操作系统 lab3 进程切换

邱梓豪

141130077

1.首先在 kernel/include/pcb.h 中建立 pcb 数据结构及定义该数据结构上可以进行的操作。

```
#define MAX_PCB 20          // max of PCBs

#define NEW 0           // define some states of a process
#define RUNNABLE 1
#define RUNNING 2
#define SLEEPING 3
#define DEAD 4

struct ProcessTable {  // struct of PCB
    struct TrapFrame* tf;
    unsigned char kStack[1024];
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    struct ProcessTable* prev;
    struct ProcessTable* next;
};

typedef struct ProcessTable PCB;

PCB pcb_pool[MAX_PCB];     // build a pool to store all pcbs
```

PCB 结构体的定义如上，我这里使用双向循环链表对 PCB 就绪队列和 PCB 等待队列进行组织，所以结构体中有两个相应的指针 prev 和 next。tf 指向的是进程切换时保存的 TrapFrame，我这里将 TrapFrame 保存在每个进程相应的内核栈 kStack 中。state 表示当前进程状态，timeCount 为剩余时间片，sleepTime 为剩余

等待时间，pid 为进程 id。

然后我建立了一个大小为 20 的进程池，保存所有 pcb。

```
//the first node of the following 2 list are NOT belong to any process
PCB pcb_ready;   // point to the list of ready process
PCB pcb_wait;    // point to the list of waitting process(sleepTime!=0)

PCB* current;     // point to current process
PCB* idle;        // point to idle process
```

建立两个队列：就绪队列和等待队列。建立两个指针：指向当前进程的指针和指向 idle 的指针。

```
static inline void pcb_delete(PCB* p)  // delete a PCB
{
    p->prev->next = p->next;
    p->next->prev = p->prev;
}

/*  add a new PCB to the tail of the list
 *  p must point a new node which must be 'new' before
 */
static inline void pcb_add(PCB* list, PCB* p)
{
    p->next = list;
    list->prev->next = p;
    p->prev = list->prev;
    list->prev = p;
}
```

随后再添加对数据结构上的两个操作：从队列中删除一个 pcb 和向队列上添加一个 pcb。

这样，我就相当于在 pcb.h 这个文件中建立了一个“类”。

2.进程初始化，将用户程序包装成一个进程，通过调度来进入这个进程从而执行用户程序。

下面这个函数主要对 pcb 进行初始化，以及将用户程序包装成用户进程。参数 entry 是通过 loadUMain 函数获得的。进程初始化时，tf 指向自己的 kStack，ds、es、ss 指向自己的数据段，cs 指向自己的代码段。timeCount 设为 10，pid 设为 1，最后将这个 pcb 放入就绪队列中，当前进程设为 idle。

```
static inline void initPCB(uint32_t entry)  // init state of all PCB
{
    // init pcb_pool
    for (int i=0;i<MAX_PCB;i++){
        pcb_pool[i].state = DEAD;
    }

    // initialize the ready list
    pcb_ready.prev = &pcb_ready;
    pcb_ready.next = &pcb_ready;

    // initialize the wait list
    pcb_wait.prev = &pcb_wait;
    pcb_wait.next = &pcb_wait;

    idle = &pcb_pool[0];          // initialize process idle
    idle->pid = 0;
    idle->state = RUNNING;

    PCB* first_proc = &pcb_pool[1];      // make user app be the first proc
    first_proc->state = RUNNABLE;
    first_proc->tf = (void*)(pcb_pool[1].kStack+sizeof(pcb_pool[1].kStack)-
    first_proc->tf->ds = USEL(SEG_UDATA);
    first_proc->tf->es = USEL(SEG_UDATA);
    first_proc->tf->ss = USEL(SEG_UDATA);
    first_proc->tf->cs = USEL(SEG_UCODE);
    first_proc->tf->ebp = 0x200000;
    first_proc->tf->esp = 0x200000;
    first_proc->tf->eip = entry;         // entry ??
    first_proc->tf->eflags = 0x202;
    first_proc->timeCount = 10;
    first_proc->sleepTime = 0;
```

