Project2 A Simple Kernel 设计文档

中国科学院大学
陈灿宇
2018.10.31

1　任务启动与 Context Switching 设计流程

(1) PCB 包含的信息

```c
/* Process Control Block */
typedef struct pcb
{
    /* register context */
    regs_context_t kernel_context;
    regs_context_t user_context;
    //312
    uint32_t kernel_stack_top;
    uint32_t user_stack_top;
    //320
    uint32_t entry_point;

    task_mode_t mode;
    /* 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;

    uint32_t wait_time;
    priority_t priority;

    uint32_t sleeping_deadline;

} pcb_t;
```

图 1: PCB

进程控制块 (process control block, PCB) 中存储了与操作系统中一个进程所处状态相关的全部信息, 保存的内容通常包括进程状态、程序计数器 (PC)、各通用寄存器、进程号 (PID) 等。每个进程的创建与销毁都由内核负责, 它们都需要在 PCB 中登记信息。在操作系统发生上下文切换时, 我们需要保存和载入相应任务的 PCB。

本次实验中, 我记录在 PCB 中的信息包括: 进程状态 **status**，包括被阻塞 (TASK_ BLOCKED) 、运行中 (TASK_RUNNING) 、就绪待运行 (TASK_READY) 、已退出 (TASK_EX ITED) 、创建完成 (TASK_CREATED)、睡眠状态 (SK_SLEEPING)；核心态上下文寄存器 **kernel_ context**; 用户态上下文寄存器 **user_context**; 内核栈和用户栈栈顶指针 **kernel_stack_top**、**user_stack_top**; 进程入口地址 **entry_point**; 进程当前模式（内核态/用户态）**mode**; 指向前一个进程的指针 **prev**; 指向后一个进程的指针 **next**; 进程编号 **pid**; 进程类型 **type**，包含 (KERNEL_PROCESS, KERNEL_THREAD, USER_ PROCESS, USER_THREAD); 当前光标位置 **cursor_x**、**cursor_y**; 进程等待时间 **wait_time**; 进程优先级 **priority**; 进程睡眠截止时间 **sleeping_deadline**。

(2) 如何启动第一个 task，例如如何获得 task 的入口地址，启动时需要设置哪些寄存器等

非抢占式调度启动第一个 task 比较简单，首先初始化所有 task 的 PCB, 其中寄存器上下文中初始化栈指针和入口点; 之后将 task 的 PCB 依次推入 ready_queue, 并启动调度 do_scheduler。

do_scheduler 寻找第一个 task 的入口地址时，会通过 jr ra 跳转到 ra 寄存器中的地址值，所以只需要在初始化 PCB 的时候将内核栈的 ra 寄存器置为测试进程 sched_tasks 的入口地址即可。

```
1    //main.c (task1,2)
2    pcb[i].kernel_context.regs[31] = sched1_tasks[i]->entry_point;
```

抢占式调度则相对更为复杂，首先在初始化之前需要关中断，将 CP0_STATUS 寄存器末位置为 0, 然后初始化所有 task 的 PCB, 之后将 task 的 PCB 依次推入 ready_queue, 在初始化结束之后开中断，将 CP0_STATUS 寄存器末位置为 1。

需要注意的一点是在进行初始化的过程中是在内核态进行的，而时钟中断必须从用户态切换到内核态，所以我设计了一个 first_entry 函数，在初始化 PCB 的时候将内核栈的 ra 寄存器置为 first_entry 函数地址，在第一次 do_scheduler 结束之后，会跳转到 first_entry 函数，然后通过 RESTORE_CONTEXT(USER) 和 eret 指令跳回到用户态, 可以造成一种 “我是从用户态进来的假象”，回到 EPC 所存放的地址值，开始第一次运行。关键代码如下：

```
    //main.c (task3,4)
    pcb[i].kernel_context.regs[31] = (uint32_t)first_entry;
    //every process first entry point
    //acccomplish the shift from kernel mode to user mode

    pcb[i].user_context.cp0_epc = sched2_tasks[i]->entry_point;
    //finish the shift from kernel mode to user mode, entering the process in user mode
    //cp0_epc add 4 automatically when encountering interrupt
```

```
    //entry.S
...
int_finish:
    RESTORE_CONTEXT(USER)
    j     return_from_exception
    nop
...
    LEAF(first_entry)
    j     int_finish
    END(first_entry)

    LEAF(return_from_exception)
    STI
    eret
    END(return_from_exception)
```

