Project2 Non-Preemptive Kernel 设计文档

中国科学院大学
[姓名] 袁峥
[学号] 2015K8009929008
[日期] 2017.10.13

1. Context Switching 设计流程

（1）PCB 包含的信息

PCB 每个表项中包含进线程切换时 31 个寄存器的值，以及该进程的 pid，该进程的状态（阻塞、就绪、运行、退出）、该进程所分配的栈顶地址、该进程所分配的栈大小和该任务类型（进程还是线程）。特别指出，进程切换时，原进程的运行下一条指令地址被直接保存在 31 号寄存器，在切换时直接使用寄存器跳转即可。

```
typedef struct pcb {
    /* need student add */
    context context;
    uint32_t pid;
    process_state state;
    uint32_t stack_top;
    uint32_t stack_size;
    uint8_t task_type;
} pcb_t;
```

（2）如何启动第一个 task

首先将进行初始化，建好一个就绪进程队列和阻塞进程队列，并初始化每个 task 的 PCB 表项，然后逐个压入就绪进程队列。初始化完毕后，启动第一个 task 任务时，只需调用 scheduler_entry 函数，取出就绪进程队列中的一个 task，然后装载寄存器并切换到这个 task 即可。

（3）scheduler 的调用和执行流程

scheduler 函数是被 entry_mips.S 中的 scheduler_entry 函数所调用的，scheduler 函数首先弹出一个就绪进程队列中的 task 并赋值到 current_running，并将该 task 的状态由就绪该为运行。接着 scheduler 函数结束并回到 scheduler_entry 函数，该函数将 current_running 所指 PCB 表项中的保存的寄存器的值全部更新到寄存器中，然后使用 jr ra 指令跳转到该 task 上次切换退出时所运行到的指令。

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

保存 PCB 的过程在 entry_mips.S 中的 save_pcb 函数，该函数将 task 切换退出时 31 个寄存器的值保存到该 task 所在的 PCB 表项中，以便下次切换回来后能正常运行。其中有两个寄存器的值需要特别说明，由于进行保存 PCB 时会要调用 save_pcb 函数，函数调用时本身就会压栈并更改 31 号寄存器的值，因此在保存原 task 切换时寄存器的值时，sp 和 ra 寄存器的值不能直接保存。通过查看反汇编的代码，发现在运行 save_pcb 函数前 sp 指针先被减去了 24，因此在保存真正 task 切换时的 sp 时，需要将 save_pcb 函数中的 sp 加上 24，而 task 切换时真正需要保存的 ra 寄存器的值在进入 save_pcb 已经被压入栈中，通过反汇编可以看

到偏移是 20(sp)，因此需要将 20(sp)中的数据存入 PCB 中保存的 ra 寄存器。

（5）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

该部分遇到的问题首先是初始化两个队列时，没有仔细查看队列定义中的元素，所以没有初始化 capacity 项，导致在后面队列操作时出现模 0，从而出现错误。

其次是在 save_pcb 函数中保存真正的 ra 寄存器的值时，从老师课件中看到的是 16(sp)，编写代码的时候就先这样写了，后来上板运行时发生错误，发现 PC 跳转到了 0x800XXXXXX，而这个 PC 地址在这次的实验中是不可能出现的，因此我猜测可能是保存 ra 时出现了错误，然后检查了反汇编代码，发现在自己的程序中被压入栈的 ra 值的偏移是 20(sp)，修改后最终解决了这个错误。

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

该任务中 task 的执行顺序是 thread4、thread5、process3 三者不断循环，因此测量线程间切换的开销，只需在切换离开 thread4 前进行计时，然后切换进入 thread5 时，再次计时，并将两者作差就是线程间切换的开销。

（2）如何测量线程切换到进程时的开销

考虑到 thread4 和 thread5 是在一个文件中，在文件中使用全局变量较为方便，因此在 process3 中没有做任何事，一进去就切换离开，而测量线程和进程间切换的开销时，在切换离开 thread5 前进行计时，然后在切换进入 thread4 时再次计时，两者之间的时间包含从 thread5 切换到 process3，再从 process3 切换到 thread4，也就是两次进程与线程间的切换，因此将该时间除 2 就是所要求的线程与进程间切换的开销。

（3）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

该任务在实现输出显示的时候发现 print_location 和 printint 函数中输入的行列坐标和实际是相反的，因此在观察结果后发现了这个问题并进行了修改。

3. Mutual lock 设计流程

（1）spin-lock 和 mutual lock 的区别

spin-lock 在 lock_acquire 时首先先无限循环检测当前锁是否被占用，如果获取锁失败那么该线程切换离开，进入就绪队列队尾，下次切换进入后继续检测锁是否被占用，一直如此循环。如果获取成功，那么修改锁的状态为被占用，然后退出 lock_acquire 函数。

mutual lock 在 lock acquire 时也是先检测当前锁是否被占用，如果获取锁失败，那么将此线程放入阻塞队列，将该线程阻塞，等到其从阻塞队列中回到就绪队列并切换到时，再次检测是否可以获得锁，如果仍旧失败则继续放入阻塞队列并不断如此循环，直到获取成功，修改锁的状态为被占用，然后退出。