3.启动时钟源，并注册时钟中断。

```
#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);
}

void kEntry(void) {
    initSerial();// initialize serial port
    initIdt(); // initialize idt
    initIntr(); // initialize 8259a
    initSeg(); // initialize gdt, tss
    uint32_t entry = loadUMain(); // load user program, enter user space

    initPCB(entry);   // initialize PCB
    initTimer(); // initialize timer
    enableInterrupt();
    while(1){                // idle process
        waitForInterrupt();
        putScreen('0');
    }
```

在 kernel 的 main.c 中初始化时钟源。再在 idt.c 中为时钟中断添加处理函数：

```
setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);

setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER); // for int 0x80
setIntr(idt + 0x20, SEG_KCODE, (uint32_t)timer, DPL_KERN); // timer
/* 写入IDT */
saveIdt(idt, sizeof(idt));
```

然后在 doIrq.s 中加上 timer 的处理，这里我为时钟中断定的 irq number 为 0x10：

```
.global timer
timer:
    pushl $0x10
    jmp asmDoIrq
```

最后在 irqHandle 函数中加上对时钟中断的相应，即可：

```
switch(tf->irq) {
    case -1:
        break;
    case 0xd:
        GProtectFaultHandle(tf);
        break;
    case 0x80:
        syscallHandle(tf);
        break;
    case 0x10:
        disableInterrupt();
        schedule();
        enableInterrupt();
        break;
    default: assert(0,tf->irq);
}
```

4.重写 kernel 你的 mian.c

```
void kEntry(void) {
    initSerial();// initialize serial port
    initIdt(); // initialize idt
    initIntr(); // initialize 8259a
    initSeg(); // initialize gdt, tss
    uint32_t entry = loadUMain(); // load user program, enter

    initPCB(entry);   // initialize PCB
    initTimer(); // initialize timer
    enableInterrupt();
    while(1){                // idle process
        waitForInterrupt();
        putScreen('0');
    }

    assert(0,0);
}
```

kernel 的主要过程如下：加载用户程序获得用户程序的 entry 后，先初始化 pcb，此时将用户程序包装成用户进程并加入就绪队列中。然后初始化时钟源，开中断。这之后便会进入到一个无

限循环中，这个循环便是在背后运行的 idle“进程”，也就是说当程序处于sleep状态是，便会执行idle进程的任务，即输出‘0’。

5.创建调度程序，并在 irqHandle 中添加对时钟中断的处理。

我在 kernel 中新建了 schedule.c，用于进程调度。

```
extern TSS tss;

void schedule()
{

    for (PCB* p = pcb_wait.next; p != &pcb_wait && p->sleepTime>0; p=p->next){
        p->sleepTime--;
    }

    // first: renew wait list and ready list
    for(PCB* p = pcb_wait.next; p!=&pcb_wait; p=p->next)
    {
        if (p->sleepTime == 0){
            PCB* prev = p->prev;
            pcb_delete(p);
            p->state = RUNNABLE;
            pcb_add(&pcb_ready, p);
            p=prev;
        }
    }
```

显然每当一个新的时钟中断到来时，都应该先更新等待队列中的 pcb，如果某个 pcb 的等待结束，那么应该将这个 pcb 放入就绪队列中。

```
if (current != idle) {
    current->timeCount--;
    return;
}
```

如果当前进程不是 idle 的话，就将当前进程的时间片减 1，直接返回即可，不用切换进程。

```
    if (current == idle) {          // current process is idle and have other ready proc
ess
        if (pcb_ready.next != &pcb_ready){
            current = pcb_ready.next;
        }
    }
    else if (current->timeCount == 0) // current process time out
    {

        if (current->next != &pcb_ready){  // if have other ready process
            current->state=RUNNABLE;
            PCB* newProc = current->next;
            pcb_delete(current);
            pcb_add(&pcb_wait, current);
            current = newProc;
        }
        else                            // no other ready process
        {
            pcb_delete(current);
            pcb_add(&pcb_wait, current);
            current = idle;
        }
    }
    else return;
```

