

Report for Lab08

Stu : 金泽文

No.PB15111604

实验目的：

在实验 7 的基础上，实现 Round Robin 算法的进程调度机制。

实验内容：

1. 实现中断初始化和中断管理，提供缺省中断处理入口
2. 实现开中断和关中断，接口 enableIRQ()、disableIRQ()
3. 实现时钟初始化和时钟中断
4. 实现时间片轮转调度算法
5. 修改 osStart 原语，以增加相关功能的初始化

我的完成情况：

我完成了所有内容：

1. 时钟中断的实现，右上角时钟的实现。
2. FCFS 调度
3. Round Robin 调度。

当然，完成的还有整整四天的调试经历之后的心性的提升。

时间片：1s

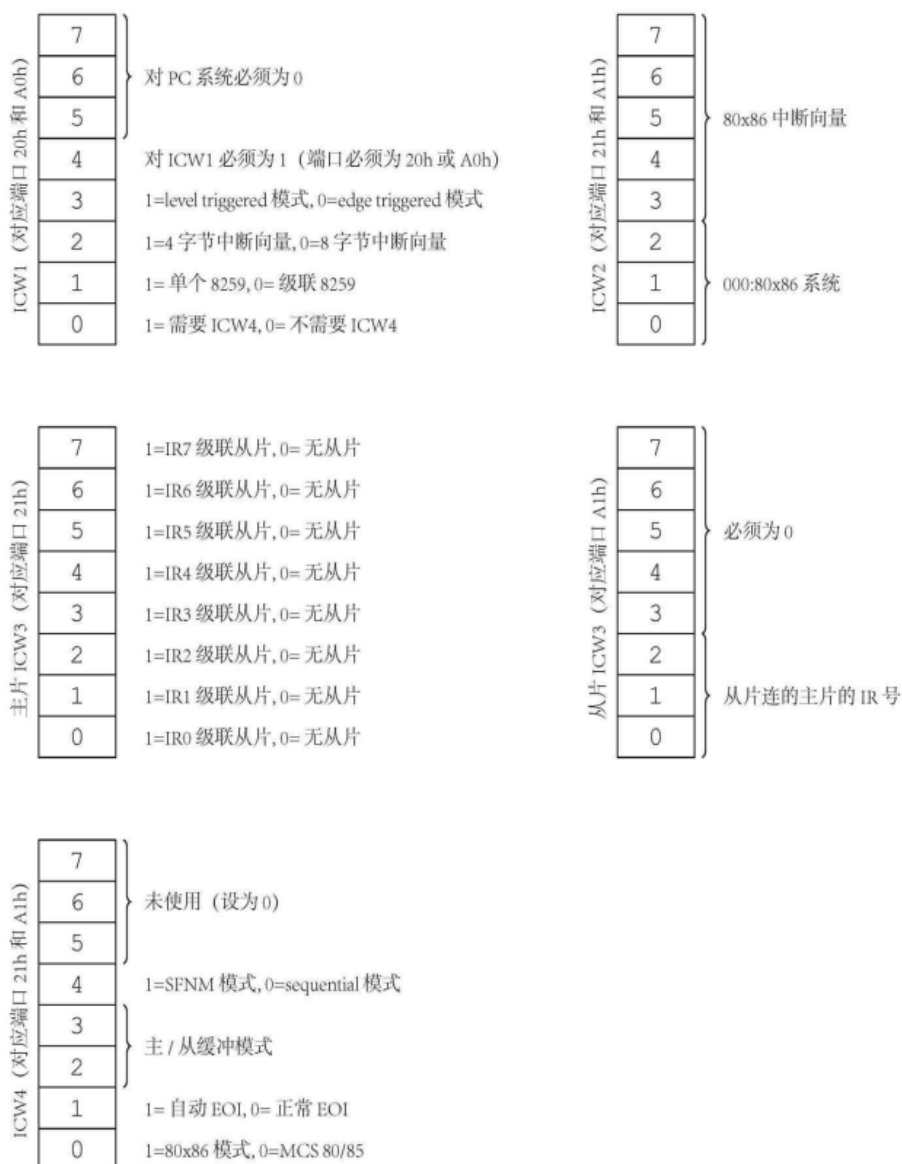
关于中断机制的学习与调研，以及对应的实现：

1. 可编程中断控制器 8259A：

可屏蔽中断与 CPU 的关系通过 8259A 建立起来。8259A 由主从两片组成。初始化的过程是：

- 1.往端口 20h（主片）或 A0h（从片）写入 ICW1。
- 2.往端口 21h（主片）或 A1h（从片）写入 ICW2。
- 3.往端口 21h（主片）或 A1h（从片）写入 ICW3。
- 4.往端口 21h（主片）或 A1h（从片）写入 ICW4。

ICW 的格式如下：



所以，先置 ff：

```
mov $0xff, %al
out %al, $0x21
out %al, $0xA1
```

再按照格式依次设置。其中中断向量 0x20 对应 IRQ0，0x28 对应 IRQ8。

所以 IRQ0~IRQ7 对应用户定义中断的 0x20~0x27，IRQ8~IRQ15 对应 0x28~0x2f。

2.8253-时钟中断的机制：

