

# Lab5: RV64 内核线程调度

朱熙哲

3220103361

2023 年 12 月 24 日

## 目录

<b>1</b>	<b>代码编写</b>	<b>2</b>
1.1	基本调整 . . . . .	2
1.2	线程初始化 . . . . .	2
1.3	添加 dummy 和 __dummy . . . . .	4
1.4	实现线程切换 . . . . .	4
1.5	实现调度入口函数 . . . . .	6
1.6	实现线程调度 . . . . .	6
1.7	运行结果 . . . . .	8
<b>2</b>	<b>思考题</b>	<b>10</b>
2.1	为什么 context_switch 中只保存 14 个通用寄存器 . . . . .	10
2.2	线程切换流程追踪, 关注 ra 变化 . . . . .	10

# 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

```
1 // proc.c
2
3 void task_init(){
4     // 1. 调用 kalloc() 为 idle 分配一个物理页
5     idle = (struct task_struct*)kalloc();
6     // 2. 设置 state 为 TASK_RUNNING;
7     idle->state = TASK_RUNNING;
8     // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置
9     ↪ 为 0
10    idle->counter = 0;
11    idle->priority = 0;
12    // 4. 设置 idle 的 pid 为 0
13    idle->pid = 0;
14    // 5. 将 current 和 task[0] 指向 idle
15    current = idle;
16    task[0] = idle;
17    // 1. 参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进
18    ↪ 行初始化
19    // 2. 其中每个线程的 state 为 TASK_RUNNING, counter 为 0,
20    ↪ priority 使用 rand() 来设置, pid 为该线程在线程数组中的
21    ↪ 下标。
```

```

18 // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct`
   ↳ 中的 `ra` 和 `sp`,
19 // 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
20 for(int i = 1; i < NR_TASKS; i++){
21     struct task_struct* task_i = (struct
   ↳ task_struct*)kalloc();
22     task_i->state = TASK_RUNNING;
23     task_i->counter = 0;
24     task_i->priority = rand()%(PRIORITY_MAX - PRIORITY_MIN
   ↳ + 1) + PRIORITY_MIN;
25     task_i->pid = i;
26     task_i->thread.ra = (uint64)__dummy;
27     task_i->thread.sp = (uint64)task_i + PGSIZE;
28     task[i] = task_i;
29 }
30 printk("...proc_init done!\n");
31 return;
32 }

```

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。

```

1 _start:
2     la sp, stack_top
3     call mm_init

```

```

4   call task_init
5   ...

```

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:

```

1  .extern dummy
2  .global __dummy
3  __dummy:
4      la a0, dummy
5      csrw sepc, a0
6      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。

```

1  // proc.c
2
3  extern void __switch_to(struct task_struct* prev, struct
    ↪ task_struct* next);
4
5  void switch_to(struct task_struct* next) {

```

```

6     if(next != current){
7         printk("switch to [PID = %d, PRIORITY = %d, COUNTER =
           ↪ %d]\n", next->pid, next->priority, next->counter);
8         struct task_struct* previous = current;
9         current = next;
10        __switch_to(previous, next);
11    }
12 }

```

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。

```

1  .globl __switch_to
2  __switch_to:
3      sd ra,40(a0)
4      sd sp,48(a0)
5      sd s0,56(a0)
6      ...
7      sd s11,144(a0)
8
9      ld ra,40(a1)
10     ld sp,48(a1)
11     ld s0,56(a1)
12     ...
13     ld s11,144(a1)
14
15     ret

```

## 1.5 实现调度入口函数

```
1 // proc.c
2 void do_timer(void)
3 {
4     /* 1. 将当前进程的 counter--, 如果结果大于零则直接返回 */
5     /* 2. 否则进行进程调度 */
6     if(current == idle || current->counter == 0){
7         schedule();
8     }
9     else{
10        current->counter--;
11        if(!current->counter) schedule();
12    }
13 }
```

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。

```
1 // proc.c
2 void schedule(void)
3 {
4     struct task_struct* next = idle;
```

```

5   while(1){
6       uint64 counter_min = UINT64_MAX;
7       for(int i = 1; i < NR_TASKS; i++){
8           if(task[i]->state == TASK_RUNNING){
9               if(task[i]->counter &&
10                  task[i]->counter < counter_min)
11               {
12                   counter_min = task[i]->counter;
13                   next = task[i];
14               }
15           }
16       }
17       if(next != idle) break;
18       for(int i = 1; i < NR_TASKS; i++){
19           task[i]->counter = task[i]->priority;
20           printk("SET [PID = %d PRIORITY = %d COUNTER =
↳ %d]\n", task[i]->pid, task[i]->priority,
↳ task[i]->counter);
21       }
22   }
23   switch_to(next);
24 }

```

## 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。

```

/mnt/d/software/git/sys2-fa23/spike_tool | master *1 +20 !2 ?102
> make run View in external v... 220 }
mkdir -p /mnt/d/software/git/sys2-fa23/spike_tool/build/opensbi
make -C /mnt/d/software/git/sys2-fa23/spike_tool/opensbi O=/mnt/d/
CROSS_COMPILE=riscv64-linux-gnu- \ for(int i = 1; i
PLATFORM=generic 224 task[i]->count
make[1]: 进入目录"/mnt/d/software/git/sys2-fa23/spike_tool/opensbi
make[1]: 对"all"无需做任何事。 226
make[1]: 离开目录"/mnt/d/software/git/sys2-fa23/spike_tool/opensbi
./bin/spike --kernel /mnt/d/software/git/sys2-fa23/src/lab5/arch/r
ump.elf 229
}
end{tblisting} 230
TeX 231
OpenSBI v1.3 232 subsection(运行结果)
为在spike中运行并执行预期结果，
主调过程中再次发生了中断，导致
end{document} 236
\end{document}
Platform Name : ucbbbar,spike-bare
Platform Features : medeleg
Platform HART Count : 1
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 10000000Hz
Platform Console Device : uart8250

```

spike 工具链



