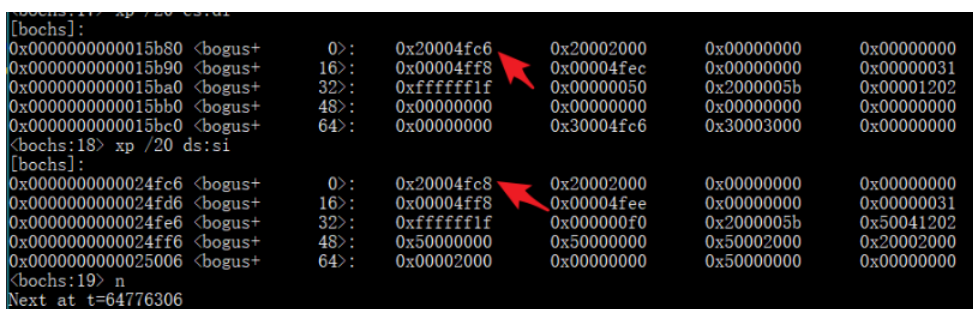


图 4.5 栈指针的神秘减 2

中间经过很长时间的调试，发现了一个与我预期不一致的行为。关键在 `push sp` 这一条指令! 在以前计组的学习中，我了解到一般的是 `sp` 先减 2，在将减 2 后的值 `push` 进栈中，因此这一次，我也是按照这样的想法，写了修改栈的代码，但是在这一次调试的过程中，我发现，在虚拟机中跑的时候，却是先将当前栈指针 `push` 进栈中，然后再对栈指针减 2。这样代码行为的差异带来的，便是用户栈操作

不正确带来的内存泄漏，当运行一段时间，内存泄漏完了之后，便会导致用户程序崩溃。如图 4.6所示。

```
| STACK 0x24fea [0xffff]
<bochs:32> n
Next at t=64799449
(0) [0x0000000101ce] 1000:01ce (unk. ctxt): push sp
<bochs:33> print-stack
Stack address size 2
| STACK 0x24fca [0x2000]
| STACK 0x24fcc [0x2000]
| STACK 0x24fce [0x2000]
| STACK 0x24fd0 [0x0000]
| STACK 0x24fd2 [0x0000]
| STACK 0x24fd4 [0x0000]
| STACK 0x24fd6 [0x0000]
| STACK 0x24fd8 [0x4ff8]
| STACK 0x24fda [0x0000]
| STACK 0x24fdc [0x4ff0]
| STACK 0x24fde [0x0000]
| STACK 0x24fe0 [0x0000]
| STACK 0x24fe2 [0x0000]
| STACK 0x24fe4 [0x0031]
| STACK 0x24fe6 [0x0000]
| STACK 0x24fe8 [0xff1f]
<bochs:34> n
Next at t=64799450
(0) [0x0000000101cf] 1000:01cf (unk. ctxt): mov si, sp
<bochs:35> print-stack
Stack address size 2
| STACK 0x24fc8 [0x4fca]
| STACK 0x24fca [0x2000]
| STACK 0x24fcc [0x2000]
| STACK 0x24fce [0x2000]
| STACK 0x24fd0 [0x0000]
| STACK 0x24fd2 [0x0000]
| STACK 0x24fd4 [0x0000]
| STACK 0x24fd6 [0x0000]
| STACK 0x24fd8 [0x4ff8]
| STACK 0x24fda [0x0000]
| STACK 0x24fdc [0x4ff0]
| STACK 0x24fde [0x0000]
| STACK 0x24fe0 [0x0000]
| STACK 0x24fe2 [0x0000]
| STACK 0x24fe4 [0x0031]
| STACK 0x24fe6 [0x0000]
| STACK 0x24fe8 [0xff1f]
```

