Project2 A Simple Kernel 设计文档

中国科学院大学
[薛峰]
[2018/10/29]

1. 任务启动与 Context Switching 设计流程

（1）PCB 包含的信息

PCB 的结构体定义如右图所示。

其中 kernel_context 和 user_context 分别为内核寄存器堆和用户态寄存器堆，分别用于保存内核态的 32 个通用寄存器和一些特殊寄存器；

prev 和 next 用于指向其他的 PCB，从而用来实现队列操作；

pid 用于表示该进程的标号；

addr 用于存放返回的地址；

type 用于表明该进程的类型；

status 用于表明该进程的状态，例如 RUNNING，READY，BLOCKED 等；

priority 用于表明该进程的优先级，用于实现基于优先级的调度算法；

sleep_deadline 用于表明该进程被重新唤醒的时间；

cursor_x 和 cursor_y 用于表明该进程正在打印的光标的位置，防止在打印前出现时钟中断从而打印到其他进程的位置的情况。

```c
/* Process Control Block */
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;

    /* return address */
    uint32_t addr;

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

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

    /* priority */
    priority_t priority;

    /* sleep time */
    uint32_t sleep_deadline;

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

} pcb_t;
```

（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

对于非抢占式调度（PCB 中只需要一套寄存器堆即可，假设只用 user_context），在 PCB 初始化时，需要将 ra 寄存器赋值为对应进程的入口地址。main 函数会执行 do_schedule()，当保存现场、更改 current_running 并恢复现场之后，此时 ra 即为将要执行的进程的入口地址，通过 jr ra 指令便可启动第一个 task。其中 task 的入口地址保存在结构体 task_info 的 entry_point 域中。task_info 结构体的定义如下如所示：

```c
/* task information, used to init PCB */
typedef struct task_info
{
    uint32_t entry_point;
    task_type_t type;
} task_info_t;
```

对于抢占式调度，启动第一个 task 需要通过时钟中断实现。并且 do_schedule 中保存和恢复的寄存器为内核态的寄存器，因此，在对 PCB 进行初始化的过程中，需要将 kernel 的 31 号寄存器赋值为 handle_int 的返回部分，并且将用户态的 cp0_epc 寄存器赋值为用户程序的入口地址，这样便可通过 eret 指令启动第一个 task。

（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

需要保存的寄存器有：30 个通用寄存器（不包含 k0 和 k1），以及一些特殊寄存器：cp0_status，hi，lo，cp0_badvaddr，cp0_cause，cp0_epc。

应保存在对应的 PCB 当中。这样，在进行进程切换后，只需将新的进程的 pcb 中保存的寄存器的值通过 lw 指令恢复到对应的寄存器中即可。

（4）设计、实现或调试过程中遇到的问题和得到的经验

问题：

一、开始时，对 MIPS 汇编不太熟练，因此会犯一些语法上的错误。例如 do_schedule() 中，在写 SAVE_CONTEXT 和 RESTORE_CONTEXT 时，offset 前没有加\因此编译器一直报错。

二、没有对 queue 进行初始化，所以有时会打印不出结果，但有时可以正确执行。因此为确保正确性，在 main 函数中加入了对各个队列的初始化操作。

收获：

通过这一部分的实验，我真正学会了如何进行上下文的切换并掌握了进程转换的过程，并且更加熟悉了 MIPS 汇编。

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

第一步：硬件检测到时钟中断后，自动跳转到 0x80000180 这个地址；

第二步：在初始化的时候，操作系统已将该 exception_handler_entry 函数拷贝到 0x80000180 这个地址。因此第一步过后，开始执行该 exception_handler_entry 函数，该函数首先关中断，然后保存用户态的寄存器，最后根据 CP0_CAUSE 寄存器的 ExcCode 域跳转到 handle_int 函数；

第三部：handle 首先调用 interrupt_helper 函数。interrupt_helper 函数根据 CP0_CAUSE 寄存器的 IP 域确定为时钟中断，从而调用时钟中断处理函数 irq_timer()。当返回到 interrupt_helper 的时候，恢复用户态的寄存器，再开终端，最后通过 eret 指令返回到用户进程。

