操作系统 lab3 实验报告

151220094 宋俊举

1、进程控制块设计

a)首先完善 TrapFrame 结构，如图：

```
41  struct TrapFrame {
42      uint32_t gs,fs,es,ds;
43      uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
44      int32_t irq;
45      uint32_t error;
46      uint32_t eip,cs,eflags,esp,ss;
47  };
```

b)pcb 进程控制块结构设计：

在 kernel/include 中新建 sched.h 文件。

```
#include "x86.h"
#include "device.h"
#define DEAD 0
#define RUNNING 1
#define RUNNABLE 2
#define BLOCKED 3
#define MAX_STACK_SIZE 4096
#define MAX_PCB_NUM 10
typedef struct PCB{
    struct TrapFrame *tf;
    int state;
    int timeCount;
    int sleepTime;
    unsigned int pid;
    unsigned char stack[MAX_STACK_SIZE];
    struct PCB *prev;
    struct PCB *next;
}PCB;

extern PCB pcb[MAX_PCB_NUM];
extern PCB idle;
extern PCB *current;
extern PCB *runnable;
extern PCB *blocked;
extern PCB *free;
```

设置了四个状态，依次为无进程态(DEAD)，运行态(RUNNING),就绪态(RUNNABLE),阻塞态(BLOCKED)。

声明 4 个指针，current 为当前正在运行的进程，runnable 为就绪态链表指针，Blocked 为阻塞态链表指针，free 为可以使用的进程控制块。

Idle 表示 idle 线程，用于判断，无实际意义。

2、FORK、SLEEP、EXIT 系统调用

三个系统调用均在 kernel/kernel/sched.c 中

a) FORK 调用：

```
85  void FORK()
86  {
87      gdt[7] = SEG(STA_X|STA_R,0x100000,0x100000,DPL_USER);
88      gdt[8] = SEG(STA_W,0x100000,0x100000,DPL_USER);
89      int i;
90      for(i=0;i<0x100000;i++)
91      {
92          *(char *)(0x300000+i) = *(char *)(0x200000+i);
93      }
94      PCB *t = delfirst(&free);
95      *t = (*current);
96      t->pid = t - pcb;
97      t->tf = (void*)(((uint32_t)t - (uint32_t)current) + (uint32_t)current->tf);
98      t->tf->ds = (USEL(8));
99      t->tf->es = (USEL(8));
100     t->tf->ss = (USEL(8));
101     t->tf->cs = (USEL(7));
102     t->state = RUNNABLE;
103     t->timeCount = 10;
104     t->tf->eax = 0;
105     current->tf->eax = t->pid;
106     addpcb(&runnable,t);
107 }
```

新建两个段，分别表示代码段和数据段，基址为 0x100000，并将数据复制到新的内存区域中。

从 free 链表中取出一个可用的进程控制块 t，依次设置各项。并将新进程控制块加入到就绪态中。

b)SLEEP 调用：

```
108 void SLEEP(unsigned time)
109 {
110     current->sleepTime = time;
111     current->state = BLOCKED;
112     addpcb(&blocked,current);
113     current = delfirst(&runnable);
114     if(current==NULL)
115     {
116         current = &idle;
117     }
118     else
119     {
120         current->state = RUNNING;
121         current->timeCount = 10;
122         tss.esp0 = (int)current->stack + 4096;
123     }
124 }
```

c)EXIT 调用:

```
125  void EXIT()
126  {
127      current->state = DEAD;
128      addpcb(&free,current);
129      current = delfirst(&runnable);
130      if(current==NULL)
131          current = &idle;
132      else
133      {
134          current->state = RUNNING;
135          current->timeCount = 10;
136          tss.esp0 =(int)current->stack + 4096;
137      }
138  }
```

d)注意事项：

FORK 系统调用时，使用新段来控制访问的堆栈位置，实现完整的分段机制时 bug 太多因此目前只实现了可以开一个进程的段。

每次如果无进程可以运行，就将 current 置为&idle

每次切换进程时，更新 tss.esp0 指向一个进程的核心栈顶，保证通过系统调用陷入内核态时的切换到正确的核心栈。

e)注册系统调用：

在 lib/syscall.c 中新建如下三个函数，并在 lib.h 中进行了声明。

```
145  int fork()
146  {
147      return syscall(3,0,0,0,0,0);
148  }
149  void sleep(unsigned time)
150  {
151      syscall(4,0,time,0,0,0);
152  }
153  void exit()
154  {
155      syscall(5,0,0,0,0,0);
156  }
157
```

f)在 kernel/kernel/irqHandle.c 的 syscallHandle 下添加三个函数

```
        case 3:
            FORK();
            break;
        case 4:
            SLEEP(tf->ecx);
            break;
        case 5:
            EXIT();
            break;
        default:assert(0);
```