(3) context switching 时保存哪些寄存器，保存在内存什么位置，使得进程再切换回来后能正常运行

在非抢占式调度中，当调用 do_scheduler 函数时，ra 寄存器中已经存入了 PC+4, SAVE_CONTEXT 宏只需要将 ra 寄存器的值保存到内核栈中，就可以使得进程再切换回来后能正常运行。

在抢占式调度中，由于还需要 RESTORE_CONTEXT(USER) 和 eret 来返回用户态，所以 SAVE_CONTEXT 宏除了需要将 ra 寄存器的值保存到内核栈中，还需要将 EPC 寄存器中的值保存到用户栈中。

(4) 设计、实现或调试过程中遇到的问题和得到的经验

在这阶段遇到的最严重的一个 BUG 是我利用 GDB 跟踪的结果与 kernel.txt 中的指令不符，使我一度百思不得其解，后来在通过反汇编查看 image 中的实际指令之后，我才发现 kernel.txt 与 image 中实际的指令也不相符。然后我推测是 createimage.c 中有问题，最后我发现是我在保存 kernel_size 时，由于 kernel 的大小已经远超过了实验一，所以用 puts 写入时发生了溢出（puts 只能写入一个 byte）。

2　时钟中断、系统调用与 blocking sleep 设计流程

(1) 时钟中断处理的流程，请说明你认为的关键步骤即可

首先，中断处理的一般流程：当触发异常时,CPU 自动跳转到 0xbfc00180 处执行, 首先 STATUS 寄存器第 0 位置零关中断, 之后根据异常号跳转至不同的处理程序 (如中断处理程序、系统调用处理程序或其他处理程序)。

进入中断处理程序后, 保存当前任务的上下文, 根据中断源跳转至相应处理程序 (若中断源未定义则不跳转) , 处理完成后将 CAUSE 寄存器的 IP7-IP0 置零表示清中断, 读取任务上下文, 开中断并返回 EPC 继续执行。

而时钟中断处理流程建立在中断处理的一般流程的基础之上。判断中断源为时钟中断时跳转至时钟中断处理程序。首先将 COUNT 寄存器置零, 重写 COMPARE 寄存器, 表示清时钟中断; 之后更改 time_elapsed , 用于 sleep 功能的计时。通过进程的 mode 判断当前任务处于内核态还是用户态, 若处于内核态表示任务是内核线程, 不参与时钟中断的调度, 因此直接跳过调度, 执行中断处理结束的流程 (即保存任务上下文, 开中断并返回 EPC) ; 若处于用户态表示任务是进程, 需要在时钟中断内调度, 因此先保存内核上下文, 将正在执行的任务放入 ready 队列, 启动调度器调度新的任务, 恢复内核上下文, 之后执行中断处理结束的流程。

对于处于 sleep 状态的任务, 在每次任务调度时都会检查是否有任务需要唤醒, 若已经到达唤醒时间则将其从 sleeping 队列中弹出, 并加入 ready 队列。

(2) 你所实现的时钟中断的处理流程中，如何处理 blocking sleep 的任务，你如何决定何时唤醒 sleep 的任务？

do_sleep 函数首先更改进程 status, 计算出唤醒时间, 将正在运行的任务加入 sleeping 队列, 该队列是以唤醒时间为标准排好序的队列，队首是 deadline 最小的一个进程，然后启动调度, 运行其他任务。

