Project2 Non-Preemptive Kernel 设计文档

中国科学院大学
李云志
2017.10.18

1. Context Switching 设计流程

（1） PCB 包含的信息
PCB 中包含的信息：上下文、ID、状态、栈顶、栈大小以及此次试验中的 task 的类型（线程与进程）

（2） 如何启动第一个 task
a) kernel 在 kernel.c 中的__stat()函数处开始执行
b) 初始化相关队列以及 PCB 表
c) 调用 scheduler_entry()函数
d) 调用 scheduler()函数；从 ready_queue 中弹出一个任务，并恢复现场
e) 第一个 task 启动

（3） scheduler()的调用和执行流程
a) scheduler()被 scheduler_entry()调用，进一步调用 queue_pop()从 ready 队列中弹出一个任务
b) 返回至 scheduler_entry()中恢复现场，从而完成切换

（4） context switching 时如何保存 PCB，使得进程再切换回来后能正常运行
a) 通过 current_running 来找到当前需保存的任务的 PCB
b) 通过 load 指令将寄存器中的值存到 PCB 中，以完成现场的保存

（5） 任何在设计、开发和调试时遇到的问题和解决方法
a) save_pcb()函数的编写中需要的问题是栈指针的恢复以及返回值的寻找。在此次设计中，save_pcb()被 do_yield()函数调用，通过查看反汇编代码可知在 save_pcb()执行之前，除了 sp 和 ra 寄存器以外，其他寄存器的值并未发生变化，同时我们需要的 ra 值被存入了栈中，所以需要手动寻找相关参数并进行保存

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

（2） 如何测量线程切换到进程时的开销
注：（1）、（2）问并做一题回答
此部分的任务切换流程如下：thread4 => thread5 => process =>thread4……循环执行。而两个线程声明定义在同一个 c 文件中，所以我们利用全局变量来进行相关时间开销的测量
a) 从 thread4 记录时间到 thread5 结束：为线程切换到线程的时间
b) 从 thread5 记录时间到 thread4 结束，并且 process 中只进行切换工作：为 2 倍的线程切进程时间

c) 循环 50 次后取平均值

（3） 任何在设计、开发和调试时遇到的问题和解决方法

此部分只是在打印相关信息上需要进行代码阅读及学习，并无其他问题

3. Mutual lock 设计流程

（1） spin-lock 和 mutual lock 的区别

自旋锁与互斥锁在此次实验中的区别为：

自旋锁循环查询锁的状态，如果获取到则继续执行，否则放入 ready_queue 的队尾

互斥锁也循环查询锁的状态，如果获取到则继续执行，否则放入 block_queue，在发生一次 lock_release()操作时，从 block_queue 中弹出一个任务放到 ready_queue 队尾

（2） 能获取到锁和获取不到锁时各自的处理流程

获取到锁则继续执行

获取不到锁则进入 block_queue 等待 lock_release()后的 unblock()弹出一个 task，被弹出后继续查询锁的状态

（3） 被阻塞的 task 何时再次执行

lock_release()函数中的 unblock()会从 block_queue()中弹出一个被阻塞的任务，并将其加入 ready_queue()中，再次 yield 到它后会再次执行

（4） 任何在设计、开发和调试时遇到的问题和解决方法

主要问题就是互斥锁相关函数的设计与编写，执行顺序为：

lock_acquire() => block() => save_pcb() => queue_push() => scheduler_entry() => scheduler() => new task

lock_release() => unblock() => queue_pop() => queue_push()

4. 关键函数功能

请列出你觉得重要的代码片段、函数或模块（可以是开发的重要功能，也可以是调试时遇到问题的片段/函数/模块）

1) PCB

```
typedef struct {
    uint32_t reg[32];
}reg;

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

2) save_pcb()

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

3) lock_acquire()与 lock_release()

```
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;
    }
}
```

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

        unblock();
        l->status = UNLOCKED;
    }
}
```

4) block()与 unblock()

```
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;
    if(blocked_tasks())
    {
        temp = queue_pop(blocked_queue);
        (*temp).state = PROCESS_READY;
        queue_push(ready_queue, temp);
    }
}
```