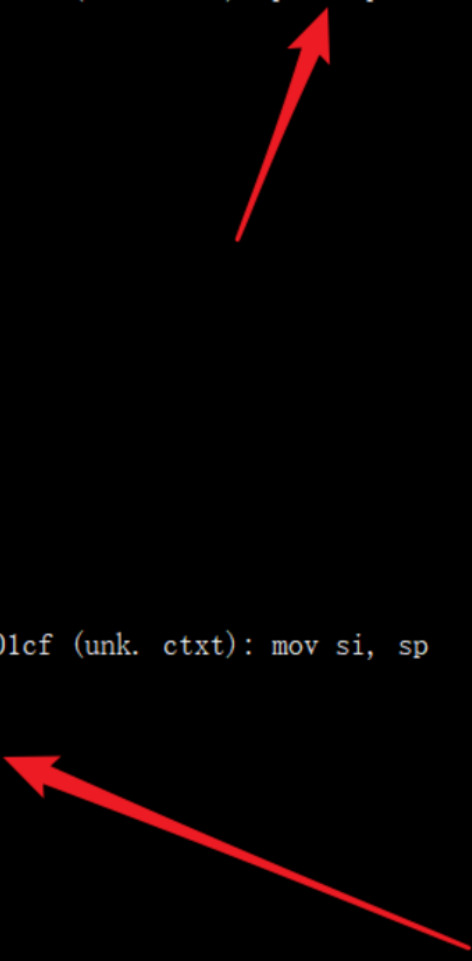


图 4.6 push sp 的行为与课本不一致。

根本问题在于我使用了一条机器相关的指令，后来我把 push sp 改成了没有歧义的指令，代码终于运行正常！

```
1 ;push sp 这样的写法是不好的，这个是先push再减2
2 ;push sp ; 这个sp的值是不确定的
3 sub sp, 2
4 mov bp, sp
5 mov [ss:bp], sp
```

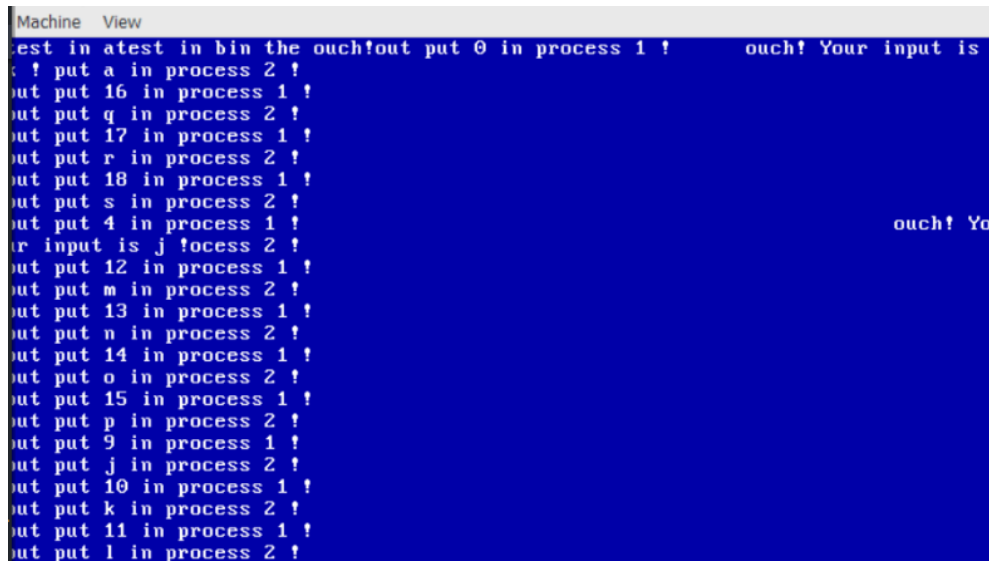
5 实验结果

测试方法：运行镜像，会有三个程序并发的执行。如图 5.1所示。

- 第一个程序会顺序的输出 1,2,3,4 等数字，输出到 1000 后会回到 0 继续输出。可见 os/user/test_a.c
- 第二个程序会顺序输出 a,b,c 等字符，输出到 z 后会回到 a 继续输出。可见 os/user/test_b.c
- 第三个程序能够响应用户的键盘操作，一旦有键盘操作，在屏幕随机位置输出一个”ouch!”还有用户对应输出的字母。可见 os/user/ouch.c

在实验文件夹中，提供了两个镜像，一个是增加了延时的镜像，方便查看效果，另一个是没有延时模块的镜像，方便暴力测试，寻找问题。

5.1 进程切换的测试结果



```
Machine View
test in atest in bin the ouch!out put 0 in process 1 !      ouch! Your input is
! put a in process 2 !
out put 16 in process 1 !
out put q in process 2 !
out put 17 in process 1 !
out put r in process 2 !
out put 18 in process 1 !
out put s in process 2 !
out put 4 in process 1 !      ouch! Yo
r input is j !ocess 2 !
out put 12 in process 1 !
out put m in process 2 !
out put 13 in process 1 !
out put n in process 2 !
out put 14 in process 1 !
out put o in process 2 !
out put 15 in process 1 !
out put p in process 2 !
out put 9 in process 1 !
out put j in process 2 !
out put 10 in process 1 !
out put k in process 2 !
out put 11 in process 1 !
out put l in process 2 !
```

图 5.1 进程切换测试，键盘输出测试

两个进程能够交替的输出递增的数字和字符，并且还能够响应我的键盘输入，在屏幕随机位置输出”ouch! Your input is X”

