操作系统实验报告

Lab3

姓名：刘博

学号：141220065

计算机科学与技术系

2016.4.23

邮箱：1610266604@qq.com

1. 实验目的：

1. 添加时钟中断
2. 加载用户程序
3. 初始化用户进程
4. 开中断
5. 从时钟中断调度到用户进程中
6. 用户进程调用 fork（）系统调用创建子进程
7. 利用时钟中断调度父子进程
8. 一段时间后利用 exit（）结束进程，返回内核进程

2. 实验过程：

首先将 kernel 的函数进行更新，初始化 time 时钟中断，初始化用户 pcb，打开中断，并从时钟中断进入用户函数；

```
init_serial();                    //初始化串口输出
init_idt();
init_intr();
init_seg();
init_timer();
init_idle();
create_uthread(load_umain());
enable_interrupt();
while(1){
        putchar('^');
        wait_for_interrupt();
}
assert(0);
```

其中 load_umain（）函数是用来加载用户代码（与 lab2 一致），其中 create_uthread（）函数用来创建用户的 pcb，函数如下：

```
PCB *create_uthread(uint32_t entry) {
        //allocate a pcb
        assert(pcbs_avl < NR_PCBS);
        PCB *pcb = &(PCBs[pcbs_avl]);
        pcb->pid = pcbs_avl;
        pcb->lock_depth = 0;

        //set space to restore trapframe
        pcb->sf = (struct p_process_table *)(pcb->p_stack + STACK_SIZE) - 1;
        //set initial registers
        pcb->sf->p_trap.ds= USEL(SEG_UDATA);
        pcb->sf->p_trap.es= USEL(SEG_UDATA);
        pcb->sf->p_trap.eax = 0;
        pcb->sf->p_trap.ebx = 0;
        pcb->sf->p_trap.ecx = 0;
        pcb->sf->p_trap.edx = 0;
        pcb->sf->p_trap.esi = 0;
        pcb->sf->p_trap.edi = 0;
        pcb->sf->p_trap.xxx = 0;
        pcb->sf->p_trap.irq = 0;

        pcb->sf->ss = USEL(SEG_UDATA);
        pcb->sf->eflags = 0x202;
        pcb->sf->esp = 0xc00000;
        pcb->sf->cs = USEL(SEG_UCODE);
        pcb->sf->eip = (uint32_t)entry;

        pcb->time_count = time_chips;
        pcb->sleep_time = 0;

        pcb->state = READY;
```

将所用的 pcb 中的通用寄存器以及栈帧初始化（注意 eflags 要初始化成 0x202（打开 if 位以便检测时钟中断的到来），将 eip 赋值成为 elf->entry，以便函数在中断返回时可以成功跳入用户进程代码处；

在等待时钟中断时，cpu 会接受到第一个时钟中断，然后判断其状态，如果是 ready，那么可以判断其时间片是否到期，如果该进程时间片到期，则可以进行调度算法；

```
tmp = current->state_list.next;
if(tmp == &readyq_h)
{
        tmp = tmp->next;
}
//search next pcb
int i;
for(i = 0; PCBs[i].state_list.next != tmp->next && PCBs[i].state_list.prev != tmp->prev; i++);
PCB *next = (void *)&(PCBs[i]);
//fresh new pcb time count
if(current->pid != 0)
{
        current->time_count = time_chips;
}
//save current pcb
struct p_process_table *frame = current->sf;
current->sf = (struct p_process_table *)(current->p_stack + STACK_SIZE) - 1;
current->sf->p_trap.ds = frame->p_trap.ds;
current->sf->p_trap.es = frame->p_trap.es;
current->sf->p_trap.eax = frame->p_trap.eax;
current->sf->p_trap.ebx = frame->p_trap.ebx;
current->sf->p_trap.ecx = frame->p_trap.ecx;
current->sf->p_trap.edx = frame->p_trap.edx;
current->sf->p_trap.esi = frame->p_trap.esi;
current->sf->p_trap.edi = frame->p_trap.edi;
current->sf->p_trap.xxx = frame->p_trap.xxx;
current->sf->p_trap.irq = frame->p_trap.irq;
current->sf->ss = frame->ss;
current->sf->esp = frame->esp;
current->sf->eip = frame->eip;
current->sf->eflags = frame->eflags;
current->sf->cs = frame->cs;
```