时钟中断由 PIT (Programmable Interval Timer) 芯片触发。在 IBM XT 中，这个芯片用的是 Intel 8253，所以用 8253 称呼。

8253 有 3 个计数器，如图：

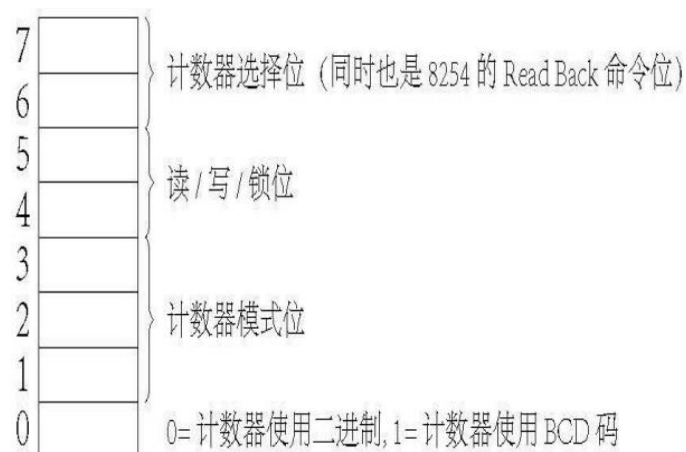
计数器	作用
Counter0	输出到 IRQ0，以便每隔一段时间让系统产生一次时钟中断
Counter1	通常被设为 18，以便大约每 15 μ s 做一次 RAM 刷新
Counter2	连接 PC 喇叭

时钟中断由 Counter0 产生。

计数器的工作原理：PC 上的输入频率是 1193180Hz，每个周期计数器值减 1，减到 0 时触发。为了让系统每 10ms 触发一次中断，也就是让我们的时间片为 10ms，所以需要设置计数器为 11931。为了设置，需要知道 8253 的端口，如下：

端口	描述
40h	8253 Counter0
41h	8253 Counter1
42h	8253 Counter2
43h	8253 模式控制寄存器 (Mode Control Register)

为了写我们的 Counter0 端口，需要先设置 43h 的 MCR 寄存器。MCR 格式如下：



模式位如下：

模式位值			模式	名称
3	2	1		
0	0	0	模式 0	interrupt on terminal count
0	0	1	模式 1	programmable one-shot
0	1	0	模式 2	rate generator ← 我们的时钟中断采用此模式
0	1	1	模式 3	square wave rate generator
1	0	0	模式 4	software triggered strobe
1	0	1	模式 5	hardware triggered strobe