每次任务调度时都检查是否有任务需要唤醒, 由于 sleeping 队列是有序队列, 只需不断检验队头任务的唤醒时间, 若已经到达唤醒时间则将其从 sleeping 队列中弹出并加入 ready 队列, 直至队头任务的唤醒时间晚于当前时间。

(3) 你实现的时钟中断处理流程和系统调用处理流程有什么相同步骤，有什么不同步骤？

相同步骤：时钟中断处理流程和系统调用处理流程在下图中除了第 3 步不同，其它部分基本一样。

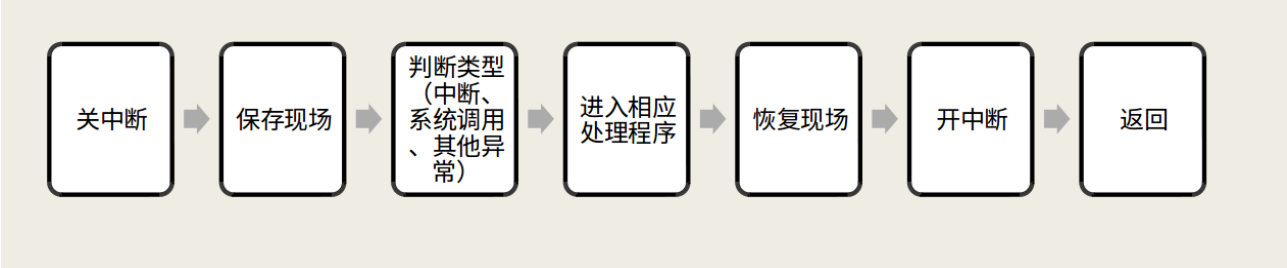

图 2: 例外处理过程

不同步骤：在时钟中断处理流程中一般会发生进程切换（除非其它进程全被 block 或 sleep 了），而系统调用不一定会引起进程切换，比如当 syscall[SYSCALL_SLEEP] 和 syscall[SYSCALL_BLOCK] 时会发生进程切换，而其它的系统调用如 syscall[SYSCALL_WRITE] 和 syscall[SYSCALL_CURSOR] 则不会。

(4) 设计、实现或调试过程中遇到的问题和得到的经验

在实现了 syscall 之后，我遇到了一个非常严重的 bug，就是会表现出多个进程同时抢到一把锁。

我一读非常困惑，为什么 task2 中还能正常抢锁，到 task4 之后就不能正常抢锁了呢？后来经过长时间的思考，我发现还是因为我对于 syscall 的理解不够透彻。这个 BUG 与下面三个函数有关，在原来一个锁被 release 之后，被 block 的进程就进入 ready_queue，下一次不用再次查询该锁是否被锁住了，这一点严重欠考虑，将其改为 while 以后就解决了这个问题，下次切换到该进程的时候还需要再次查询该锁是否被锁住。

```
1  void do_mutex_lock_acquire(mutex_lock_t *lock)
2  {
3  /*
4      //Before revising
5      if(lock->status == LOCKED){
6          do_block(&block_queue);
7      }
8  */
9      //After revising
10     while(lock->status == LOCKED){
11         do_block(&(lock->mutex_lock_queue));
12     }
13     lock->status = LOCKED;
14 }
15
16 void do_mutex_lock_release(mutex_lock_t *lock)
17 {
18     if(lock->status == LOCKED){
19         lock->status = UNLOCKED;
20         do_unblock_one(&(lock->mutex_lock_queue));
21     }
22 }
23
24 void do_block(queue_t *queue_ptr)
25 {
26     // block the current_running task into the queue
27     if(current_running->status == TASK_RUNNING){
28         current_running->status = TASK_BLOCKED;
29
30         queue_sort(queue_ptr, current_running, priority_comp);
31     }
32     do_scheduler();
33 }
```

