Lab5: RV64 内核线程调度

朱熙哲
3220103361

2023 年 12 月 24 日

目录

1　代码编写

1.1　基本调整

调整部分头文件的引用，按要求在 defs.h 中添加宏，注释掉上个实验对时钟中断信息打印的代码。添加 proc.h 头文件。

1.2　线程初始化

```c
// proc.c

void task_init(){
    // 1. 调用 kalloc() 为 idle 分配一个物理页
    idle = (struct task_struct*)kalloc();
    // 2. 设置 state 为 TASK_RUNNING;
    idle->state = TASK_RUNNING;
    // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
    idle->counter = 0;
    idle->priority = 0;
    // 4. 设置 idle 的 pid 为 0
    idle->pid = 0;
    // 5. 将 current 和 task[0] 指向 idle
    current = idle;
    task[0] = idle;
    // 1. 参考 idle 的设置, 为 task[1] ~ task[NR_TASKS - 1] 进行初始化
    // 2. 其中每个线程的 state 为 TASK_RUNNING, counter 为 0, priority 使用 rand() 来设置, pid 为该线程在线程数组中的下标。
```

```c
    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct` 中的 `ra` 和 `sp`,
    // 4. 其中 `ra` 设置为 __dummy （见 4.3.2) 的地址， `sp` 设置为 该线程申请的物理页的高地址
    for(int i = 1; i < NR_TASKS; i++){
        struct task_struct* task_i = (struct task_struct*)kalloc();
        task_i->state = TASK_RUNNING;
        task_i->counter = 0;
        task_i->priority = rand()%(PRIORITY_MAX - PRIORITY_MIN + 1) + PRIORITY_MIN;
        task_i->pid = i;
        task_i->thread.ra = (uint64)__dummy;
        task_i->thread.sp = (uint64)task_i + PGSIZE;
        task[i] = task_i;
    }
    printk("...proc_init done!\n");
    return;
}
```

依照要求一步步进行即可，注意 priority 的随机的方法，并且要把 ra 指向 __dummy，sp 指向物理内存的高地址，也就是当前 task_struct 加上一页 (4KB) 的大小。

在 head.S 中 _start 里添加对 task_init 和 mm_init 的调用，来初始化物理内存和线程。

```asm
_start:
  la sp, stack_top
  call mm_init
```

```
  call task_init
  ...
```

选择在开启时钟中断前进行初始化，避免在初始化时发生时钟中断，导致调度提前发生，进而可能产生段错误 (引用了尚未分配的 task 内存)。

1.3 添加 dummy 和 __dummy

依照要求在 proc.c 添加 dummy() 就好。在 entry.S 中添加 __dummy：

```
.extern dummy
.global __dummy
__dummy:
  la a0, dummy
  csrw sepc, a0
  sret
```

1.4 实现线程切换

判断下一个执行的线程 next 与当前的线程 current 是否为同一个线程，如果是同一个线程，则无需做任何处理，否则调用 __switch_to 进行线程切换，并打印切换信息。

```
// proc.c

extern void __switch_to(struct task_struct* prev, struct task_struct* next);

void switch_to(struct task_struct* next) {
```

```c
    if(next != current){
        printk("switch to [PID = %d, PRIORITY = %d, COUNTER = %d]\n", next->pid, next->priority, next->counter);
        struct task_struct* previous = current;
        current = next;
        __switch_to(previous, next);
    }
}
```

在 entry.S 中实现 __switch_to，注意 a0 接收 prev，a1 接收 next。根据 task_struct 的结构，有一个 uint64 的指针和 4 个 uint64 的值，然后才是 thread_struct 结构体。所以需要偏移 5*8=40 个 bytes。

```asm
.globl __switch_to
__switch_to:
  sd ra,40(a0)
  sd sp,48(a0)
  sd s0,56(a0)
  ...
  sd s11,144(a0)

  ld ra,40(a1)
  ld sp,48(a1)
  ld s0,56(a1)
  ...
  ld s11,144(a1)

  ret
```

1.5 实现调度入口函数

```c
// proc.c
void do_timer(void)
{
    /* 1. 将当前进程的 counter--, 如果结果大于零则直接返回 */
    /* 2. 否则进行进程调度 */
    if(current == idle || current->counter == 0){
        schedule();
    }
    else{
        current->counter--;
        if(!current->counter) schedule();
    }
}
```

首先判断是否是 counter 为 0 或者是 idle 线程，如果是，则直接调度。否则对当前线程的 counter 做减一操作，如果做完 counter 归零了，也要执行调度。