（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？

当任务发出睡眠请求时，先将该任务添加到 sleep 队列当中，然后根据睡眠时间和当前时间计算出 sleep_deadline，即睡眠结束的时刻，并保存到对应的 pcb 中，随后进行 do_schedule()。

在每次切换进程之前，即 schedule()函数的开始位置，都需要检查 sleep 队列中的任务是否已经到了睡眠的 deadline，即检查当前时间是否大于 pcb 中的 sleep_deadline。如果是，则将该任务从 sleep 队列中取出，放到 ready 队列中去，否则跳过该进程检查下一个进程。

（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：最开始都是从 0x80000180 地址开始取指，即一开始都是执行

exception_handler_entry 函数：先关中断，再保存用户态寄存器，最后根据 CP0_CAUSE 寄存器的 EXCCODE 域判断例外类型，从而跳转到对应的的例外处理函数；另外，在返回之前 进行的操作也是一样的，都需要恢复用户态寄存器，开中断，用 eret 返回。

不同步骤：中断处理函数不同，即 handle_int 和 handle_syscall 不同。handle_int 根据 IP 的值确定中断类型，而 handle_syscall 根据用户态传的参数判断系统调用类型；并且在返回时，handle_syscall 需要将 epc 加 4，而 handle_int 不需要。

（4）设计、实现或调试过程中遇到的问题和得到的经验

问题：

一、在进行 Task3 的过程中，发现打印不出东西。于是用 gdb 调试发现是初始化的过程中中断开的过早造成的。即开中断的操作放在 init_exception 函数中，而 init_exception 函数较早执行，导致后续队列、pcb 等还未初始化便有了第一次时钟中断。于是将 init_exception 函数最后执行，从而避免了该问题；

二、在做 Task3 时，只保存和恢复 pcb 中用户态寄存器，没用用到内核的寄存器。虽然 Task3 能够通过，但是到 Task4 的时候遇到了问题——此时既有时钟中断，又有系统调用，因为 eret 前，系统调用需要加 4 而时钟中段不需要，这就导致了返回地址的混乱。所以需要保存两套寄存器，即用户态和内核态，这样便可以解决这个问题。

三、在进行 Task4 的过程中，发现打印不出东西。通过 gdb 调试发现，程序一直卡在 syscall 处，即 syscall 处理完之后返回地址还是 syscall，因此在 syscall 通过 eret 返回前需要将 epc 加 4。

四、get_IP 函数（即返回 cp0_cause 寄存器的 IP 域）应该读 cp0_cause 寄存器却因为手误读了 cp0_status 寄存器，导致出错，并且这个 bug 调了好久才发现。

五、在做 Task4 的过程中，会发现有时会出现打印串行的情况，即本应该在第一行显示的结果却出现在第二行。后来反复用 gdb（因为种情况出现的时机是随机的）才发现错误——task1 设置过光标之后，出现了时钟中断，之后开始执行 task2，并且 task2 也设置了光标，当返回到 task1 的时候，自然打印出的结果就跑到了 task2 的位置。所以在任务调度的过程中还应该保存每个任务的光标。因为这种情况出现的时机并不可预知，所以这个 bug 查了好久才查出来。

六、在用 QEMU 调试的过程中，发现打印不出 task 中的信息，并且第一次中断时用 ir 观察寄存器时发现 cause 寄存器最后一位是 8，查表后发现是 TLB 例外。于是检查初始化部分的代码，发现是在初始化 PCB 时发现数组越界的情况，因此会出现 TLB 例外。

收获：

首先，这一部分的内容让我完全理清楚了中断和系统调用处理的过程，通过亲自写代码更能增加印象。还有就是这一部分的 bug 确实很难定位，很多 bug 需要反复用 gdb 才能找到，所以写代码的时候会很认真，尽量避免因手误而产生的错误，并且代码写完之后会再重新理一边代码的整体思路，确保无误之后再上板测试，但还是难免会出现差错。但是调 bug 的过程确实让我更加清楚地理解了例外的处理流程。