如果当前的进程是 idle，则检查就绪队列，如果就绪队列中有进程，则将 current 置为此进程。若当前时间片耗尽，则从就绪队列中找就绪进程，找不到则换上 idle 进程。

```
current->state = RUNNING;
current->timeCount = 10;
tss.esp0 = (uint32_t)(current->kStack+sizeof(current->kStack)); // change process
```

最后将内核栈换为当前进程的内核栈。

```
movl (current), %eax
movl (%eax), %esp
```

在内核栈切换时，current 表示的地址正是 tf 所指向的，tf 指向 pcb 内核栈的栈顶，所以通过这两步操作便完成了进程切换。

6.实现 fork，sleep，exit 三个系统调用。

（1）fork

```
gdt[gdt_index++] = SEG(STA_X|STA_R, startPos, segLen, DPL_USER); // code
gdt[gdt_index++] = SEG(STA_W,       startPos, segLen, DPL_USER); // data


memcpy((void*)startPos, (void*)startPos-segLen, segLen);
startPos += segLen;

int child_proc_i=19;            // index of child process
for (int i=0;i<MAX_PCB;i++)
    if(pcb_pool[i].state == DEAD)
        {child_proc_i = i; break;}
fork_copy(&pcb_pool[child_proc_i], current);
break;
```

fork 首先要为新进程分配数据段和代码段，然后将父进程的数据段和代码段拷贝过来，然后分配一个新的 PCB 给子进程，然后填写这个 PCB，具体过程如下：

```
void fork_copy(PCB* dst, PCB* src)
{
    *dst = *src;
    dst->tf = (void*)(dst->kStack+sizeof(dst->kStack)-sizeof(struct TrapFrame));
    dst->tf->eax = 0;
    dst->tf->ss = USEL(gdt_index-1); // point to data segment
    dst->tf->es = USEL(gdt_index-1);
    dst->tf->ds = USEL(gdt_index-1);
    dst->tf->cs = USEL(gdt_index-2); // point to code segment
    dst->pid = child_pid++;
    dst->state = RUNNABLE;
    pcb_add(&pcb_ready, dst);
    src->tf->eax = dst->pid;
}
```

首先将父进程的 pcb 完全拷贝给子进程，然后依次设置子进程 pcb 的 eax、ss、ds、es、cs、pid、state，最后将子节点的 pcb 加入等待队列中，父进程 pcb 的 eax 设为子进程的 pid 即可。

（2）sleep

```
case 3: current->state = SLEEPING;    // sleep
        current->sleepTime = tf->ecx;
        pcb_delete(current);
        pcb_add(&pcb_wait, current);   // put the sleeping process into wait lis

        current = idle;
        break;
```

将当前进程状态置为 SLEEPING，设置 sleepTime，然后将此进程

放入等待队列，将当前进程置为 idle 即可

（3）exit

```
current->state = DEAD;          //exit
pcb_delete(current);
current = idle;
break;
```

将当前进程状态置为 DEAD，然后从队列中删除，当前进程置为 idle 即可。

最后运行效果如下：

```
         Specify the 'raw' format explicitly to remove the
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
Father Process: Ping 1, 6;
Child Process: Pong 2, 6;
Father Process: Ping 1, 5;
Child Process: Pong 2, 5;
Father Process: Ping 1, 4;
Child Process: Pong 2, 4;
Father Process: Ping 1, 3;
Child Process: Pong 2, 3;
Father Process: Ping 1, 2;
Child Process: Pong 2, 2;
Father Process: Ping 1, 1;
Child Process: Pong 2, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
qemu-system-i386: terminating on signal 2
```

进程直接的切换如下：


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980