Project2 Non-Preemptive Kernel 设计文档

中国科学院大学
[王苑铮]
[2017.10.18]

1. Context Switching 设计流程

（1） PCB 包含的信息
进程或线程的 pid（可能没什么用，这个是参考 xv6 的 pcb），进程或线程的状态，几个重要寄存器的值，包括：s0~s7，sp（栈指针），ra（返回地址）

（2） 如何启动第一个 task
在完成队列初始化、pcb 初始化之后，调用 scheduler_entry(), scheduler_entry() 进入调度器函数 scheduler(), scheduler 就把第一个 task 出队给 current_running，之后返回到 scheduler_entry(),scheduler_entry ()把上下文（此处的上下文是在初始化 pcb 时填好的。其 ra 是 task1 的执行地址）切换进寄存器，再返回，就返回到了 task1 的执行地址，之后 task1 就开始执行。

（3） scheduler 的调用和执行流程
问题（2）已经回答

（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：将 s0~s7,sp,ra 寄存器的值通过 sw 指令存到当前 task 的 pcb 对应的域，即从寄存器存入内存。其中 sp、ra 的值不能直接 sw。对于 sp，在 task 调用 yeild/block 时，sp 会偏移，保存进内存的 sp 要先去掉这个偏移。对于 ra：task 的 ra 在 yeild/block 时被存入了内存，在 save_pcb 时 ra 寄存器的值是返回到 yeild/block 的执行地址而不是 task 的执行地址。想要返回到 task 的地址则要到内存中对应的位置去找。

（5） 任何在设计、开发和调试 Context Switching 时遇到的问题和解决方法
1.我的这个偏移量和课件里给的并不一致。我开始是写的 16，但无法执行，返回信息显示 ra 的值不对。objdump 了 kernel 后发现了这个偏移量不一样的问题，改掉以后这个错误就没了。

```
a0800658:   27bdffe8    addiu   sp,sp,-24
a080065c:   afbf0014    sw  ra,20(sp)
a0800660:   0c200152    jal a0800548 <save_pcb>
```

```
398 a08007e4 <do_yield>:
399 a08007e4:   27bdffe8    addiu   sp,sp,-24
400 a08007e8:   afbf0010    sw  ra,16(sp)
401 a08007ec:   0c20015a    jal a0800568 <save_pcb>
402 a08007f0:   00000000    nop
```

2.我的保存上下文是这么做的，但是上板后发现各个寄存器里的值不对。后来查了一下我的 pcb 的定义，发现在 pcb 结构体的存上下文的域前面，还有两个域：pid，state。我把着两个域移动到上下文域的后面，这个问题就改正了。

```
la t0, current_running
lw t0, 0(t0)
#move context from mem to reg
lw s0, 0(t0)
lw s1, 4(t0)
lw s2, 8(t0)
lw s3, 12(t0)
lw s4, 16(t0)
lw s5, 20(t0)
lw s6, 24(t0)
lw s7, 28(t0)
lw sp, 32(t0)
lw ra, 36(t0)
jr ra
nop
```

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

在线程的文件中设个全局变量 time，线程 1 先 time=gettimer(),之后 do_yeild()切到线程 2，线程 2 的第一句代码是 time=gettimer()-time，这样就可以算出从线程 1 切到线程 2 的时间

（2） 如何测量线程切换到进程时的开销

我觉得这个问题并不合适。从线程切到其他 task 的流程是线程->do_yeild->scheduler_entry->scheduler->scheduler_entry->下一个 task，无论下一个 task 是进程还是线程，这个时间应该没有区别。

有区别的是从进程切到其他 task 的时间。从进程切到其他 task 的流程是进程->系统调用-> do_yeild->scheduler_entry->scheduler->scheduler_entry->下一个 task，这个会比从线程切出来多一个系统调用的时间。并且无论这个 task 是进程还是线程，时间都没有区别。

也就是说，上下文切换的时间只有两个：从进程切到其他 task 的时间，从线程切到其他 task 的时间。

所以，我算了一个从进程切到进程的时间。方法是先把前两个线程都 exit 之后，在进程中 time=gettimer(),之后 yeild()。此时由于只剩下进程这一个 task，所以调度器调完后，切回来的还是它自己。这时 time=gettimer()-time 就算出了从进程切出来所需的时间。

（3） 遇到的问题和解决方法

一开始是想在线程中设个全局变量，让进程也能访问。但是线程是属于 kernel 这个进程的，另一个进程无法访问 kernel 进程里的全局变量。所以我采用了上面那个让进程自己切换到自己的方法。