读写位如下：

位		描述
5	4	
0	0	锁住当前记数值 (以便于读取)
0	1	只读写高字节
1	0	只读写低字节
1	1	先读写低字节, 再读写高字节

选择位如下：

位		描述
7	6	
0	0	选择 Counter0
0	1	选择 Counter1
1	0	选择 Counter2
1	1	对 8253 而言非法，对 8254 是 Read Back 命令

根据以上格式，0x43 处应该写入 0x34。

```
init8253:
    mov $0x34, %al
    out %al, $0x43
```

按字节输出到 0x40。

```
mov $(11932 & 0xff), %al
out %al, $0x40

mov $(11932 >> 8), %al
out %al, $0x40
```

最后，为了打开时钟中断，需要

```
mov $0xFE, %al
out %al, $0x21
```

2.建立并初始化 IDT

根据 google 到的下图，得到 lidt 指令的用法。

Opcode	Mnemonic	Description
0F 01 /2	LGDT m16&32	Load m into GDTR.
0F 01 /3	LIDT m16&32	Load m into IDTR.

Description
Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.
The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.
See "SGDT-Store Global Descriptor Table Register" in Chapter 4 for information on storing the contents of the GDTR and IDTR.

所以得到：

```
idt_reg:
    .word 0x0800
    .long 0x00001000
```

查到 idt 中 entry 的格式：

Name	Bit	Full Name	Description																				
Offset	48..63	Offset 16..31	Higher part of the offset.																				
P	47	Present	Set to 0 for unused interrupts.																				
DPL	45,46	Descriptor Privilege Level	Gate call protection. Specifies which privilege Level the calling Descriptor minimum should have. So hardware and CPU interrupts can be protected from being called out of userspace.																				
S	44	Storage Segment	Set to 0 for interrupt and trap gates (see below).																				
Type	40..43	Gate Type 0..3	Possible IDT gate types : <table><tr><td>0b0101</td><td>0x5</td><td>5</td><td>80386 32 bit task gate</td></tr><tr><td>0b0110</td><td>0x6</td><td>6</td><td>80286 16-bit interrupt gate</td></tr><tr><td>0b0111</td><td>0x7</td><td>7</td><td>80286 16-bit trap gate</td></tr><tr><td>0b1110</td><td>0xE</td><td>14</td><td>80386 32-bit interrupt gate</td></tr><tr><td>0b1111</td><td>0xF</td><td>15</td><td>80386 32-bit trap gate</td></tr></table>	0b0101	0x5	5	80386 32 bit task gate	0b0110	0x6	6	80286 16-bit interrupt gate	0b0111	0x7	7	80286 16-bit trap gate	0b1110	0xE	14	80386 32-bit interrupt gate	0b1111	0xF	15	80386 32-bit trap gate
0b0101	0x5	5	80386 32 bit task gate																				
0b0110	0x6	6	80286 16-bit interrupt gate																				
0b0111	0x7	7	80286 16-bit trap gate																				
0b1110	0xE	14	80386 32-bit interrupt gate																				
0b1111	0xF	15	80386 32-bit trap gate																				
0	32..39	Unused 0..7	Have to be 0.																				
Selector	16..31	Selector 0..15	Selector of the interrupt function (to make sense - the kernel's selector). The selector's descriptor's DPL field has to be 0.																				
Offset	0..15	Offset 0..15	Lower part of the interrupt function's offset address (also known as pointer).																				