3　基于优先级的调度器设计

(1) priority-based scheduler 的设计思路，包括在你实现的调度策略中优先级是怎么定义的，何时给 task 赋予优先级，测试结果如何体现优先级的差别

在我的设计中在 ini_pcb 中为每一个任务设定一个初始优先级 INITIAL_PRIORITY，然后每次调度的时候在 ready_queue 中查找优先级最高的进程，如果最高优先级的有多个进程，则选择靠近队首的那个进程。同时为了避免高优先级的进程无穷运行下去，所以每当一个进程运行一次，其优先级减一；同时，如果当所有的进程的优先级都小于 0, 则将所有的进程大的优先级重新初始化一次。

不过我的设计还比较 naive，还在思考和改进中。

```
1  void scheduler(void)
2  {
3  ...
4      pcb_t *_current_running = ((pcb_t *)(ready_queue.head));
5      while(_current_running != ((pcb_t *)(ready_queue.tail)) \
6      && _current_running->priority < ((pcb_t *)(_current_running->next))->priority){
7          _current_running = ((pcb_t *)(_current_running->next));
8      }
9      if(_current_running->priority < 0){
10         _current_running = ((pcb_t *)(ready_queue.head));
11         while(_current_running != ((pcb_t *)(ready_queue.tail))){
12             _current_running->priority = INITIAL_PRIORITY;
13             _current_running = ((pcb_t *)(_current_running->next));
14         }
15         current_running = queue_dequeue(&ready_queue);
16     }
17     else{
18         current_running = _current_running;
19         queue_remove(&ready_queue, _current_running);
20     }
21
22     current_running->priority--;
23 ...
24 }
```

基于我的设计，如果每一个进程的 INITIAL_PRIORITY 是相同的，一段时间内每个进程得到调度的次数应该是比较平均的，而在我的测试中也验证了这一点，可以在每一个 task 中打印出该进程得到调度的次数。如果想要增加某一个进程运行的次数只需要改变该进程的 INITIAL_PRIORITY 即可。

(2) 设计、实现或调试过程中遇到的问题和得到的经验

4 Mutex lock 设计流程

(1) spin-lock 和 mutual lock 的区别

自旋锁 (spin-lock) 与互斥锁 (mutual lock) 都是为了保证多线程在同一时刻只能有一个线程操作临界区 (critical section) 而引入的。不同点在于, 对于互斥锁, 如果目前是加锁状态, 那么后面的线程就会进入“休眠”状态, 解锁之后, 又会被唤醒继续执行; 如果是自旋锁, 那么后面的线程会一直等待, 直到锁被释放后立刻执行。因此, 自旋锁比较适合执行较短的任务, 否则会产生较大的性能消耗。

(2) 能获取到锁和获取不到锁时各自的处理流程

当一个任务申请锁时, 如果能获取到锁, 我们就加锁, 然后继续执行任务的剩余部分; 否则调用 do_block() , 将该任务加入阻塞队列并修改 PCB 记录为阻塞状态, 并切换进程。

```
1   void do_mutex_lock_acquire(mutex_lock_t *lock)
2   {
3       while(lock->status == LOCKED){
4           do_block(&(lock->mutex_lock_queue));
5       }
6       lock->status = LOCKED;
7   }
8
9   void do_block(queue_t *queue_ptr)
10  {
11      // block the current_running task into the queue
12      if(current_running->status == TASK_RUNNING){
13          current_running->status = TASK_BLOCKED;
14          queue_sort(queue_ptr, current_running, priority_comp);
15      }
16      do_scheduler();
17  }
```