```

Boot HART_MIDELEG ... : 0x000000000000222
Boot HART_MELEG : 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>: 86 sd __dumpra, 40(a0)
0x80200160 <__switch_to+4>: 87 sd # sp, 48(a0)
0x80200164 <__switch_to+8>: 88 sd la s0, 56(a0)
0x80200168 <__switch_to+12>: 89 sd cs s1, 64(a0)
0x8020016c <__switch_to+16>: 90 sd sr s2, 72(a0)
0x80200170 <__switch_to+20>: 91 sd s3, 80(a0)
0x80200174 <__switch_to+24>: 92 sd __switch s4, 88(a0)
0x80200178 <__switch_to+28>: 93 sd # s5, 96(a0)
0x8020017c <__switch_to+32>: 94 sd s6, 104(a0)
0x80200180 <__switch_to+36>: 95 sd s7, 112(a0)
0x80200184 <__switch_to+40>: 96 sd s8, 120(a0)
0x80200188 <__switch_to+44>: 97 sd s9, 128(a0)
0x8020018c <__switch_to+48>: 98 sd s10, 136(a0)
0x80200190 <__switch_to+52>: 99 sd s11, 144(a0)
0x80200194 <__switch_to+56>: 100 ld ra, 40(a1)
0x80200198 <__switch_to+60>: 101 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:
102 sd s5, 96(a0)
103 sd s7, 112(a0)
104 sd s9, 128(a0)
105 sd s10, 136(a0)
106 sd s11, 144(a0)
```

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

```
94 # do store the prev process address
95 # YOUR CODE HERE
96 sd ra, 40(a0)
97 sd sp, 48(a0)
98 sd s0, 56(a0)
99 sd s1, 64(a0)
100 sd s2, 72(a0)
101 sd s3, 80(a0)

[ #0 ] Id 1, stopped 0x8020015c in __switch_to (), reason: BREAKPOINT

[ #0 ] 0x8020015c → __switch_to()
[ #1 ] 0x802006b0 → switch_to(next=0x87ffe000)
[ #2 ] 0x80200928 → schedule()
[ #3 ] 0x80200708 → do_timer()
[ #4 ] 0x80200b5c → trap_handler(scause=0x8000000000000005, sepc=0x80200bc0)
[ #5 ] 0x802000cc → traps()

gef> p ra
No symbol "ra" in current context.
gef> i registers
ra 0x802006b0 0x802006b0 <switch_to+128>
```

ra 中存储了 switch+128 的地址，也就是调用 \_\_switch\_to 后的返回地址。

```
gef> x/16i switch_to+100
0x80200694 <switch_to+100>: auipc a5,0x4
0x80200698 <switch_to+104>: addi a5,a5,-1668
0x8020069c <switch_to+108>: ld a4,-40(s0)
0x802006a0 <switch_to+112>: sd a4,0(a5)
0x802006a4 <switch_to+116>: ld a1,-40(s0)
0x802006a8 <switch_to+120>: ld a0,-24(s0)
0x802006ac <switch_to+124>: jal ra,0x8020015c <__switch_to>
0x802006b0 <switch_to+128>: nop
0x802006b4 <switch_to+132>: ld ra,40(sp)
0x802006b8 <switch_to+136>: ld s0,32(sp)
0x802006bc <switch_to+140>: addi sp,sp,48(a0)
0x802006c0 <switch_to+144>: ret
0x802006c4 <do_timer>: addi sp,sp,-16(a0)
0x802006c8 <do_timer+4>: sd ra,8(sp)
0x802006cc <do_timer+8>: sd s0,0(sp)
0x802006d0 <do_timer+12>: addi s0,sp,16(a0)
```

然后运行到下一个断点观察 ra。

```
113 M ld ra, 40(a1) 101 sd s3, 80(a0)
→ 114 C mm ld sp, 48(a1) 102 sd s4, 88(a0)
115 ld s0, 56(a1) 103 sd s5, 96(a0)
116 ld s1, 64(a1) 104 sd s6, 104(a0)
117 ld s2, 72(a1) 105 sd s7, 112(a0)
118 ld s3, 80(a1) 106 sd s8, 120(a0)
119 C sbi ld s4, 88(a1) 107 sd s9, 128(a0)
108 sd s10, 136(a0)

[#0] Id 1, stopped 0x80200198 in __switch_to (), reason: SINGLE STEP
[#0] 0x80200198 → __switch_to()
[#1] 0x8020014c → _traps()

gef> i r ra
ra > 大綱 0x8020014c 0x8020014c <__dummy>
```

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。

```
113 ld ra, 40(a1)
→ 114 CPU 调度 ld sp, 48(a1)
115 进程 ld s0, 56(a1)
116 ld s1, 64(a1)
117 系统 ld s2, 72(a1)
118 系统 ld s3, 80(a1)
119 系统 ld s4, 88(a1)

[#0] Id 1, stopped 0x80200198 in __switch_to (), reason: SINGLE STEP
[#0] 0x80200198 → __switch_to()
[#1] 0x802006b0 → switch_to(next=0x87ffe000)
[#2] 0x80200928 → schedule()
[#3] 0x8020073c → do_timer()
[#4] 0x80200b5c → trap_handler(scause=0x8000000000000005, sepc=0x802005f4)
[#5] 0x802000cc → _traps()

gef> i r ra
ra 备注相关 0x802006b0 0x802006b0 <switch_to+128>
```