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

共有五个优先级（一级最高，五级最低），并对应有五个不同的 ready_queue。在初始化的过程中，每个优先级都会赋予最高优先级。每次经过一个时间片，在 schedule()函数中，

会将进程的优先级降低一级，然后放到对应的准备队列中。再调用新的进程时，会按照优先级，先检查高优先级的准备队列中是否有进程，如果有，则调用，如果没有则检查低优先级的队列。这样便可实现优先级的动态赋值和基于优先级的调度。

对于测试结果，可以在初始化 PCB 时将打印飞机的任务优先级设置为三级，这样测试结果可以看出，打印字符串的两个任务执行了两次之后，才会开始打印飞机。

（2）设计、实现或调试过程中遇到的问题和得到的经验

无。

4. Mutex lock 设计流程

（1）spin-lock 和 mutual lock 的区别

对于自旋锁，如果目前是加锁状态，则该线程会一直等待，直到锁被释放后立刻执行。

而对于互斥锁，如果目前是加锁状态，那么后面的线程就会进入“休眠”状态，并被挂起到阻塞队列。当该锁被释放后，该线程才会被调入到准备队列。

（2）被阻塞的 task 何时再次执行

当被阻塞的 task 申请的锁被其他线程释放后，该线程才会被调入到准备队列中。之后被 schedule()调用时，才会被执行。

（4）设计、实现或调试过程中遇到的问题和得到的经验

无。

5. Bonus 设计思路

（1）如何处理一个进程获取多把锁

每个锁中设置 pid 域，用来表示占有该锁的进程的 pid。这样每次查看锁的 pid 域便可知道是哪个进程占有该锁，也可以通过便利锁知道一个进程占有哪几个锁。

（2）如何处理多个进程获取一把锁

对于每个锁都创建一个队列，我的测试用例中有三个锁，因此创建了三个阻塞队列。如果某个进程申请某个锁没申请到，那么就挂进这个锁对应的队列当中。当释放这个锁的时候，就从这个锁对应的队列中释放一个优先级最高的进程，放到准备队列中。

（3）你的测试用例和结果介绍

测试用例有三个锁，有三个进程申请这三个锁，并且三个进程的优先级关系为：进程 1 > 进程 3 > 进程 2。每个进程都会同时申请这三把锁，首先进程先申请到这三把锁，之后根据优先级关系切换到进程 3，

然后进程三申请锁 1，发现锁 1 被占有，因此会放到锁 1 的阻塞队列；再切换到进程 2，同理，进程 2 也被阻塞，放到锁 1 的阻塞队列中。

再切换到进程 1，进程 1 运行一段时间后释放三把锁，释放锁 1 后，根据优先级关系，把进程 3 从锁 1 的队列中释放出来。之后进程 3 便拥有锁 1，随后切换到进程 3 后，进程 3 还会申请锁 2，和锁 3。