(3) 被阻塞的 task 何时再次执行

当锁被释放时, 被阻塞的 task 会被插入 ready_queue 队列, 待下次调度到时再次执行。

(4) 设计、实现或调试过程中遇到的问题和得到的经验

5 Bonus 设计思路

(1) 何处理一个进程获取多把锁

不需要做改动，只需要修改测试用例即可。下图展示了 LOCK TASK 1 获取两把锁的结果图。

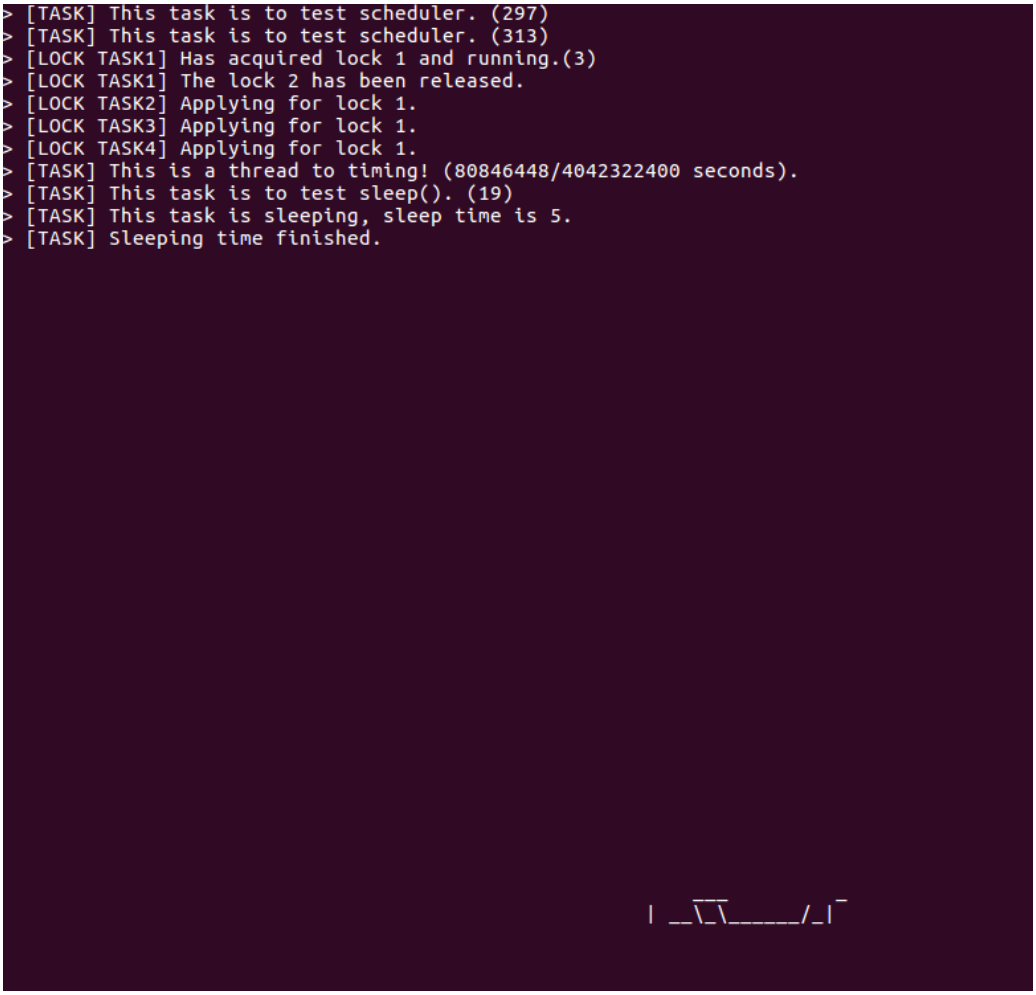

图 3: 一个任务获取多把锁结果图

(2) 如何处理多个进程获取一把锁

只需要一个互斥锁维护一个阻塞队列即可，然后 do_mutex_lock_init, do_mutex_lock_acquire, do_mutex_lock_release 函数分别作相应调整。

```
1    typedef struct mutex_lock
2    {
3    queue_t mutex_lock_queue;
4    lock_status_t status;
5    } mutex_lock_t;
```