如果需要进行调度，则可以直接将 current 当前的状态保存到当前的 pcb 中，然后将 current 赋值更新成为就绪队列的下一个 pcb；

```
if(current->state == READY)
        current->sf = (void *)tf;               //save old trapframe
switch(tf->irq) {
        case 0x80: do_syscall(tf);break;
        case 13:assert(0);break;
        case 1000:
        {
                change_state();
                if(current->state == READY && current->time_count > 0)
                {
                        current->time_count--;
                        assert(current->time_count >= 0);       //time count should not less than 0
                }
                else
                {
                        assert(current->time_count == 0);
                        schedule();
                        if(current->pid != 0)
                                //schedule has run once , should decrease one on time count
                                current->time_count--;
                }
        }break;
```

判断当前是否需要调度；

```
asm_do_irq:
        pushal
        pushl %ds
        pushl %es
        pushl %esp
        call irq_handle

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

        popl %es
        popl %ds
        popal
        addl $4, %esp
        iret
```

更改完 current 指针后，就可以将 current 中的陷阱帧结构与原来的 tf 结构进行替换，这样原来的进程就会通过 iret 指令跳转到下一个进程的现场，并且执行下一个进程的代码；这样就完成了基本的进程切换；

Fork（）：

Fork 函数本身并不改变 current 的值，所以 fork 函数执行过后进程没有切换；所以 forkj 函数仅仅是创建了一个新的 pcb 以便调用；所以该过程与原来的创建 pcb 的过程相似，所以只是赋值以便后续调用；

```
pcb->sf = (struct p_process_table *)(pcb->p_stack + STACK_SIZE) - 1;
//copy father to child
//general registers are copied to new pcb;
pcb->sf->p_trap.ds = current->sf->p_trap.ds;
pcb->sf->p_trap.es = current->sf->p_trap.es;
pcb->sf->p_trap.eax = pcb->pid;
pcb->sf->p_trap.ebx = current->sf->p_trap.ebx;
pcb->sf->p_trap.ecx = current->sf->p_trap.ecx;
pcb->sf->p_trap.edx = current->sf->p_trap.edx;
pcb->sf->p_trap.esi = current->sf->p_trap.esi;
pcb->sf->p_trap.edi = current->sf->p_trap.edi;
pcb->sf->p_trap.xxx = current->sf->p_trap.xxx;
pcb->sf->p_trap.irq = current->sf->p_trap.irq;

pcb->sf->ss = current->sf->ss;
pcb->sf->eflags = current->sf->eflags;
pcb->sf->cs = current->sf->cs;
pcb->sf->eip = current->sf->eip;

//copy stack(assert user whole stack size less than 4096)
pcb->sf->esp = current->sf->esp - STACK_SIZE;
uint8_t *father = (void *)(current->sf->esp + USER_SS_BASE);
uint8_t *son = (void *)(pcb->sf->esp + USER_SS_BASE);
int i;
for(i = 0; i < STACK_SIZE; i++)
        son[i] = father[i];
```

注意 fork 函数应该复制一个与原来进程一模一样的 pcb，所以除了返回值 eax 之外，所有的

pcb 值都应该与原来的 pcb 一样，而且注意要将用户进程的栈帧复制到另外一个地址中；

Sleep（）：
Sleep 函数是用来对当前进程进行阻塞的，调用 sleep 函数后，程序进入中断，并且在中断当中停留一定时间后才可以继续执行，注意这个时候 cpu 是被占用的，不可以进行其他进程的切换和抢占；
（sleep 函数进入中断后不到一点的时间是不能返回的）

```
void sys_sleep(uint32_t time) {
        lock();
        if (current->state == READY) {
                current->state = SLEEP;
                current->sleep_time = time * 100;
                list_del(&(current->state_list));
                list_add(&(current->state_list),&blockq_h);
        }
        unlock();
        while(1)
                wait_for_interrupt();
}
```

这里利用 while（1）进行无限循环等待时钟中断，时钟中断可以在某一个时刻调用 wake_up 函数来对阻塞的 sleep 程序进行唤醒；

```
void wake_up() {
        lock();
        if(current->state == SLEEP) {
                current->state = READY;
                list_del(&(current->state_list));
                list_add(&(current->state_list),&readyq_h);
        }
        unlock();
}
```