这样不断循环，可以看到最后的效果为，进程 1、进程 3、进程 2 依次轮流占据这三个锁。并且该顺序也体现了优先级关系。

6. 关键函数功能

一、保存上下文代码：

代码如右图所示（之列出部分代码，未列出的部分与之类似，保存一些特殊寄存器）。

因为 k0 和 k1 寄存器的作用为保存中断信息，因此使用 k0 和 k1 寄存器保存上下文，并且这两个寄存器不用保存在 PCB 中。

除了通用寄存器外，还需要保存些特殊寄存器：cp0_status，hi，lo，cp0_badvaddr，cp0_cause，cp0_epc。

```
.macro SAVE_CONTEXT offset
    .set noat
    la  k0, current_running
    lw  k0, 0(k0)
    addi k0, k0, \offset

    sw  $0 , OFFSET_REG0(k0)
    sw  $1 , OFFSET_REG1(k0)
    sw  $2 , OFFSET_REG2(k0)
    sw  $3 , OFFSET_REG3(k0)
    sw  $4 , OFFSET_REG4(k0)
    sw  $5 , OFFSET_REG5(k0)
    sw  $6 , OFFSET_REG6(k0)
    sw  $7 , OFFSET_REG7(k0)
    sw  $8 , OFFSET_REG8(k0)
    sw  $9 , OFFSET_REG9(k0)
    sw  $10, OFFSET_REG10(k0)
    sw  $11, OFFSET_REG11(k0)
    sw  $12, OFFSET_REG12(k0)
    sw  $13, OFFSET_REG13(k0)
    sw  $14, OFFSET_REG14(k0)
    sw  $15, OFFSET_REG15(k0)
    sw  $16, OFFSET_REG16(k0)
    sw  $17, OFFSET_REG17(k0)
    sw  $18, OFFSET_REG18(k0)
    sw  $19, OFFSET_REG19(k0)
    sw  $20, OFFSET_REG20(k0)
    sw  $21, OFFSET_REG21(k0)
    sw  $22, OFFSET_REG22(k0)
    sw  $23, OFFSET_REG23(k0)
    sw  $24, OFFSET_REG24(k0)
    sw  $25, OFFSET_REG25(k0)

    sw  $28, OFFSET_REG28(k0)
    sw  $29, OFFSET_REG29(k0)
    sw  $30, OFFSET_REG30(k0)
    sw  $31, OFFSET_REG31(k0)

    mfc0 k1, CP0_STATUS
    sw   k1, OFFSET_STATUS(k0)
```

二、do_scheduler()函数

代码如下图所示。

先保存上下文，再改变切换进程，然后恢复新的进程的上下文，最后通过 jr ra 指令返回到用户程序。

需要注意的是，该部分恢复和保存的都是核心态的寄存器，因为在例外处理的过程中，do_schedule()函数是在运行内核代码的过程中调用的，调用前后都是在执行内核代码。只有在从用户态切换到核心态时，才会保存用户态寄存器，同样地，在从核心态返回到用户程序时，才需要恢复内核态的寄存器。

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

三、例外处理入口

这一部分代码在初始化的时候被 copy 到地址 0x80000180。其流程是先关中断，然后保存用户态的寄存器，然后根据 cp0_cause 寄存器的 EXCCODE 域判断例外类型，从而跳转到对应的的例外处理函数。代码如下：

```
NESTED(exception_handler_entry, 0, sp)
exception_handler_begin:
  // Leve2 exception Handler.
  // close interrupt
  CLI

  // save contex
  SAVE_CONTEXT(USER)

  // jmp exception_handler[i] which decided by CP0_CAUSE
  mfc0  k0, CP0_CAUSE
  andi  k0, k0, CAUSE_EXCCODE
  la    k1, exception_handler
  add   k0, k0, k1
  lw    k0, 0(k0)
  jr  k0
exception_handler_end:
END(exception_handler_entry)
```

四、handle_int()函数

其功能为跳转到 interrupt_helper()函数（判断中断类型，并调用对应的中断处理函数），并返回到用户程序。返回之前应恢复用户态的寄存器，并开中断，最后由 eret 返回。其中 eret 指令的功能是返回到 cp0_epc 寄存器内的地址，而 cp0_epc 寄存器的值是由产生中断时硬件自动设置的。需要注意的是 handle_syscall 函数与 handle_int 类似，不过 handle_syscall 在返回前需要将 epc 加 4，因为当遇到 syscall 指令时，硬件会将 cp0_epc 的值设为 syscall 指令所在的地址，而不是其下一条指令的地址。所以如果不加 4，便会产生死循环。

```
NESTED(handle_int, 0, sp)
  // interrupt handler
  // Leve3 exception Handler.
  mfc0  a0, CP0_STATUS
  mfc0  a1, CP0_CAUSE
  addiu sp, sp, -8
  jal   interrupt_helper
  addiu sp, sp, 8

  RESTORE_CONTEXT(USER)

  STI
  eret
END(handle_int)
```

五、init_exception()函数

该函数的作用是例外的初始化。

首先需要关中断，然后将例外入口地址的代码 copy 到 0x80000180 地址处，然后刷新计时器，最后开终端。

其中 memcpy 函数需要用到要 copy 的代码的大小，并且其大小可以用 exception_handler_begin - exception_handler_end 得到，这样便可以很方便地计算出该部分代码的大小。另外因为最后要开中断，因此我将 init_exception 这一部分放在初始化的最后，不然很有可能其他初始化工作还没做完便出现了第一次时钟中断。

