Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学
张磊
2019 年 9 月 25 日

1. 任务启动与 Context Switch 设计流程

（1）PCB 中包含的信息：

```
typedef struct pcb
{
    /* register context */
    regs_context_t kernel_context;
    regs_context_t user_context;

    uint32_t kernel_stack_top;
    uint32_t user_stack_top;

    /* previous, next pointer */
    void *prev;
    void *next;

    /* process id */
    pid_t pid;

    /* kernel/user thread/process */
    task_type_t type;

    /* BLOCK | READY | RUNNING */
    task_status_t status;

    /* cursor position */
    int cursor_x;
    int cursor_y;

} pcb_t;
```

参照 PCB 结构体定义，其中包含内核态和用户态的上下文，内核态和用户态的栈顶，用于队列操作的指针，进程 id，进程类型，进程状态，以及飞机的坐标信息；

```c
typedef struct regs_context
{
    /* Saved main processor registers.*/
    /* 32 * 4B = 128B */
    uint32_t regs[32];

    /* Saved special registers. */
    /* 7 * 4B = 28B */
    uint32_t cp0_status;
    uint32_t hi;
    uint32_t lo;
    uint32_t cp0_badvaddr;
    uint32_t cp0_cause;
    uint32_t cp0_epc;
    uint32_t pc;

} regs_context_t; /* 128 + 28 = 156B */
```

参照 regs_context_t 的定义，内核态和用户态的上下文中包含有 32 个通用寄存器和 7 个特殊寄存器的相关信息；

```c
typedef enum {
    TASK_BLOCKED,
    TASK_RUNNING,
    TASK_READY,
    TASK_EXITED,
} task_status_t;
```

参照 task_status_t 的定义，任务状态可分为阻塞态，运行态，就绪态，和退出态；

```c
typedef enum {
    KERNEL_PROCESS,
    KERNEL_THREAD,
    USER_PROCESS,
    USER_THREAD,
} task_type_t;
```

参照 task_type_t 的定义，任务类型可分为内核进程，内核线程，用户进程，用户线程 4 种；

（2）如何启动一个 task，如何获得 task 的入口地址，启动时需要设置哪些寄存器：

a. 启动一个 task 首先需要初始化一个 PCB，然后将对应的 tsk 的入口地址加载到 pcb 内核上下文的 ra 寄存器中；

b. 获取 task 的入口地址，可以在 test.c 文件和 test.h 中找到对应的结构体，在这次任务中，task 的入口地址即为对应函数的入口地址，也就是函数名对应的地址；

```c
struct task_info task2_1 = {(uint32_t)&printk_task1, KERNEL_THREAD};
struct task_info task2_2 = {(uint32_t)&printk_task2, KERNEL_THREAD};
struct task_info task2_3 = {(uint32_t)&drawing_task1, KERNEL_THREAD};
struct task_info *sched1_tasks[16] = {&task2_1, &task2_2, &task2_3};
```

c. 启动时，需要将 task 的入口地址放入内核态的 ra 寄存器中，并为该进程分配足够的栈空间，即修改 sp 寄存器的值，其他寄存器，无需修改；

（3）context switch 时保存了哪些寄存器，保存在内存什么位置，使得进程再切换回来后能

正常运行：

a. context switch 时保存了除 k0，k1 之外的所有寄存器，包括通用寄存器和特殊寄存器；

b. 保存在 PCB 结构体内对应的 kernel_context 中，这样进程切换回来后只要重新加载这些数据，进程就能正常运行；

（4）问题和经验：

a. 一上来没有整体阅读代码，不知道整个程序是怎样运行的，导致开始时毫无头绪，所以在开始写代码前应该要整体阅读代码，了解直到自己究竟要干些什么，然后再来写代码，这样才能事半功倍；

b. 由于在 prj1 中，在读写 kernel 的时候直接读写了 memsz 的大小，导致这次试验的定义为 0 的全局变量出现了垃圾值，后来改为读取 filesz 的大小，并补 0 到 memsz，问题才得到了解决；

2. Mutex lock 设计流程

（1） spin-lock 和 mutual lock 的区别:

spin-lock 和 mutex_lock 最大的区别在于，spin-lock 是一旦一个进程获取 lock 失败就会一直等待，虽然设计简单，但会浪费许多 CPU 时间，mutex_lock 是一旦一个进程获取失败就会被挂起，切换为执行下一个进程，直到正在使用 lock 的进程结束才会被释放，更好的利用了 CPU 时间；

（2） 无法获得锁时的处理流程:

a. 首先，将当前进程的状态修改为 BLOCKED 状态，然后 push 到 BLOCKED 队列中；

b. 然后，如果当前进程没有出现在等待用锁的队列中，则将当前进程 push 到等待队列中；

c. 调用 do_scheduler 函数切换进程，保存恢复上下文；

（3） 被阻塞的 task 何时再次执行:

正在使用 lock 的进程结束时，会调用 mutex_release 函数，这时会将所有因为没有获取到 lock 而被阻塞的进程重新添加到 ready 队列中等待执行；

（4） 问题与经验：

在写代码前，可以在脑中想象一般进程调度的过程，模拟每个函数的执行情况，对于 mutex_lock 的设计，每个人都有自己的想法，只要切实可行即可；

3. 关键函数功能

（1） while (1) 函数：

```
while (1)
{
    // (QAQQQQQQQQQQ)
    // If you do non-preempti

    do_scheduler();
};
```

这个函数即我们的主进程 PCB[0]，由于它的存在，我们才会不断地执行进程调度函数，实现每个进程的切换；

（2） do_sheduler()函数：

```
NESTED(do_scheduler, 0, ra)
    SAVE_CONTEXT(KERNEL)
    jal     scheduler
    RESTORE_CONTEXT(KERNEL)
    jr      ra
END(do_scheduler)
```

这个函数在 entry.S 文件中，联系了汇编代码个 c 语言代码，完成了进程的上下文的保存和恢复以及进程的切换；

（3） init_pcb()函数：

这个函数完成了进程块的初始化操作，并将初始化完毕的进程添加到 ready_queue 队列中，从而让进程得以不断切换；

（4） do_mutex_acquire()函数：

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    if(lock->status == UNLOCKED)
    {
        queue_push(&(lock->wait_queue),current_running);
        lock->status = LOCKED;
    }
    else if((lock->wait_queue).head != current_running)
    {
        if(!is_in_queue(&(lock->wait_queue),current_running))
        {
            queue_push(&(lock->wait_queue),current_running);
            do_block(&block_queue);
        }
    }
    do_scheduler();
}
```

这个函数完成了对 lock 的分配，确保只有一个进程能获得 lock；

参考文献

[1] 感谢康齐翰和王浩宇同学的帮助，让我能快速的了解完成这次的 task；