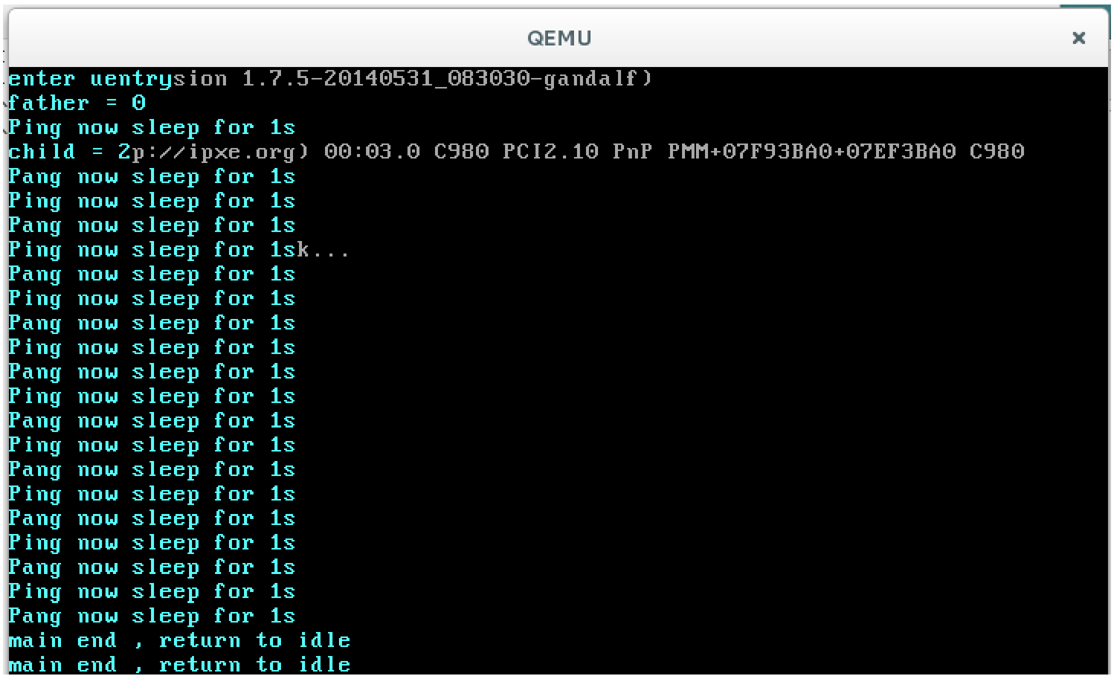
在 sleep 和 wake_up 函数进行时应该关闭外部中断的响应；

```
void sys_exit(int status) {
        lock();
        if(current->state == READY) {
                //assert(current->time_count != 0);
                PCB *free_pcb = current;

                struct list_head *tmp;
                tmp = current->state_list.next;
                //search next pcb
                int i;
                for(i = 0; PCBs[i].state_list.next != tmp->next && PCBs[i].state_list.prev != tmp->prev; i++);
                current = (void *)&(PCBs[i]);

                free_pcb->state = FREE;
                list_del(&(free_pcb->state_list));
                list_add(&(free_pcb->state_list),&freeq_h);
        }
        unlock();
}
```

Exit 函数与此类似，当前进程如果调用 exit 函数，则将当前进程的 pcb 从就绪队列中删除，并且将 current 指针向后偏移，使其进行下一个程序的执行和调度；

3. 实验结果：

当每个进程时间片为 1，循环 10 次，每次停留 1s 时：



此时可以看到 ping 和 pang 循环出现，也就是父子进程循环进行

4. 实验疑问：

对于 tss 中的 esp0 决定了每个进程中断时的陷阱帧的位置，那么如果不会进行嵌套中断时，是否不需要进行 esp0 的切换，由于每个 pcb 都有专用的内核栈，所以是否只要将中断时的陷阱帧存放到 pcb 中即可？这样会不会出现问题；

对于段寄存器，是否需要 ldt，在实验中，可以通过固定栈帧大小来确定拷贝栈帧的位置，而实际在 linux 中，我们需要 ldt 表来进行不同进程的地址切换，所以在本次试验中是否也应该通过 ldt 表来进行加载？