代码如下：

```
static void init_exception()
{
  // Get CP0_STATUS
  uint32_t cp0_status = get_cp0_status();

  // Disable all interrupt
  dis_interrupt();

  //initialize exception_handlers
  init_exception_handler();

  // Copy the level 2 exception handling code to 0x80000180
  memcpy(BEV0_EBASE+BEV0_OFFSET, exception_handler_begin, exception_handler_end-exception_handler_begin);

  // intialize CP0_STATUS & CP0_COUNT & CP0_COMPARE
  init_timer();
  init_cp0_status(STATUS_CU0 | cp0_status | 0x8001);
}
```

六、schedule()函数

该函数的主要任务是进行进程的调度，并且我将保存光标的操作放在了这里。并且每次还需要检查在 sleep 队列中的任务是否需要被唤醒。

其调度部分的流程是：将当前 current_running 指向任务的优先级降低一级，并放到对应的准备队列中。随后从优先级最高的准备队列开始，依次往下寻找第一个任务，并将其从准备队列中拿出，用 current_running 指向该任务的 PCB。

需要主义的是，如果当前进程正处在 BLOCKED 状态，那么不需要将该任务放到准备队列，直接调度下一个任务即可。代码如下：

```
void scheduler(void)
{
  //save the cursor
  current_running->cursor_x = screen_cursor_x;
  current_running->cursor_y = screen_cursor_y;

  check_sleeping();

  if(current_running == NULL || current_running->status != TASK_RUNNING)
    switch_current_running();
  else
  {
    push_to_ready_queue(current_running);
    switch_current_running();
  }

  //restore the cursor
  screen_cursor_x = current_running->cursor_x;
  screen_cursor_y = current_running->cursor_y;
}
```

七、check_sleeping()函数

该函数需要遍历 sleep 队列中的 PCB，检查每个任务是否过了 sleep_deadline，如果是，则唤醒该任务。并根据其优先级，将其放到对应的准备队列。

```
static void check_sleeping()
{
  pcb_t * PCB;
  pcb_t * next = sleep_queue.head;

  uint32_t current_time = get_timer();

  if(!queue_is_empty(&sleep_queue))
  {
    do{
      PCB = next;

    if(current_time >= PCB->sleep_deadline)
      {
        next = queue_remove(&sleep_queue, PCB);
        push_to_ready_queue(PCB);
      }
      else
        next = PCB->next;
    }
    while(next != NULL);
  }
}
```

八、irq_timer()函数

在该函数中还需要刷新 cp0_count 和 cp0_compare 寄存器的值，使其为新的任务分配时间片，并刷新 cp0_cause 寄存器。

```
static void irq_timer()
{
  // TODO clock interrupt handler.
  // scheduler, time counter in here to do, emmmmmm maybe.
  time_elapsed += 100000;
  screen_reflush();
  do_scheduler();
  init_timer();
};
```

九、sys_call_helper()函数

其功能是根据用户传入的参数调用相应的函数。其中 syscall 数组为各个系统调用函数的地址，在 init_syscall 中为其赋值。

```
void system_call_helper(int fn, int arg1, int arg2, int arg3)
{
  if(fn >= 0 && fn <NUM_SYSCALLS)
    syscall[fn](arg1, arg2, arg3);
  else
  {
    printk("UNKNOW SYSCALL\n");
    while(1);
  }
}
```

十、do_mutex_lock_acquire()函数和 do_mutex_lock_release()函数

其中 do_mutex_lock_acquire()函数的功能为申请锁，若申请的锁被占有，则将该任务当

道阻塞队列中。

do_mutex_lock_release()函数的任务是释放锁，并从阻塞队列中释放释放一个申请该锁的进程。

```
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) )
    lock->status = UNLOCKED;
  else
    do_unblock_one(&block_queue);
}
```

■