Project2 Non-Preemptive Kernel 设计文档

中国科学院大学
李静逸
2017.10.18

1. Context Switching 设计流程

（1）PCB 包含的信息

PCB 包含 task 的状态、类型（进程 or 线程）、上下文信息。

```
typedef struct pcb {
        /* need student add */
        process_state state;
        process_type type;
        context* context;
} pcb_t;
```

（2）如何启动第一个 task

在 kernel.c 中进行了所有 task 的初始化，然后调用 scheduler 函数，将 ready_queue 队尾的 pcb 指针弹出给 current_running，并加载第一个 task，跳到 task 的起始地址，启动了第一个 task。

（3）scheduler 的调用和执行流程

线程：do_yield(scheduler.c)->scheduler_entry(entry_mips.s)->scheduler(scheduler.c)
do_exit(scheduler.c)->scheduler_entry(entry_mips.s)->scheduler(scheduler.c)

do_yield()函数将保存正在执行的 task 的上下文，然后将正在执行的 task 压入就绪队列并改变该 task 的状态，调用 scheduler_entry()。scheduler_entry()一开始就跳到 scheduler 函数的起始地址，然后执行 scheduler()，将就绪队列（如果非空的话）队头的元素赋给 current_running，在返回到 scheduler_entry(), 加载马上要执行的 task，即把 pcb.context 里存的值赋给相应的寄存器。下图为相应部分的代码。

线程退出也是相似的，只是在 do_exit()函数里没有把 current_running 压入就绪队列，而是直接把 current_running 状态变为 PROCESS_EXITED，也就是说，以后都不会再执行这个 task 了。

```
void do_yield(void)
{
        save_pcb();
        /* push the qurrently running process on ready queue */
        /* need student add */
        current_running->state=PROCESS_READY;
        queue_push(ready_queue, current_running);
        // call scheduler_entry to start next task
        scheduler_entry();

        // should never reach here
        ASSERT(0);
}

void do_exit(void)
{
        /* need student add */
        current_running->state=PROCESS_EXITED;
        scheduler_entry();
}
```

```
scheduler_entry:
        # call scheduler, which will set new current process
        # need student add
        jal scheduler
        nop

        la t0, current_running  #obtain the address of current_running
        lw t0, (t0)             #get the start address of pcb
        lw t0, 8(t0)            #get the start address of struct context
        lw sp, (t0)             #start recover registers
        lw ra, 4(t0)
        lw s0, 8(t0)
        lw s1, 12(t0)
        lw s2, 16(t0)
        lw s3, 20(t0)
        lw s4, 24(t0)
        lw s5, 28(t0)
        lw s6, 32(t0)
        lw s7, 36(t0)
        lw s8, 40(t0)

        jr ra
        nop
```

```
void scheduler(void)
{

        ++scheduler_count;
        // pop new pcb off ready queue
        /* need student add */
        if(ready_queue->isEmpty)
                current_running=NULL;
        else{
                current_running=queue_pop(ready_queue);
                current_running->state=PROCESS_RUNNING;
        }

}
```

进程：yield(syslib.S)->kernel_entry(entry.S)->do_yield(scheduler.c)

exit (suslib.S)-> kernel_entry(entry.S)->do_yield(scheduler.c)

由于进程在用户空间（其实都在内核，但我们模拟了一下），所以要通过系统调用来获得 scheduler 函数，因此现在 yield()和 exit()里用一个 SYSCALL()系统调用，SYSCALL()的目的是跳到 kernel_entry()的起始地址，然后通过 a0 寄存器里的值是 0 还是 1 选择是调用 do_yield()还是 do_exit()函数，然后就和线程 scheduler 调用和执行是一样的了。下面为系统调用部分代码。

```
#define SYSCALL(i) \
        li a0,i; \
        li $8,0xa0800470; \
        jal $8; \
        nop
        nop

.globl yield
.globl exit

yield :
        addiu sp,sp,-24
        sw ra,0(sp)         #memory[sp]<-GPR[ra]
        sw a0,8(sp)         #memory[8+sp]<-GPR[a0]
        sw a1,16(sp)        #memory[16+sp]<-GPR[a1]
        SYSCALL(0)
        lw a1,16(sp)
        lw a0,8(sp)
        lw ra,0(sp)
        addiu sp,sp,24
        jr ra
        nop
```

```
exit :
        addiu sp,sp,-24
        sw ra,0(sp)
        sw a0,8(sp)
        sw a1,16(sp)
        SYSCALL(1)
        lw a1,16(sp)
        lw a0,8(sp)
        lw ra,0(sp)
        addiu sp,sp,24
        jr ra
        nop
```

```
kernel_entry:
        addiu sp, sp, -24   #stack pointer
        sw ra, 0(sp)        #memory[GPR[sp]]<--GPR[ra]
        bnez $4, 1f         #if GPR[4]!=0 jump to 1f
        nop

        jal do_yield
        nop
        beqz $0,2f          #if GPR[0]==0 jump to 2f
        nop

1:
        jal do_exit
        nop
2:
        lw ra, 0(sp)        #GPR[ra]<--memory[GPR[sp]]
        addiu sp, sp, 24
        jr ra               #PC<--GPR[ra]
        nop
```