这一部分我选择的方案是编写 c 函数然后在 32.s 中调用

根据上图，p 设置为 1，dpl，s 设置为 0，type 设置为 e；selector 设置为 ds，

offset 为中断调用的函数，所以得到：

```

158 struct{
159     short offset_h;
160     char type;
161     char zero;
162     short ds;
163     short offset_l;
164 }idt_entry;
165 long IDT_ADDRESS = 0x1000;
166 extern INTERRUPT();
167 void set_idt(){
168     long x = (void (*)())INTERRUPT;
169
170     //long INTERRUPT = 0x7e82;
171     long address = IDT_ADDRESS;
172     idt_entry.offset_h = (short)((x >> 16) & 0xff);
173     idt_entry.type = 0x8e;
174     idt_entry.zero = 0;
175     idt_entry.cs = 0x08;
176     idt_entry.offset_l = (short)(x & 0xffff);
177     for(int i = 0; i < 0x100; i++){
178         *(short*)(address + 8 * i) = idt_entry.offset_l;
179         *(short*)(address + 8 * i + 2) = idt_entry.cs;
180         *(char*)(address + 8 * i + 4) = idt_entry.zero;
181         *(char*)(address + 8 * i + 5) = idt_entry.type;
182         *(short*)(address + 8 * i + 6) = idt_entry.offset_h;
183     }
184 }

```

期间因为外部变量声明，外部函数声明，变量类型转换，花了大量的时间，内核级的调试还是不够熟练，也暴露了对c的掌握的不足。

中断机制的检查与时钟的实现：

在上述调研与实现之后，先检查一下是否有 bug。检查的方法是设计一个 clock 函数，并且在 32.s 里的中断处理程序中调用，通过不调用 myMain 而是死循环观察时钟。

幸亏发现及时，发现了很多很多小 bug 并且一一排除，这一过程用了一天半的时间。

实现的 clock 函数如下：

```
129 void clock(){
130     msecond ++;
131     if(msecond != 1 && msecond % 100 != 0) return;
132     int hour = msecond / 100 / 60 / 60;
133     int minute = (msecond / 100 / 60) % 60;
134     int second = (msecond / 100) % 60;
135     int where = 0xb8000 + 160;
136     *(char *)(where - 2) = (second % 10) + '0';
137     *(char *)(where - 1) = 0x7;
138     *(char *)(where - 4) = (second - second % 10) / 10 + '0';
139     *(char *)(where - 3) = 0x7;
140
141     *(char *)(where - 6) = ':';
142     *(char *)(where - 5) = 0x7;
143
144     *(char *)(where - 8) = (minute % 10) + '0';
145     *(char *)(where - 7) = 0x7;
146     *(char *)(where - 10) = (minute - minute % 10) / 10 + '0';
147     *(char *)(where - 9) = 0x7;
148
149     *(char *)(where - 12) = ':';
150     *(char *)(where - 11) = 0x7;
151
152     *(char *)(where - 14) = (hour % 10) + '0';
153     *(char *)(where - 13) = 0x7;
154     *(char *)(where - 16) = (hour - hour % 10) / 10 + '0';
155     *(char *)(where - 15) = 0x7;
156 }
157
```

外部有一个 msecond 全局变量。

32.S 中代码如图：

```

new:
    call set_idt

    movl idt_reg, %ebx
    add $2, %ebx
    movl IDT_ADDRESS, %eax

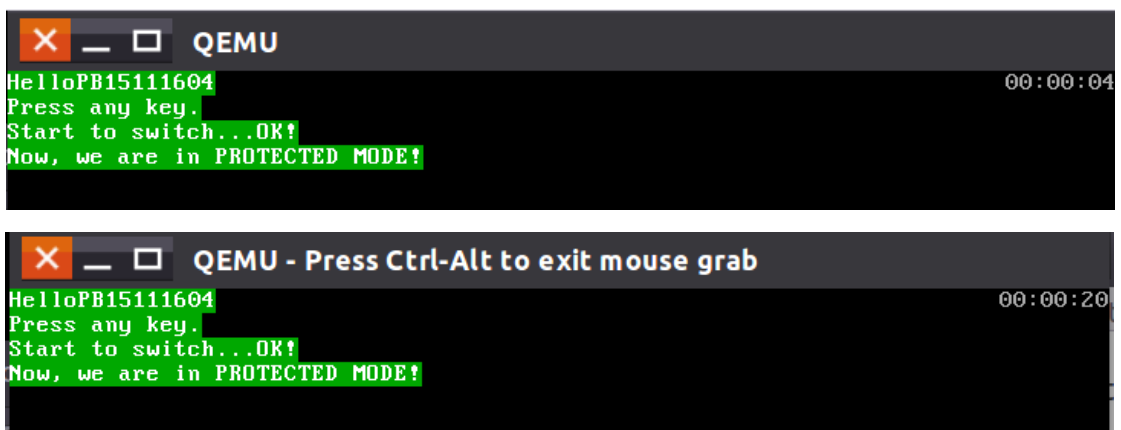
    movl %eax, (%ebx)
    lidt idt_reg
    call init8259A
    call init8253

    call clock
    #int $0x80
    sti

while:
    mov $10000, %ax
ss:
    sub $1, %ax
    jnz ss
    jmp while