3、开启时钟中断

a) 在 kernel/kernel 下新建 timer.c 文件将下列代码写入文件中

```
#include "x86.h"
#define TIMER_PORT 0x40
#define FREQ_8253 1193182
#define HZ 100

void initTimer() {
    int counter = FREQ_8253 / HZ;
    outByte(TIMER_PORT + 3, 0x34);
    outByte(TIMER_PORT + 0, counter % 256);
    outByte(TIMER_PORT + 0, counter / 256);
}
```

b) 在 kernel/kernel/idt.c 中添加 idt 表项，新建 0x20 号中断向量，并在 doIrq.S 中新建中断处理函数 irqTimer()

```
setTrap(idt + 0x20,SEG_KCODE,(uint32_t)irqTimer,DPL_KERN);
```

```
.global irqTimer
irqTimer:
    pushl $0
    pushl $0x20
    jmp asmDoIrq
```

c)在 kernel/main.c 中调用 initTimer()开启时钟中断。

4、实现进程调度

a)实现 schedule()函数

```
140  void schedule()
141  {
142      if(current == &idle)
143      {
144          current = delfirst(&runnable);
145          if(current==NULL)
146              current = &idle;
147          else
148          {
149              current->state = RUNNING;
150              current->timeCount = 10;
151              tss.esp0 = (int)current->stack + 4096;
152          }
153          return;
154      }
155      if(current->timeCount<=0)
156      {
157          current->state = RUNNABLE;
158          addpcb(&runnable,current);
159          current = delfirst(&runnable);
160          current->state = RUNNING;
161          current->timeCount = 10;
162          tss.esp0 = (int)current->stack + 4096;
163      }
164      return;
165  }
```

判断当前进程是否为 idle，如果为 idle 则判断是否有就绪进程可以切换，有则切换，无则返回；如果当前进程不为 idle 则判断其时间片是否耗尽，如果耗尽则将其加入就绪队列中，并从就绪队列中取出一个进程执行，更新 tss。

b)实现 Block2Runnable 函数

```
void Block2Runnable()
{
    PCB *t = blocked;
    while(t!=NULL)
    {
        t->sleepTime--;
        if(t->sleepTime<=0)
        {
            t->state=RUNNABLE;
            t->timeCount = 10;
            PCB *r = t;
            if(r->prev==NULL&&r->next==NULL)
                blocked = NULL;
            if(r->prev!=NULL)
                r->prev->next = r->next;
            if(r->next!=NULL)
                r->next->prev = r->prev;
            t=t->next;
            addpcb(&runnable,r);
        }
        else
            t=t->next;
    }
}
```

依次将阻塞态的各进程 sleepTime-1 并判断 sleeptime 是否耗尽，如果耗尽则将其取出，并置入到就绪态队列中。

c)更改 irqHandle 函数，保存寄存器状态并执行调度。

```
void irqHandle(struct TrapFrame *tf) {
    /*
     * 中断处理程序
     */
    /* Reassign segment register */
    current->tf = tf;
    switch(tf->irq) {
        case -1:
            break;
        case 0xd:
            GProtectFaultHandle(tf);
            break;
        case 0x80:
            syscallHandle(tf);
            break;
        case 0x20:
            putChar('t');
            current->timeCount--;
            Block2Runnable();
            break;
        default:assert(0);
    }
    schedule();
}
```

首先将当前进程的 tf 指针指向保存的 tf，新添加 0x20 号中断处理，将当前时间片减一，并执行 Block2Runnable 函数处理阻塞态队列。最后在 switch 外执行调度函数。