（4）context switching 是如何保存 PCB，使得进程再切换回来后能正常运行

如下图，首先得到 task 的 pcb 的起始地址，然后得到存 context 的起始地址，最后把 sp,ra,s0-s8 寄存器里的值存到存 context 的地方。我给每一个 task 都、分配了一块空

间专门存 context 内容，pcb 结构里有一个域是指向这块地址空间的指针。

```
save_pcb:
        # save the pcb of the currently running process
        # need student add
        la t0, current_running  #get the address of current_ru
        lw t0, (t0)             #get the start address of pcb
        lw t0, 8(t0)            #get the start address of struc
        lw t1,16(sp)            #get ra
        addiu sp,sp,24
        sw sp, (t0)             #store sp
        addiu sp,sp,-24
        sw t1, 4(t0)            #store ra
        sw s0, 8(t0)
        sw s1, 12(t0)
        sw s2, 16(t0)
        sw s3, 20(t0)
        sw s4, 24(t0)
        sw s5, 28(t0)
        sw s6, 32(t0)
        sw s7, 36(t0)
        sw s8, 40(t0)
        jr ra
        nop
```

```
typedef struct pcb {
        /* need student add */
        process_state state;
        process_type type;
        context* context;
} pcb_t;
```

```
typedef struct context{
        uint32_t sp;
        uint32_t ra;
        uint32_t s0;
        uint32_t s1;
        uint32_t s2;
        uint32_t s3;
        uint32_t s4;
        uint32_t s5;
        uint32_t s6;
        uint32_t s7;
        uint32_t s8;
} context;
```

（5）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

问题 1：bootblock 执行不正确

解决 1：没有改跳转到 kernel_main 的地址。

问题 2：bootblock 应读取的 kernel 的大小没有正确更改。

解决 2：我最开始是通过直接更改 lw $6, length 这条指令里的立即数部分来实现，但是后来发现直接改指令很容易出错，当立即数大于 0xffff 时，这条指令会分成两条指令，因此需要同时更改两个不同地址的指令，尝试了几次都失败了，最后在 bootblock.s 里设了一个全局变量 os_size，os_size 的地址是固定的，通过更改这个地址里的值来更改 bootblock 应该读取的 kernel 的长度。代码见下图：

```
main:
        # check the offset of main
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

        li $4, 0xa0800200  #start
        li $5, 0x200      #offset
        #li $6, 0x50000
        la $8, os_size
        lw $6, ($8)         #length

        jal 0x8007b1a8     #call read function
        jal 0xa08002bc
```

2. Context Switching 开销测量设计流程

（1）如何测量线程切换到线程时的开销

首先在 thread4()里通过 get_timer()函数得到一个 cycle 数，然后通过 do_yield()函数调用下一个 thread5(),再通过 get_timer()函数得到另一个 cycle 数，两个 cycle 数相减，再把结果除以 MHZ 就得到了线程切换到线程的时间。

```
void thread4(void)
{
        clear_screen(0,0,3,3);
        sticks_th4=get_timer();
        do_yield();
        do_exit();
}

void thread5(void)
{
        if(count==0){
                sticks_th5=get_timer();
                time1=(sticks_th5-sticks_th4)/MHZ;
                print_str(1,3,"Context switch time(ms) from thread4 to thread 5 is: ");
                printint(4,4,time1);
                ++count;
                do_yield();
        }
        else if(count){
                sticks_th5=get_timer();
                do_yield();
                sticks_pro3=get_timer();
                time2=(sticks_pro3-sticks_th5)/MHZ/2;
                print_str(1,6,"Context switch time(ms) from thread5 ro process3 is: ");
                printint(4,7,time2);
                do_yield();
        }
}
```

（2）如何测量线程切换到进程时的开销

由于 th3.c 和 process3.c 之间无法用同一个变量，所以计算从 thread5 到 process3 再回到 thread5 的时间，除以 2，就得到了线程切换到进程的开销。

（3）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

这一阶段的 bootblock 和上一阶段相同，所以没有遇到什么问题，就是全局变量 os_size 的地址偶尔会发生变化，需要在 createiamge.c 中做调整。