3. Mutual lock 设计流程

（1） spin-lock 和 mutual lock 的区别

在一个 task1 获得锁的情况下，另一个 task2 申请锁会失败。这时如果是 spin-lock，则 task2 还会在 ready 队列中，调度器之后还有可能把 task2 再调度出来去申请这个锁。

如果是 mutual-lock，则 task2 申请锁失败时，就被放进了 block 队列，后续不会被调度器再调度出来，直到 task1 放弃锁为止，这时 task2 被从 block 队列弹出重新进入 ready 队列，后续去申请锁。

（2） 能获取到锁和获取不到锁时各自的处理流程

获取到锁：将锁状态置为 LOCK，然后拥有这个锁，之后继续执行，直到主动放弃锁为止。放弃锁时将锁的状态置为 UNLOCK

没获取到锁:如（1）所述，被放进 block 队列，直到当前有锁的 task 放弃锁，才被从 block 队列弹出放回 ready 队列

（3） 被阻塞的 task 何时再次执行

（4） 有锁的 task 放弃锁，才被从 block 队列弹出放回 ready 队列。之后 scheduler 再把它从 ready 队列弹出时，才能再次被执行

（5） 任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

这部分没出错

4. 关键函数功能

1.我的 do_yeild objdump 出来后给的偏移量和老师的 ppt 上不一样，导致 ra 的值不对。这个 bug 卡了好久。

```
a0800658:   27bdffe8    addiu   sp,sp,-24
a080065c:   afbf0014    sw  ra,20(sp)
a0800660:   0c200152    jal a0800548 <save_pcb>
```

2. kernel 中的关键部分：初始化 pcb，将调用 scheduler_entry（），用调度器调度出第一个 task。

```
    //init all pcb ,and pop into ready_queue
    int i;
    //pcb_t *pcb;
    for(i=0 ; i<NUM_TASKS ; ++i){
        //init
        //pcb = (pcb_t*)malloc(sizeof(pcb_t));
        pcb[i].pid = i;
        //resigters:
        pcb[i].reg_s0 = 0;
        pcb[i].reg_s1 = 0;
        pcb[i].reg_s2 = 0;
        pcb[i].reg_s3 = 0;
        pcb[i].reg_s4 = 0;
        pcb[i].reg_s5 = 0;
        pcb[i].reg_s6 = 0;
        pcb[i].reg_s7 = 0;

        //todo
*
        pcb[i].reg_k0 = 0;
        pcb[i].reg_k1 = 0;
/   pcb[i].reg_sp = STACK_MAX - i*STACK_SIZE; // stack start from high address.
        pcb[i].reg_ra = task[i]->entry_point;

        printstr("debug pcb:");printnum(pcb[i].pid);
        printstr("    entry:\n");printnum(pcb[i].reg_ra);
            //push
        queue_push(ready_queue,&pcb[i]);

    }
    printstr("debug before clean screen\n");
    clear_screen(0, 0, 30, 24);

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

3. scheduler_entry():进入 scheduler()调度出新的 task，恢复上下文（将新 task 的关键寄存器的值从内存的 pcb 结构体中移到寄存器中），跳转到新 task 的执行地址

```
scheduler_entry:
    # call scheduler, which will set new current process
    # need student add
    jal scheduler
    nop
    #get     the addr stored in var current_running
    la t0, current_running
    lw t0, 0(t0)
    #move context from mem to reg
    lw s0, 0(t0)
    lw s1, 4(t0)
    lw s2, 8(t0)
    lw s3, 12(t0)
    lw s4, 16(t0)
    lw s5, 20(t0)
    lw s6, 24(t0)
    lw s7, 28(t0)
    lw sp, 32(t0)
    lw ra, 36(t0)
    jr ra
    nop
```

4.保存 pcb：

```
save_pcb:
    # save the pcb of the currently running process
    # need student add

    #get the addr stored in var current_running
    la t0, current_running
    lw t0, 0(t0)
    #move context from mem to reg
    sw s0, 0(t0)
    sw s1, 4(t0)
    sw s2, 8(t0)
    sw s3, 12(t0)
    sw s4, 16(t0)
    sw s5, 20(t0)
    sw s6, 24(t0)
    sw s7, 28(t0)
    #do_yeild() and block() have a operation: addiu sp,sp,-24
    addiu sp, sp, 24
    sw sp, 32(t0)
    addiu sp,sp, -24
    lw t1, 20(sp)
    sw t1, 36(t0)
    jr ra
    nop
```

5.互斥锁的实现：

```
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 */
    while (LOCKED == l->status)
        {
            block();
        }
        l->status = LOCKED;
    }
}

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