```

得到的结果：



The image shows two screenshots of a QEMU virtual machine window. The first screenshot shows the initial state of the VM with the message "Now, we are in PROTECTED MODE!". The second screenshot shows the same state but with the title bar "QEMU - Press Ctrl-Alt to exit mouse grab".

结果合理。

可以开始设计 schedule 机制了。

Round Robin 的设计与实现：

Round Robin 的实现有以下几个难点：(出去上面的时钟中断的使用)

1. **Round Robin 中进程的顺序跟 pid 不完全一致。** 由于队列顺序会改变，不能只维护一个顺序数组。
2. **保存现场以返回。** 不同于 lab6，lab7，这次的进程调度室用 round robin，之前都是 fcfs，不用考虑返回之前运行过的函数的问题，而这次必须保存好。
3. **iret。** 调用中断 routine 的时候，如果中途 call schedule，那么有可能会发生栈溢出。因为中断次数很多。所以应避免直接 call。

针对这些难点，解决方法与实现如下：

维护队列

在之前进程数组 jobs 之外，设置 fcfs 队列：

```
unsigned int queue[MAX_TASK + 1];
```

出入队列：

```
void enqueue(unsigned int pid){
    queue[0] += 1;
    int number = queue[0];
    queue[number] = pid;
}

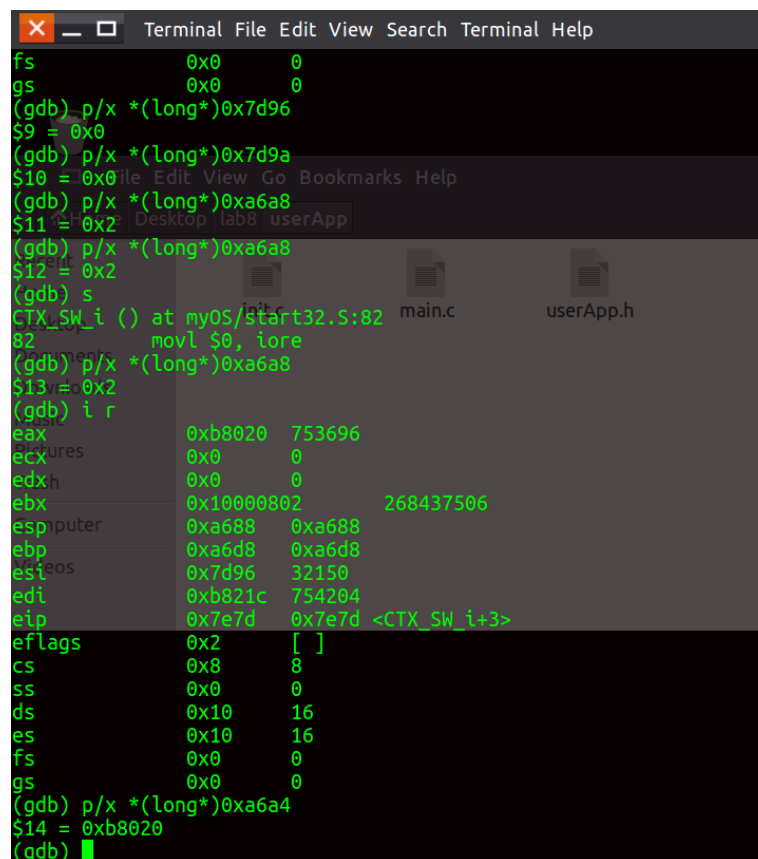
void dequeue(){
    int pid = pid_now;
    int number = queue[0];
    for(int i = 1; i <= number; i++){
        if(queue[i] == pid)
            break;
    }
    queue[0]--;
    for(; i < number; i++){
        queue[i] = queue[i + 1];
    }
}
```