3. Mutual lock 设计流程

（1）spin-lock 和 mutual lock 的区别

Spin-lock：程序到临界区之前检查锁是否 LOCKED，如果是，则 CPU 一直等待知道锁变为 LOCKED，在这期间 CPU 不会做任何其他事情。

Mutual lock：程序到邻接区之前检查锁是否被 LOCKED，如果是，CPU 不会等待解锁，而是把当前进程阻塞，然后去执行其他的进程。

（2）能获取到锁和获取不到锁时各自的处理流程

如果能获取到锁，CPU 访问临界区。

如果不能获得锁，阻塞当前进程，去执行就绪队列里的其他进程。

```
void lock_acquire(lock_t * l)
{
        if (SPIN) {
                while (LOCKED == l->status)
                {
                        do_yield();
                }
                l->status = LOCKED;
        } else {
                /* need student add */
                while(l->status==LOCKED)
                        block();
                l->status = LOCKED;
        }
}
```

（3）被阻塞的 task 何时再次执行

当锁被释放后，task 被压入就绪队列队尾，然后随着 CPU 的调度 task 逐渐移到就绪队列队头，然后把该 task 的 pcb 指针赋给 current_running，加载该 task，该 task 再次执行。

（4）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

最开始测试 all tasks 的时候 thread3 在 198 的时候 failed，后来经过仔细的检查发现在获取锁的时候如果锁已经被占了，需要把正在执行的进程阻塞，由于我用了 if-else 语句，阻塞完我忘记把锁的状态赋为 LOCKED 了，改完就 pass 了。

4. 关键函数功能

部分关键代码我已经在前面的报告里贴出来了并配有解释。

下面贴出另外一部分关键代码。

PCB 初始化和调度器初始化：

```
void _stat(void){

        /* some scheduler queue initialize */
        /* need student add */
        ready_queue=&R_queue;
        ready_queue->pcbs=ready_arr;
        queue_init(ready_queue);
        ready_queue->capacity=NUM_TASKS;

        blocked_queue=&B_queue;
        blocked_queue->pcbs=blocked_arr;
        queue_init(blocked_queue);
        blocked_queue->capacity=NUM_TASKS;

        clear_screen(0, 0, 30, 24);

        /* Initialize the PCBs and the ready queue */
        /* need student add */
        int i;
        for(i=0;i<NUM_TASKS;++i){
                pcbs[i].state=PROCESS_READY;
                pcbs[i].type=task[i]->task_type;
                pcbs[i].context=&(context_save[i]);
                pcbs[i].context->ra=task[i]->entry_point;
                pcbs[i].context->sp=STACK_MAX-i*STACK_SIZE;
                queue_push(ready_queue,&(pcbs[i]));
        };
        /*Schedule the first task */
        scheduler_count = 0;
        scheduler_entry();
```

Scheduler_entry:

```
scheduler_entry:
        # call scheduler, which will set new current process
        # need student add
        jal scheduler
        nop

        la t0, current_running  #obtain the address of current_running
        lw t0, (t0)             #get the start address of pcb
        lw t0, 8(t0)            #get the start address of struct context
        lw sp, (t0)             #start recover registers
        lw ra, 4(t0)
        lw s0, 8(t0)
        lw s1, 12(t0)
        lw s2, 16(t0)
        lw s3, 20(t0)
        lw s4, 24(t0)
        lw s5, 28(t0)
        lw s6, 32(t0)
        lw s7, 36(t0)
        lw s8, 40(t0)

        jr ra
        nop
```

互斥锁：

```
enum {
        SPIN = FALSE,  //original: SPIN=TRUE  changed: SPIN
};

void lock_init(lock_t * l)
{
        if (SPIN) {
                l->status = UNLOCKED;
        } else {
                /* need student add */
                l->status = UNLOCKED;
        }
}
```

```
void lock_acquire(lock_t * l)
{
        if (SPIN) {
                while (LOCKED == l->status)
                {
                        do_yield();
                }
                l->status = LOCKED;
        } else {
                /* need student add */
                if (l->status == LOCKED)
                        block();
                else
                        l->status = LOCKED;
        }
}
```

```
void lock_release(lock_t * l)
{
        if (SPIN) {
                l->status = UNLOCKED;
        } else {
                /* need student add */
                l->status = UNLOCKED;
                unblock();
        }
}
```

阻塞：

```
void block(void)
{
        save_pcb();
        /* need student add */
        current_running->state=PROCESS_BLOCKED;
        queue_push(blocked_queue, current_running);

        scheduler_entry();
        // should never reach here
        ASSERT(0);
}

int unblock(void)
{
        /* need student add */
        if(blocked_queue->isEmpty)
                return 0;
        pcb_t *pcb_temp;
        while(!(blocked_queue->isEmpty)){
                pcb_temp=queue_pop(blocked_queue);
                pcb_temp->state=PROCESS_READY;
                queue_push(ready_queue,pcb_temp);
        }
        return 1;
}
```

参考文献

[1] Andrew S. Tanenbaum《现代操作系统》（原书第 3 版）

■