Project2 A Simple Kernel 设计文档（Part I）

中国科学院大学

蔡润泽

2019 年 9 月 28 日

1. 任务启动与 Context Switch 设计流程

（1） PCB 包含的信息：

本设计中，pcb 的数据结构如下：

```
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;
```

其中，regs_context_t 类型保存 32 个 MIPS 寄存器的值，以及几个特殊 CP0 寄存器的值，包括： cp0_status、hi、lo、cp0_badvaddr、cp0_cause、cp0_epc 和 pc。

同时，分别用 kernel_stack_top 和 user_stack_top 保存内核态和用户态的栈顶位置。prev 和 next 指针分别指向该 PCB 的前面和后继 PCB，并用 pid 记录该 PCB 的编号。

type 和 status 分别用于记录 PCB 所指任务的类型和 PCB 所指任务的运行状态。

最后，利用 cursor_x 和 cursor_y 来记录被用于打印输出的位置。

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

程序在运行时，利用 do_scheduler 函数来进行任务切换。do_scheduler 函数的运行主要分为三个阶段:

第一阶段是利用 entry.S 中的函数 SAVE_CONTEXT 来保存当前运行任务的 MIPS32 个寄存器的值（除去 k0 和 k1 寄存器），将他们存放在 PCB 中的相应位置。在后续实验中还需要存储几个 CP0 协寄存器。

第二阶段是调用 sched.c 中的 scheduler 函数进行任务指针的切换。此时，记录当前运行任务的 current_running 指针将当前任务的运行状态更改为 TASK_READY 并将其放入就绪队列。同时，current_running 指向就绪队列中首位新出队的任务，并将其的任务状态改为 TASK_RUNNING。

第三阶段是调用 entry.S 中的 RESTORE_CONTEXT 函数来对任务进行恢复，即将 PCB 中所存储的 32 个寄存器的值加载入 MIPS 的 32 个寄存器中（除去 k0 和 k1 寄存器）。同时，利用 jr 指令跳转到 ra 寄存器所存的地址。在后续实验中还需要存储几个 CP0 协寄存器。

当一个 task 被启动时，主要是涉及到了上述第二、第三阶段的两个函数。

在 main 函数里进行 PCB 初始化时，该设计将需要运行的 task 的入口地址（entry_point）置于 PCB 的 31 号寄存器（ra 寄存器）中。这样，当一个 task 在被 scheduler 调度时，会通过 RESTORE_CONTEXT 函数加载相应 32 个寄存器和特殊协寄存器的值，并通过 jr ra 指令跳转到 task 的入口地址，从而实现启动。

（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

在进行上下文切换时，函数 SAVE_CONTEXT 会保存当前运行任务的 MIPS32 个寄存器的值（除去 k0 和 k1 寄存器），在后续实验中还需要存储几个 CP0 寄存器。由于该阶段的实验只涉及到内核态，因此这些寄存器的值被保存在相应 PCB 地址的 0-156 字节的位置。

（4） 设计、实现或调试过程中遇到的问题和得到的经验

在这部分的实验设计中，相较于 project1，本次实验的代码框架分布在若干个文件夹内的不同文件中。因此，如果要掌握整个代码框架和程序运行流程，必须要先熟悉不同框架内的函数和数据结构的定义，这个过程相对比较繁琐。此时，拥有一个比较方便的代码编辑器就显得格外

重要。在这次实验中，我使用了一个未曾使用过的 sublime 内的功能：go to definition。通过该功能，我能直接对与某个函数或者变量类型，直接检索到其定义语句。

另外，我之前对于 MIPS 汇编中函数的使用并不熟悉。导致对于“.macro SAVE_CONTEXT offset”语句的理解出现了偏差。后来通过代码的调试，以及在 CSDN 上查阅相关的文档，我熟悉了这种 MIPS 汇编中函数的使用方式。[1]

2. Mutex lock 设计流程

（1） spin-lock 和 mutual-lock 的区别

自旋锁（spin-lock）的实现是通过设置一个变量，当线程要进入临界区（被加锁的操作区）时，先检查这个变量。倘若该变量在临界区，且此时没有其他线程在访问这个变量，即状态为（UNLOCKED）那么该线程就进入临界区，并且将该变量置为有线程访问状态（LOCKED）。如果这个变量状态为有其他线程在访问（LOCKED），就不断的使用 while 重试，直到可以进入临界区。因此，使用自旋锁的坏处就是倘若一旦获取锁不成功，那么线程就会一直进行循环来尝试获取锁，这会极大的消耗 CPU 资源。