(3) 你的测试用例和结果介绍

我设计的测试用例中 LOCK TASK 1 可以获取 LOCK 1 和 LOCK 2 两把锁，LOCK TASK 2,LOCK TASK 3,LOCK TASK 4 争夺 LOCK 1。下图展示了 LOCK TASK 4 获取 LOCK 1，其它进程被阻塞的结果图。

图 4: 多个任务抢一把锁结果图

(4) 设计、实现或调试过程中遇到的问题和得到的经验

6　关键函数功能

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）

下面这个队列排序的函数是我花了一番功夫才写好的，所以贴上来了。

```
1  //queue.c
2  void queue_sort(queue_t *queue, void *item, item_comp_t item_comp)
3  {
4      item_t *_item = (item_t *)item;
5      item_t *item_next = NULL;
6      item_t *item_prev = NULL;
7
8      if(queue_is_empty(queue)){
9          queue->head = item;
10         queue->tail = item;
11         _item->next = NULL;
12         _item->prev = NULL;
13     }
14     else{
15             void *head = queue->head;
16             while(((item_t *)(queue->head))->next != NULL \
17             && item_comp(((item_t *)(queue->head)),item) == 1){
18             queue->head = ((item_t *)(queue->head))->next;
19         }
20
21         if(((item_t *)(queue->head))->next == NULL
22             && item_comp(((item_t *)(queue->head)),item) == 1){
23             ((item_t *)(queue->tail))->next = item;
24             _item->next = NULL;
25             _item->prev = queue->tail;
26             queue->tail = item;
27
28             queue->head = head;
29         }
30         else if(head == queue->head
31             && item_comp(((item_t *)(queue->head)),item) == 0) {
32             _item->prev = NULL;
33             _item->next = queue->head;
34             ((item_t *)(queue->head))->next = item;
35             queue->head = item;
36         }
37         else{
38             item_next = ((item_t *)(queue->head))->next;
39             item_prev = ((item_t *)(queue->head));
40
41             item_prev->next = item;
42             _item->prev = item_prev;
43             item_next->prev = item;
```

```
            _item->next = item_next;

            queue->head = head;
        }
    }
}
```

时间中断的处理函数我为了方便就直接利用汇编来写了，没有按照老师给的框架。

```
NESTED(handle_int, 0, sp)
    .set  mips32

    SAVE_CONTEXT(USER)

    mfc0  k0, CP0_CAUSE      /* Read Cause register for IP bits */
    nop
    andi  k0, k0, CAUSE_IPL  /* Keep only IP bits from Cause */
    clz   k0, k0             /* Find first bit set, IP7..IP0; k0 = 16..23 */
                             /* The CLZ instruction counts the number of leading
                              * zeros in a word. Scan rs and write into rd.
                              */
    xori  k0, k0, 0x17       /* 16..23 => 7..0 */

    addiu k1, zero, 7
    beq   k0, k1, irq_timer
    nop

    jal   clear_int

    int_finish:
    RESTORE_CONTEXT(USER)
    j     return_from_exception
    nop

irq_timer:

    jal   reset_timer

    jal   clear_int

    lw    k1, time_elapsed
    addi  k1, k1, 15
    sw    k1, time_elapsed

    TEST_TASK_MODE
    beq   k1, zero, int_finish
    nop

    lw    k0, current_running
    sw    zero, TASK_MODE_OFFSET(k0)
```

```
42
43      jal    do_scheduler
44      nop
45
46      lw     k0, current_running
47      li     k1, 1
48      sw     k1, TASK_MODE_OFFSET(k0)
49
50      j      int_finish
51      nop
52
53  clear_int:
54      mfc0   k0, CP0_CAUSE
55      nop
56      andi   k1, k0, CAUSE_IPL
57      xor    k0, k0, k1
58      mtc0   k0, CP0_CAUSE
59      nop
60      jr     ra
61
62  END(handle_int)
```