两者的区别简言之，前者如果获取锁失败会一直不断检测，直到获取成功。而后者如果获取失败，会先进入阻塞队列，等到再次切换到该 task 时再次检测。

（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，那么将锁的状态改为被占用，然后退出 lock_acquire 函数。如果取不

到锁，那么调用 block 函数，将当前的 task 进行阻塞并存入阻塞队列，等待当前使用锁的 task 在 lock_release 时从阻塞队列中取出放入就绪队列，并等待再次切换到，然后再次检测是否可以获取锁，如果还是不能，那么再次进入阻塞队列，并不断循环，直到成功获取锁。

（3）被阻塞的 task 何时再次执行

当前占用锁的 task 进入 lock_release 函数释放锁时，检查阻塞队列，如果其中有 task 被阻塞，那么在其中取一个出来，将其状态从阻塞修改为就绪，然后将其压入就绪队列。在就绪队列中等待，直到切换到该 task。

（4）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

在关于释放锁时的处理上，一开始的处理方法时，释放锁后同时也将该 task 切换掉，换新的 task 运行，但是后来发现，如果采用这种设计，那么需要在 unblock 函数中调用 save_pcb 函数，而在调用前会先准备 32 的栈空间来保存寄存器，这与 yield 和 block 中调用 save_pcb 函数前准备 24 的栈空间不同，这也会导致 save_pcb 函数中保存 sp 和 ra 寄存器的方法不同。因此后来放弃了该设计，而是释放锁后继续运行当前 task。

4. 关键函数功能

1、kernel.c

```
void _stat(void){

    /* some scheduler queue initialize */
    /* need student add */
    ready_queue=&ready_queues;
    blocked_queue=&blocked_queues;
    queue_init(ready_queue);
    ready_queue->pcbs=&ready_arr;
    ready_queue->capacity=NUM_TASKS;
    queue_init(blocked_queue);
    blocked_queue->pcbs=&blocked_arr;
    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++) {
        pcb_table[i].pid=i;
        pcb_table[i].stack_top=STACK_MIN+i*STACK_SIZE;
        pcb_table[i].stack_size=STACK_SIZE;
        pcb_table[i].state=PROCESS_READY;
        pcb_table[i].task_type=task[i]->task_type;
        pcb_table[i].context.reg[29]=pcb_table[i].stack_top;
        pcb_table[i].context.reg[31]=task[i]->entry_point;
        queue_push(ready_queue,pcb_table+i);
    }

    /*Schedule the first task */
    scheduler_count = 0;
    scheduler_entry();

    /*We shouldn't ever get here */
    ASSERT(0);
} « end _stat »
```

这里主要完成了阻塞和就绪队列的初始化，并根据不同任务，把各自所运行的 task 的 PCB 表项初始化好，然后分别压入就绪队列，最后调用 scheduler_entry 获取第一个 task 运行。

2、entry_mips.S

```
save_pcb:
    # save the pcb of the c
    # need student add
    sw t0,(sp)
    la t0,current_running
    lw t0,(t0)
    sw AT,1*4(t0)
    sw v0,2*4(t0)
    sw v1,3*4(t0)
    sw a0,4*4(t0)
    sw a1,5*4(t0)
    sw a2,6*4(t0)
    sw a3,7*4(t0)
    //sw t0,8*4(t0)
    sw t1,9*4(t0)
    sw t2,10*4(t0)
    sw t3,11*4(t0)
    sw t4,12*4(t0)
    sw t5,13*4(t0)
    sw t6,14*4(t0)
    sw t7,15*4(t0)
    sw s0,16*4(t0)
    sw s1,17*4(t0)
    sw s2,18*4(t0)
    sw s3,19*4(t0)
    sw s4,20*4(t0)
    sw s5,21*4(t0)
    sw s6,22*4(t0)
    sw s7,23*4(t0)
    sw t8,24*4(t0)
    sw t9,25*4(t0)
    sw k0,26*4(t0)
    sw k1,27*4(t0)
    sw gp,28*4(t0)
    //sw sp,29*4(t0)
    sw fp,30*4(t0)
    //sw ra,31*4(t0)
    lw t1,20(sp)
    sw t1,31*4(t0)
    addiu sp,24
    sw sp,29*4(t0)
    addiu sp,-24
    la t1,current_running
    lw t1,(t1)
    lw t0,(sp)
    sw t0,8*4(t1)
    jr ra
```

```
scheduler_entry:
    # call scheduler, whic
    # need student add
    jal scheduler
    la ra,current_running
    lw ra,(ra)
    lw AT,1*4(ra)
    lw v0,2*4(ra)
    lw v1,3*4(ra)
    lw a0,4*4(ra)
    lw a1,5*4(ra)
    lw a2,6*4(ra)
    lw a3,7*4(ra)
    lw t0,8*4(ra)
    lw t1,9*4(ra)
    lw t2,10*4(ra)
    lw t3,11*4(ra)
    lw t4,12*4(ra)
    lw t5,13*4(ra)
    lw t6,14*4(ra)
    lw t7,15*4(ra)
    lw s0,16*4(ra)
    lw s1,17*4(ra)
    lw s2,18*4(ra)
    lw s3,19*4(ra)
    lw s4,20*4(ra)
    lw s5,21*4(ra)
    lw s6,22*4(ra)
    lw s7,23*4(ra)
    lw t8,24*4(ra)
    lw t9,25*4(ra)
    lw k0,26*4(ra)
    lw k1,27*4(ra)
    lw gp,28*4(ra)
    lw sp,29*4(ra)
    lw fp,30*4(ra)
    lw ra,31*4(ra)
    jr ra
```