6 实验总结

这是实验四的实验总结。

这一次实验四做的比较慢，主要是因为其实之前已经算是把 ppt 上的要求做好了，然后一开始也不知道自己要朝着怎样的目标去做，这时候又有其他课程的项目需要做，于是就把很多时间放在了其他课程的项目。

大概还是维持着初衷吧，想着要尽量将操作系统的实现超前一个星期，所以在这一次实验中，除去为整理文件结构，分离用户程序与系统内核所做的努力，我的时间主要花在了实现进程切换的功能上，可以说是花了整整两天时间从早到晚不停地打码调试进程切换。对编写时钟中断后的调试，甚至可以说花了我一天时间去观察时钟中断对栈的操作是否正确。有一段时间，三个用户程序切换个几百次后就会崩溃，就会跳转到一个非法地址，面对这种概率性出现的问题，我一开始几乎没有 debug 的思路，超绝望，怕自己完不成这一次实验。

后来吧，平静下来了，想看看栈的情况如何，经过细细的调试，看到我的用户程序每回来一次，用户栈就会往后退两个字节，看到这个现象的我，就像是看到了一点点黑暗中的曙光啊。至少到这时候是可以确定是我对栈的操作出了一点问题，后来重写了几遍时钟中断，调试了很多遍，终于发现，是自己用了一条“push sp”这样一条机器相关的指令，我对这条指令的理解与机器不一致，导致出现问题。

怎么说呢，在调试的过程，极其需要耐心，因为这需要我仔细的查看汇编，找到 C 代码对应的汇编代码的位置，然后在 bochs 上一步一步的走下去，查看寄存器的值，这一次的 bug，也是我一步一步的看栈的值，才看到这样一个与自己心理预期不一致的情况，这才终于发现关键的问题所在，最终才解决这个问题。

说起来，操作系统编程中，其实有一个需求特别需要满足：能够使用 gdb 调试！因为我们自己编写的 C 代码，现阶段我们甚至没有办法通过 C 指令的单步调试来缩小问题范围，而只能通过更加底层的汇编来进行排查问题。而据我了解，qemu 虚拟机是可以和 gdb 调试器联动，实现 C 源码级别的调试功能。我希望，下一次能够整理出一套 C 源码级调试的方案，不仅方便自己，也方便大家调试吧。

当然，多看点汇编也并无坏处，在这几次实验中，我不断重复地查看汇编代码，寻找其中部分与自己心理预期不一致的代码，也据此了解了不少 gcc 编译代码时会出现的行为，如结构体的自动对齐，函数调用帧栈的处理等等，同时还了解到 gcc 生成汇编时是默认四段合一，所有段寄存器都是相同的，这个坑到了不少人，

我听说过一些同学由于有一个段寄存器和别的不同，C 代码中刚好又生成了依赖这个段寄存器的代码，导致出现了极其难排查的 **bug**。从这个角度上说，汇编级别的 **debug** 是必不可少的，不过有了 C 代码级别的调试，想必我们调试的速度会更上一层楼。

附录 A 文件的组织

```
1  .
2  ├── a.img
3  ├── bochsrc.bxrc
4  ├── boot
5  │   ├── boot.asm
6  │   ├── fat.asm
7  │   ├── loader.c
8  │   ├── loader.lds
9  │   ├── loader_start.asm
10  │   ├── loader_start.h
11  │   ├── makefile
12  │   └── root.asm
13  ├── include
14  │   ├── basic.h
15  │   ├── c_run_time.a
16  │   ├── fsystem.h
17  │   ├── global.h
18  │   ├── kernel_run_time.a
19  │   ├── macro.inc
20  │   ├── proc.h
21  │   ├── stdio.h
22  │   ├── string.h
23  │   ├── style.h
24  │   ├── system_call.h
25  │   ├── type.h
26  │   └── utilities.h
27  ├── kernel
28  │   ├── kernel.asm
29  │   ├── makefile
30  │   ├── start.c
31  │   └── t.lds
32  ├── lib
33  │   ├── basic.asm
34  │   ├── makefile
35  │   └── stdio.c
```

```
36 |   ├── string.c
37 |   └── style.c
38 | ├── service
39 |   ├── filesystem.c
40 |   ├── global.c
41 |   ├── makefile
42 |   ├── proc.c
43 |   ├── system_call.c
44 |   └── tty.c
45 | ├── makefile
46 | ├── readme.md
47 | └── user
48 |     ├── makefile
49 |     ├── ouch.c
50 |     ├── test_a.c
51 |     ├── test_b.c
52 |     └── t.lds
```