而互斥锁（mutual-lock）为一旦线程请求锁失败，那么该线程会自动被挂起到该锁的阻塞队列中，从而不会被调度器调度。当占用该锁的线程释放锁之后，被阻塞的线程会从阻塞队列中重新放到就绪队列，并获得锁。因此，使用互斥锁相较于自旋锁可以节约 CPU 资源。

（2） 无法获得锁时的处理流程

当进程获取锁失败时，会将该进程的运行状态置为 TASK_BLOCKED，并将其放置于阻塞队列。同时，系统会调用 do_scheduler 函数进行上下文切换。

（3） 被阻塞的 task 何时再次执行

当锁被释放时，系统首先会检测阻塞队列是否为空。

当阻塞队列不为空时，系统会将阻塞队列中的首任务释放，将其状态改为就绪态，并将其置于就绪队列中。同时，系统会把锁的状态置为 LOCKED。

若阻塞队列为空，系统会释放锁，即将锁的状态置为 UNLOCKED。

（4） 设计、实现或调试过程中遇到的问题和得到的经验

在这一部分代码的实现时，我发现自己写的程序的四行文本输出在同样的两行内不停的交错闪现。但通过代码分析，这应该与我锁的设置和上下文切换无关。在经过各种方法调试无果后，开始考虑是不是代码框架有问题。之后我在专业 QQ 群里发现了相应的反馈，发现测试代码 test_lock1.c 中的 print_location 有问题，经过修改，该 bug 消除。

3. 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）

PCB 的初始化，其中栈的起始位置为 `STACK_MAX= 0xa1000000`，每一块空间的大小为 `0x10000`。其中 `PCB[0]`保留，不放置测试运行的任务。

```
1.  static void init_pcb()
2.  {
3.      int i,j;
4.      pcb[0].pid=process_id++;
5.      pcb[0].status=TASK_RUNNING;
6.      int stack_top=STACK_MAX;
7.      queue_id=1;
8.
9.      queue_init(&ready_queue);
10.     queue_init(&block_queue);
11.     for(i=0;i<num_sched1_tasks;i++,queue_id++){
12.         for(j=0;j<32;j++){
13.             pcb[queue_id].kernel_context.regs[j]=0;
14.         }
15.         pcb[queue_id].pid=process_id++;
16.         pcb[queue_id].type=sched1_tasks[i]->type;
17.         pcb[queue_id].status=TASK_READY;
18.
19.         pcb[queue_id].kernel_stack_top=stack_top;
20.         pcb[queue_id].kernel_context.regs[29]=stack_top;
21.         stack_top-=STACK_SIZE;
22.
23.         pcb[queue_id].kernel_context.regs[31]=sched1_tasks[i]->entry_point;
24.         queue_push(&ready_queue,(void *)&pcb[queue_id]);
25.     }
26.
27.     for(i=0;i<num_lock_tasks;i++,queue_id++){
```

```
        for(j=0;j<32;j++){
            pcb[queue_id].kernel_context.regs[j]=0;
        }
        pcb[queue_id].pid=process_id++;
        pcb[queue_id].type=lock_tasks[i]->type;
        pcb[queue_id].status=TASK_READY;

        pcb[queue_id].kernel_stack_top=stack_top;
        pcb[queue_id].kernel_context.regs[29]=stack_top;
        stack_top-=STACK_SIZE;

        pcb[queue_id].kernel_context.regs[31]=lock_tasks[i]->entry_point;
        queue_push(&ready_queue,(void *)&pcb[queue_id]);
    }
     current_running=&pcb[0];
}
```

任务调度函数，通过 current_running 指针的变化来切换任务：

```
void scheduler(void)
{
    if(current_running->status!=TASK_BLOCKED){
        current_running->status=TASK_READY;
        if(current_running->pid!=1){
            queue_push(&ready_queue,current_running);
        }
    }
    if(!queue_is_empty(&ready_queue))
        current_running=(pcb_t *)queue_dequeue(&ready_queue);
    current_running->status=TASK_RUNNING;
}
```

锁的获取与释放：

```
void do_mutex_lock_acquire(mutex_lock_t *lock)
{
    if(lock->status==LOCKED){
        do_block(&block_queue);
    }
    else
        lock->status=LOCKED;
}

void do_mutex_lock_release(mutex_lock_t *lock)
```

```
{
    if(!queue_is_empty(&block_queue)){
        do_unblock_one(&block_queue);
        lock->status=LOCKED;
    }
    else
        lock->status=UNLOCKED;

}
```

参考文献

[1] MIPS 中的异常和系统调用：
https://blog.csdn.net/jasonchen_gbd/article/details/44044091
[2] 国科大操作系统研讨课任务书 Project 2 (Part I)