该文件中最重要的两个函数是 scheduler_entry 和 save_pcb。

scheduler_entry 先调用 scheduler 获取新的就绪 task，然后将 PCB 表中该 task 上次切换离开时保存的寄存器值全部恢复到寄存器中，然后跳转到上次切换时所运行到的地址（即 31 号寄存器所保存内容）继续运行。

save_pcb 保存进程切换时的寄存器值到对应的 PCB 表项中，其中 sp 和 ra 寄存器的处理需要特别处理，在上面的文档中已经详细介绍。

3、scheduler.c

```
void scheduler(void)
{
    ++scheduler_count;

    // pop new pcb off ready queue
    /* need student add */
    current_running=queue_pop(ready_queue);
    (*current_running).state=PROCESS_RUNNING;
}

void do_yield(void)
{
    save_pcb();
    /* push the qurrently running process on ready queue */
    /* need student add */
    queue_push(ready_queue,current_running);
    (*current_running).state=PROCESS_READY;
    // 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 函数的功能就是从就绪队列中取出一个 task 并将其状态改为运行。

do_yield 函数首先保存当前 task 的 PCB 表项，然后将当前 task 压入就绪队列队尾，并将其状态从运行改为就绪，接着再调用 scheduler_entry 函数获取新的 task 并运行。

do_exit 函数首先将当前 task 的状态改为退出，并调用 scheduler_entry 函数获取新的 task 并运行。

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

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

int unblock(void)
{
    /* need student add */
    pcb_t *temp_process;
    if (blocked_tasks()) {
        temp_process=queue_pop(blocked_queue);
        (*temp_process).state=PROCESS_READY;
        queue_push(ready_queue,temp_process);
    }
    return 0;
}
```

block 函数首先保存当前 task 的 PCB 表项，然后将该 task 的状态从运行改为阻塞，并放入阻塞队列，然后获取新的 task 运行。

unblock 函数首先检查阻塞队列有没有 task，如果有，那么取出其中一个 task 将其状态从阻塞改为就绪并放入就绪队列。

4、th3.c

```
uint32_t thread4_start;
uint32_t thread4_end;
uint32_t thread5_start;
uint32_t thread5_end;
void thread4(void)
{
    int i;
    for ( i = 0; i < 100; ++i){
        thread4_start=get_timer();
        thread4_start-=thread5_end;
        thread4_start = ((uint32_t) thread4_start) / (2*MHZ);     /* divide on CPU clock frequency in
                        * megahertz */

        if (i)
        {
            print_location(0, 1);
            printstr("The Time in switch between thread and process (in us): ");
            printint(60, 1, thread4_start);
        }
        thread4_end=get_timer();
        do_yield();
    }
    do_exit();
} « end thread4 »

void thread5(void)
{
    int i;
    for ( i = 0; i < 100; ++i){
        thread5_start=get_timer();
        thread5_start-=thread4_end;
        thread5_start = ((uint32_t) thread5_start) / (MHZ);     /* divide on CPU clock frequency in
                        * megahertz */

        print_location(0, 2);
        printstr("The Time in switch between thread and thread (in us): ");
        printint(60, 2, thread5_start);
        thread5_end=get_timer();
        do_yield();
    }
    do_exit();
}
```

这两个线程主要是计算线程之间以及线程与进程间切换花销，并输出结果，具体原理已在上述文档中阐述。

5、lock.c

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

    }
}
```

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

lock_acquire 函数首先循环检测是否锁可以，如果可用则更改锁的状态为被占用然后退出，否则将当前 task 阻塞。

lock_release 函数将锁的状态更改为空闲，并调用 unblock 函数将阻塞队列中的 task 取至就绪队列。