以及得到下一个进程的 fcfs 函数：

```
//具体出队列操作在tskend实现。
//-1表示没有进程，返回i表示队列中下一个进程
int fcfs(){
    int number = queue[0];
    if(number == 0)
        return -1;
    int ret = queue[1];
    for(int i = 1; i < number ; i++)
        queue[i] = queue[i + 1];
    queue[number] = ret;
    return ret;
}
```

现场保存

这一步需要 pushf，在 gdb 调试过程中发现，只要 push 的顺序是 pushf, pusha，那么就是先存 eflags，再存其他八个 regs，pop 的时候顺序相反即可。



```
fs      0x0      0
gs      0x0      0
(gdb) p/x *(long*)0x7d96
$9 = 0x0
(gdb) p/x *(long*)0x7d9a
$10 = 0x0
(gdb) p/x *(long*)0xa6a8
$11 = 0x2
(gdb) p/x *(long*)0xa6a8
$12 = 0x2
(gdb) s
CTX_SW_i () at myOS/start32.S:82
82      movl $0, iore
(gdb) p/x *(long*)0xa6a8
$13 = 0x2
(gdb) i r
eax      0xb8020  753696
ecx      0x0      0
edx      0x0      0
ebx      0x10000802  268437506
esp      0xa688   0xa688
ebp      0xa6d8   0xa6d8
esi      0x7d96   32150
edi      0xb821c  754204
eip      0x7e7d   0x7e7d <CTX_SW_i+3>
eflags   0x2      [ ]
cs       0x8      8
ss       0x0      0
ds       0x10     16
es       0x10     16
fs       0x0      0
gs       0x0      0
(gdb) p/x *(long*)0xa6a4
$14 = 0xb8020
(gdb)
```

现在需要知道的就是 iret 都干了什么。

根据 google 的结果：

Getting to ring 3 can be done using `iret` because the way it works has been documented. When you receive an interrupt, the processor pushes:

1. The stack segment and pointer (ss:esp), as 4 words
2. EFLAGS
3. The return code segment and instruction pointer (cs:eip), as 4 words
4. An error code, if required.

只是多了一个 eflags。所以只需要：

```
66     .global INTERRUPT
67 INTERRUPT:
68     pusha
69     pushf
70     call clock
71
72     mov $0x20, %al
73     out %al, $0x20 #EOI
74     movl $0, iore
75
76     movl xxx, %eax
77     subl $1, %eax
78     jz idleidle
79     call test_tick
80     movl TICK, %eax
81     addl $0, %eax
82     jnz IRET
83
84     movl %esp, preStk
85     #movl schedule_point, %esp
86     call schedule
87 IRET:
88     popf
89     popa
90     iret
91
92 idleidle:
93     popf
94     popa
95     iret
96
```