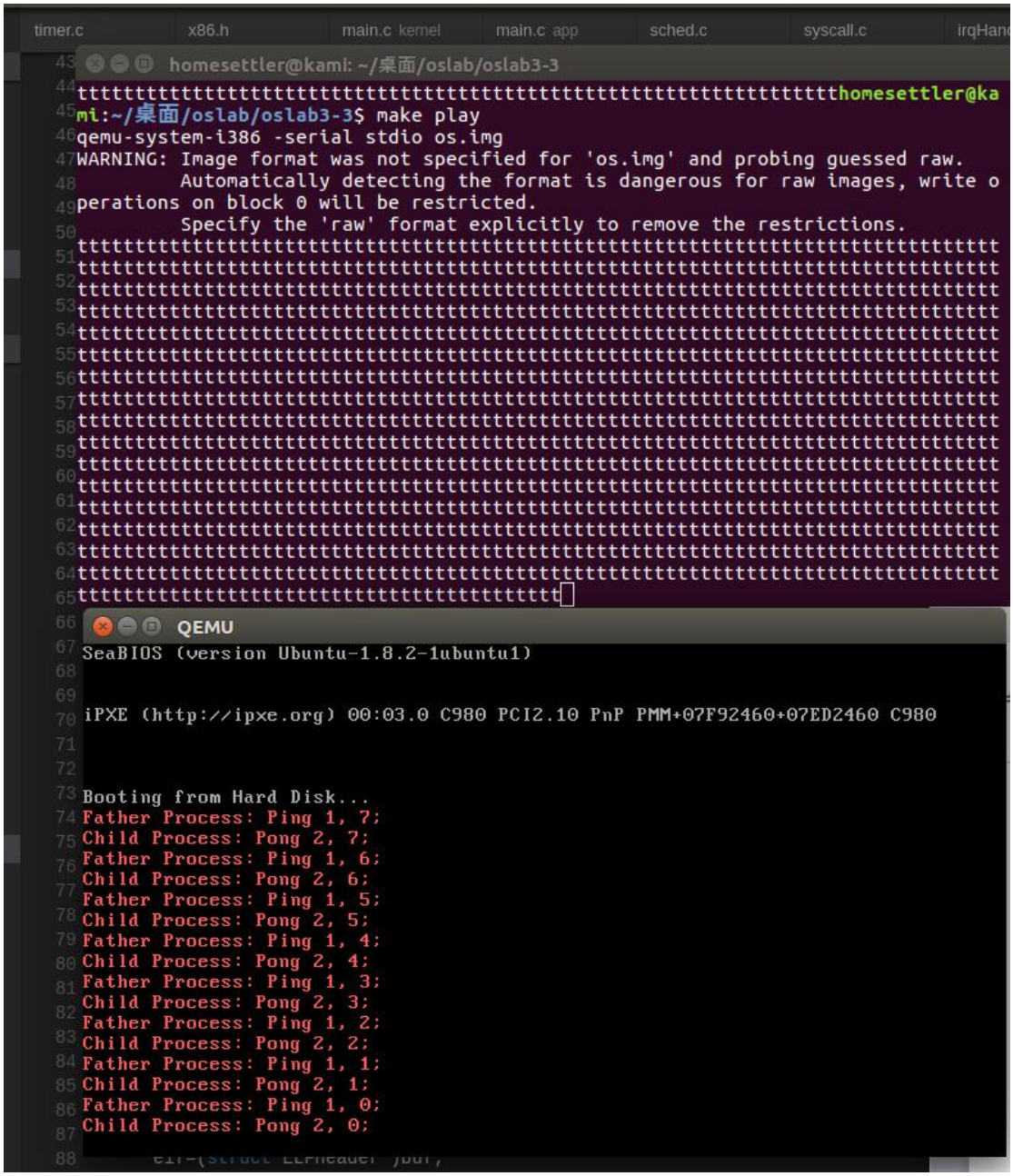
d)更改 doIrq.S 的 asmDoirq 函数

```
.global asmDoIrq
asmDoIrq:
    cli
    pushal // push process state into kernel stack
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushl %esp
    movw $0x10,%ax
    movw %ax,%es
    movw %ax,%ds
    call irqHandle
    addl $4, %esp
    mov (current),%eax
    mov (%eax),%esp
    popl %gs
    popl %fs
    popl %es
    popl %ds
    popal
    addl $4, %esp //interrupt vector is on top of kernel stack
    addl $4, %esp //error code is on top of kernel stack
    sti
    iret
```

首先关闭中断，将所有寄存器保存，切换段描述符至内核态。调用 irqHandle 函数后，将当前的 esp 切换为当前进程的 tf 位置，并依次将各寄存器出栈。就完成了一次进程调度。

c)在 kernel/kernel/kvm.c 的 loadUMain 函数中初始化所有进程控制块和进程控制队列，并设置 pcb[0],将其指向用户程序，并将其放入就绪队列中，然后使用 sti 指令开中断，并在 while 循环中，使用 hlt 指令等待中断到来。

5、实验结果：



6、遇到的问题以及解决方案

因为程序的代码执行的内存位置还是在旧进程，而又无法更改程序的执行位置，所以使用新段给新进程执行时的内存位置一个偏移量。

因为 current 位置等全局变量是常数所以在新进程时即使陷入内核态，如果不改 变段寄存器的话会使接下来执行代码时增加偏移量，所以将寄存器入栈后调用 irqHandle 之前需要更改段寄存器的值。