1.6 实现线程调度

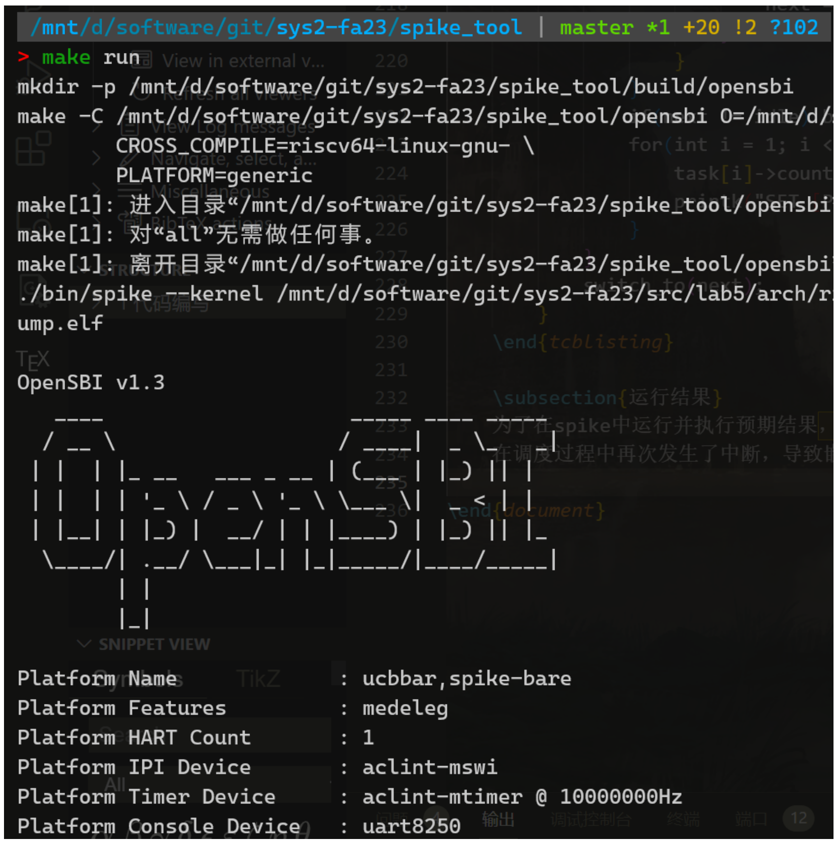
根据执行结果和参考实现，需要遍历线程，找到 counter 最小的线程并切换，如果 counter 均为零，则使用 priority 为所有线程的 counter 赋值并打印信息。

```c
// proc.c
void schedule(void)
{
    struct task_struct* next = idle;
```

```c
    while(1){
        uint64 counter_min = UINT64_MAX;
        for(int i = 1; i < NR_TASKS; i++){
            if(task[i]->state == TASK_RUNNING){
                if(task[i]->counter &&
                   task[i]->counter < counter_min)
                {
                    counter_min = task[i]->counter;
                    next = task[i];
                }
            }
        }
        if(next != idle) break;
        for(int i = 1; i < NR_TASKS; i++){
            task[i]->counter = task[i]->priority;
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]->pid, task[i]->priority, task[i]->counter);
        }
    }
    switch_to(next);
}
```

1.7 运行结果

为了在 spike 中运行并执行预期结果，可能需要调整增加时钟中断周期，或许是因为在调度过程中再次发生了中断，导致嵌套的发生，产生了不正常的线程切换，可能需要设置在处理中断时禁止中断。



spike 工具链

```
Boot HART MIDELEG          : 0x0000000000000222
Boot HART MEDELEG          : 0x000000000000b109
...mm_init done!
...proc_init done!
2022 ZJU Computer System II
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
SET [PID = 2 PRIORITY = 4 COUNTER = 4]
SET [PID = 3 PRIORITY = 5 COUNTER = 5]
switch to [PID = 1, PRIORITY = 1, COUNTER = 1]
[PID = 1] is running. auto_inc_local_var = 1
switch to [PID = 2, PRIORITY = 4, COUNTER = 4]
[PID = 2] is running. auto_inc_local_var = 1
[PID = 2] is running. auto_inc_local_var = 2
[PID = 2] is running. auto_inc_local_var = 3
[PID = 2] is running. auto_inc_local_var = 4
switch to [PID = 3, PRIORITY = 5, COUNTER = 5]
[PID = 3] is running. auto_inc_local_var = 1
[PID = 3] is running. auto_inc_local_var = 2
[PID = 3] is running. auto_inc_local_var = 3
[PID = 3] is running. auto_inc_local_var = 4
[PID = 3] is running. auto_inc_local_var = 5
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
SET [PID = 2 PRIORITY = 4 COUNTER = 4]
SET [PID = 3 PRIORITY = 5 COUNTER = 5]
switch to [PID = 1, PRIORITY = 1, COUNTER = 1]
[PID = 1] is running. auto_inc_local_var = 2
switch to [PID = 2, PRIORITY = 4, COUNTER = 4]
[PID = 2] is running. auto_inc_local_var = 5
[PID = 2] is running. auto_inc_local_var = 6
[PID = 2] is running. auto_inc_local_var = 7
[PID = 2] is running. auto_inc_local_var = 8
switch to [PID = 3, PRIORITY = 5, COUNTER = 5]
[PID = 3] is running. auto_inc_local_var = 6
[PID = 3] is running. auto_inc_local_var = 7
[PID = 3] is running. auto_inc_local_var = 8
[PID = 3] is running. auto_inc_local_var = 9
[PID = 3] is running. auto_inc_local_var = 10
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
SET [PID = 2 PRIORITY = 4 COUNTER = 4]
```