这里，xxx 是为了判断是否调度完

所有程序，在 idle 中的中断。如果是

idle 中的中断，那么当然没有必要

tick，没有必要 schedule，所以刷新

clock 之后就可以 pop 并 iret 了。

在非 idle 时，需要更新进程的

tick，并且判断是

否泡足了一个时间

片，也就是 1s。所

以：

myTCB 需要

增加 tick 变量。

test_tick 函数如下：

```
90 void test_tick(){
91     jobs[pid_now].tick = (jobs[pid_now].tick + 1) % 100;
92     TICK = jobs[pid_now].tick;
93 }
```

调度函数 schedule 如下：注意每

次调度之前需要打开中断，

```
typedef struct myTCB{
    unsigned int pid; //进程序
    enum state{
        terminated, //
        //作为进程结束的标志，后面
        ready, //creat
        running, //runni
        waiting
    }state;
    unsigned long *stack;
    int tick;
}myTCB;
```

```

void schedule(){
    // disable_interrupt();
    jobs[pid_now].stack = preStk;
    int pid;

    while((pid = fcfs()) != -1){ //如果还有进程
        nextStk = jobs[pid].stack;
        init8253();
        pid_now = pid;

        CTX_SW_s();
        if(iore == e)continue;

    }

    idle();
    //Should never arrive here.
}

```

另外，根据 CTX_SW 的调用点的不同，
 设置了两个:CTX_SW_e，CTX_SW_s，前者是
 TskEnd 函数调用的，后者是 schedule 调用
 的。有一些代码是中途调试所用。

后来优化之后发现没有必要，得到：

```

51     .global CTX_SW_s
52 CTX_SW_e:
53 CTX_SW_s:
54
55     pusha
56     pushf
57     movl %esp, preStk
58
59     movl nextStk, %esp
60     popf
61     sti
62     popa
63     ret
64

```

```

38 CTX_SW_e:
39     pusha
40     pushf
41     # movl preStk, %eax
42     # movl %esp, (%eax)
43     movl %esp, preStk
44     movl nextStk, %esp
45     popf
46     popa
47     ret
48
49     .global CTX_SW_s
50
51 CTX_SW_s:
52
53     pusha
54     pushf
55     # movl preStk, %eax
56     # movl %esp, (%eax)
57     movl %esp, preStk
58     movl %esp, schedule_point
59     movl nextStk, %esp
60     popf
61     sti
62     popa
63     ret
64

```

另外，eflags 的初始值。之前因为 eflags 设置错误，查了一天多的 bug，最后经小

```
63 ▾ unsigned Long* stack_init(unsigned Long *stack, void (*task)(void
    )){
64     *(stack--) = (unsigned Long) 0x08;
65     *(stack--) = (unsigned Long) task;
66     *(stack--) = (unsigned Long) 0xAAAAAAAA;
67     *(stack--) = (unsigned Long) 0xCCCCCCCC;
68     *(stack--) = (unsigned Long) 0xDDDDDDDD;
69     *(stack--) = (unsigned Long) 0xBBBBBBBB;
70     *(stack--) = (unsigned Long) 0x44444444;
71     *(stack--) = (unsigned Long) 0x55555555;
72     *(stack--) = (unsigned Long) 0x66666666;
73     *(stack--) = (unsigned Long) 0x77777777;
74     *(stack) = (unsigned Long)0x0287;
75     return stack;
76 }
77
```

妞提醒改成了 0287。

原语流程:

从 32.s 的 call myMain 之前的新内容：

Set_idt 建立 idt 表，将中断处理程序与 entry 关联 → 8259A 设置芯片对应的中断

→ 8253 初始化计数器，10ms 一次触发时钟中断。

call myMain 之后：

清屏 → 输出 helloworld → osstart → createTsk(initTskBody) → pMemInit →

dPartitionInit → clock 输出初始时钟 → schedule 调度 → 找到下一个进程 → CTX_sw

→ 保存现场 → 开中断 → 切入进程。

如果进程结束，→ tskend → schedule 继续。

如果进程被中断 → 设置进程的 tick，如果满足一个时间片，即 1s → 则转入调度。

如果不满足 → 则直接恢复中断。

最后 idle 开中断 → 死循环 → 触发中断 → 更新 clock → 恢复中断，继续死循环。

测试结果：

```
QEMU
myMain:HELLO WORLD! 00:00:02
memory available:0x08000000
*****INIT START

QEMU - Press Ctrl-Alt to exit mouse grab
myTSK2::2 WORLD! 00:00:19
myTSK2::3
myTSK1::7 ble:0x08000000
myTSK1::8 START
myTSK0::9
myTSK2::4
myTSK2::5
myTSK2::6
myTSK1::9
myTSK2::7
myTSK2::8
myTSK2::9 END
myTSK1::2
myTSK1::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK2::0 IDLE
myTSK2::1
myTSK1::4
myTSK1::5
myTSK1::6
myTSK0::7
myTSK0::8
```


实验总结（主要是 bug 与 debug）

实验总结的内容主要是 bug 与调试的回顾吧，因为 90%的时间都在调试。Bug 太

多，说一些印象比较深刻的：

1. x86 汇编立即数前忘了+ '\$' 符号，查得：

ax there are no register prefixes or immed prefixes. In AT&T however registers are prefixed with a '%' and immed's are prefixed with a '\$'. Intel syntax l Also if the first hexadecimal digit is a letter then the value is prefixed by a '0'.

	AT&T Syntax
ax, 1	movl \$1, %eax
0x, 0ffh	movl \$0xff, %ebx
0h	int \$0x80

of Operands.

n of the operands in Intel syntax is opposite from that of AT&T syntax. In Intel syntax the first operand is the destination, and the second operand is the and is the destination. The advantage of AT&T syntax in this situation is obvious. We read from left to right, we write from left to right, so this way is o

	AT&T Syntax
est, source	instr source, dest
ax, [ecx]	movl (%ecx), %eax

operands.

operands as seen above are different also. In Intel syntax the base register is enclosed in '[' and ']' whereas in AT&T syntax it is enclosed in '(' and ')'.

	AT&T Syntax
ax, [ebx]	movl (%ebx), %eax
ax, [ebx+3]	movl 3(%ebx), %eax

orm for instructions involving complex operations is very obscure compared to Intel syntax. The Intel syntax form of these is segreg:[base+index*scale disp/segreg are all optional and can simply be left out. Scale, if not specified and index is specified, defaults to 1. Segreg depends on the instruction and he instruction whereas in pmode its unnecessary. Immediate data used should not '\$' prefixed in AT&T when used for scale/disp.

	AT&T Syntax
--	-------------

2. 不同源文件之间变量名的引用。

在 c 文件中使用 s 文件中的变量时，必须在 s 里声明 globl，同时尽量使用 extern 声明，简单的数值变量可能只报 warning，但是把 s 中的 label 当成数值变量时，必须先生命为 extern （）函数之后经过类型转换再使用，否则会报找不到变量名。

3. 其他的大多是逻辑上的 bug。包括对原理和机制的错误理解：

比如 `iret` 没有想象的复杂，当时做到从中断返回的时候是在晚上一整天以及前一天都在调一些智障的 bug，心态很不好，所以就产生了不良心态。之后理清思路看之后好了很多。

还有整个中断机制的理解。确实是理解了机制之后 bug 就会消除大部分。

4. 还有 debug 技能的积累，学到了一些新的操作：

gdb 里的比如 `x/10i` 看内存指令，比如 `b *0x00f0098` 设置内存断点，再比如 `wa *(long *)variable` 等等。Gdb 的操作熟练了很多。

期间还熟练了一些 `grep`，`objdump` 等操作。

可以说整个调试的技能与心态的处理强大了很多。

这是最后一个实验了。

最后要感谢一下陈老师和三位助教的积极的引导、指导，还有无尽的付出。

结课辣！！！！

么么哒！！！！

！祝老师和助教假期愉快！