运行结果

2 思考题

2.1 为什么 context_switch 中只保存 14 个通用寄存器

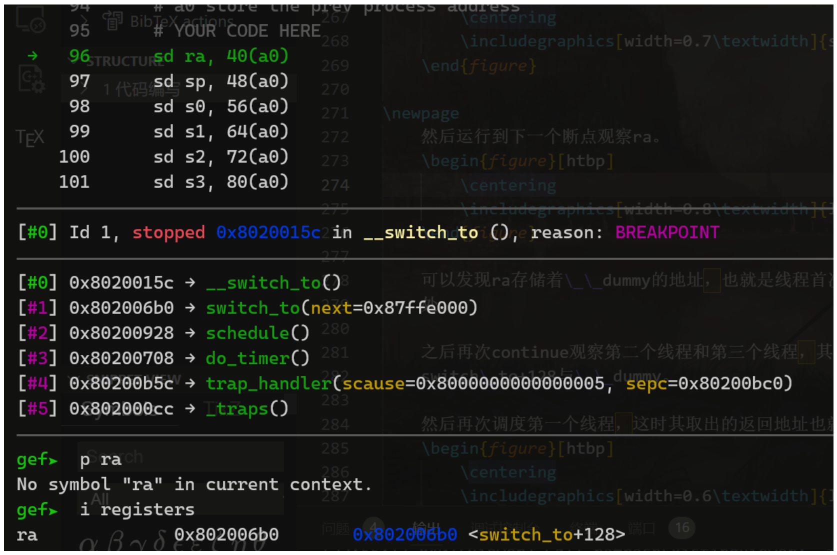
根据函数调用约定，在调用 __switch_to 函数时 caller-saved 寄存器会由汇编器主动保存到栈上，因此只需维护好 callee_saved 寄存器 sp、s0-s11 以及存储了返回地址的 ra 即可。

2.2 线程切换流程追踪，关注 ra 变化

首先 spike 开启调试，查看 __switch_to 的地址，并设置断点。

```
gef➤  x/16i __switch_to
   0x8020015c <__switch_to>:     sd      ra,40(a0)
   0x80200160 <__switch_to+4>:   sd      sp,48(a0)
   0x80200164 <__switch_to+8>:   sd      s0,56(a0)
   0x80200168 <__switch_to+12>:  sd      s1,64(a0)
   0x8020016c <__switch_to+16>:  sd      s2,72(a0)
   0x80200170 <__switch_to+20>:  sd      s3,80(a0)
   0x80200174 <__switch_to+24>:  sd      s4,88(a0)
   0x80200178 <__switch_to+28>:  sd      s5,96(a0)
   0x8020017c <__switch_to+32>:  sd      s6,104(a0)
   0x80200180 <__switch_to+36>:  sd      s7,112(a0)
   0x80200184 <__switch_to+40>:  sd      s8,120(a0)
   0x80200188 <__switch_to+44>:  sd      s9,128(a0)
   0x8020018c <__switch_to+48>:  sd      s10,136(a0)
   0x80200190 <__switch_to+52>:  sd      s11,144(a0)
   0x80200194 <__switch_to+56>:  ld      ra,40(a1)
   0x80200198 <__switch_to+60>:  ld      sp,48(a1)
gef➤  b * 0x8020015c
Breakpoint 1 at 0x8020015c: file entry.S, line 96.
gef➤  b * 0x80200194
Breakpoint 2 at 0x80200194: file entry.S, line 113.
gef➤  c
Continuing.
```

跳转到第一个断点观察 ra，此时是从 idle 切换到 thread1 的过程。



ra 中存储了 switch+128 的地址，也就是调用 ___switch_to 后的返回地址。

然后运行到下一个断点观察 ra。

可以发现 ra 存储着 __dummy 的地址，也就是线程首次被调度时，会返回到 __dummy 处。

之后再次 continue 观察第二个线程和第三个线程，其存储和取出的返回地址也都是 switch_to+128 与 __dummy。

然后再次调度第一个线程，这时其存储和取出的返回地址也就都是 switch_to+128 了。