Lab 2: RV64 内核线程调度

张志心 3210106357

1 实验目的

- 了解线程概念，并学习线程相关结构体，并实现线程的初始化功能。
- 了解如何使用时钟中断来实现线程的调度。
- 了解线程切换原理，并实现线程的切换。
- 掌握简单的线程调度算法，并完成两种简单调度算法的实现。

2 实验环境

- Environment in previous labs

3 实验步骤

3.1 线程调度功能实现

3.1.1 线程初始化

为线程分配一个 4kB 的物理页，将 `task_struct` 存放在该页的低地址部分，将线程的栈指针 `sp` 指向该页的高地址。

```
void task_init() {
    test_init(NR_TASKS);

    mm_init();
    idle = (struct task_struct *)kalloc();
    if(!idle) return;
    current = task[0] = idle;
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;

    for(int i = 1; i < NR_TASKS; ++i) {
        task[i] = (struct task_struct *)kalloc();
        if(!task[i]) return;
        task[i]->state = TASK_RUNNING;
        task[i]->counter = task_test_counter[i];
        task[i]->priority = task_test_priority[i];
        task[i]->pid = i;
        task[i]->thread.ra = (uint64)__dummy;
        task[i]->thread.sp = PGSIZE + (long)task[i];
    }

    printk("...proc_init done!\n");
}
```

3.1.2 在 `entry.S` 添加 `__dummy`

在`__dummy` 中将 sepc 设置为 `dummy()` 的地址，并使用 `sret` 从中断中返回。

```
    .globl __dummy
__dummy:
    la t0, dummy
    csrw sepc, t0
    sret
```

3.1.3 进程切换

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。

```
void switch_to(struct task_struct* next) {
    if(!current) return;
    if(!next) return;
    if(current->pid != next->pid) {
        struct task_struct *prev = current;
        current = next;
        __switch_to(prev, next);
    }
}
```

在 `entry.S` 中实现线程上下文切换 `__switch_to`:

```
.globl __switch_to
__switch_to:
    mv t0, a0
    mv t1, a1
    sd ra, 48(t0)
    sd sp, 56(t0)
    sd s0, 64(t0)
    sd s1, 72(t0)
    sd s2, 80(t0)
    sd s3, 88(t0)
    sd s4, 96(t0)
    sd s5, 104(t0)
    sd s6, 112(t0)
    sd s7, 120(t0)
    sd s8, 128(t0)
    sd s9, 136(t0)
    sd s10, 144(t0)
    sd s11, 152(t0)

    ld ra, 48(t1)
    ld sp, 56(t1)
    ld s0, 64(t1)
    ld s1, 72(t1)
    ld s2, 80(t1)
    ld s3, 88(t1)
    ld s4, 96(t1)
```

```
    ld s5, 104(t1)
    ld s6, 112(t1)
    ld s7, 120(t1)
    ld s8, 128(t1)
    ld s9, 136(t1)
    ld s10, 144(t1)
    ld s11, 152(t1)

    ret
```

3.1.4 调度入口函数

在时钟中断处理函数中调用。

```
void do_timer(void) {
    if(!current) return;
    if(current->pid == 0) {
        schedule();
    }
    else {
        if(current->counter > 0) --current->counter;
        if(current->counter == 0) {
            current->state = !TASK_RUNNING;
            schedule();
        }
    }
}
```

3.1.5 短作业优先调度算法

```
#ifdef DSJF
void schedule(void) {
    int next_id = -1;
    for(int i = 1; i < NR_TASKS; ++i) {
        if(task[i] && task[i]->counter > 0 && task[i]->state == TASK_RUNNING &&
            (next_id==-1 || task[i]->counter < task[next_id]->counter)) {
            next_id = i;
        }
    }
    if(~next_id) {
        printk("switch to [PID = %d COUNTER = %d]\n", next_id, task[next_id]-
>counter);
        switch_to(task[next_id]);
    } else {
        for(int i = 1; i < NR_TASKS; ++i) if(task[i]) {
            task[i]->state = TASK_RUNNING;
            task[i]->counter = rand();
            printk("SET [PID = %d COUNTER = %d]\n", i, task[i]->counter);
        }
        schedule();
    }
}
#endif
```

3.1.6 优先级调度算法

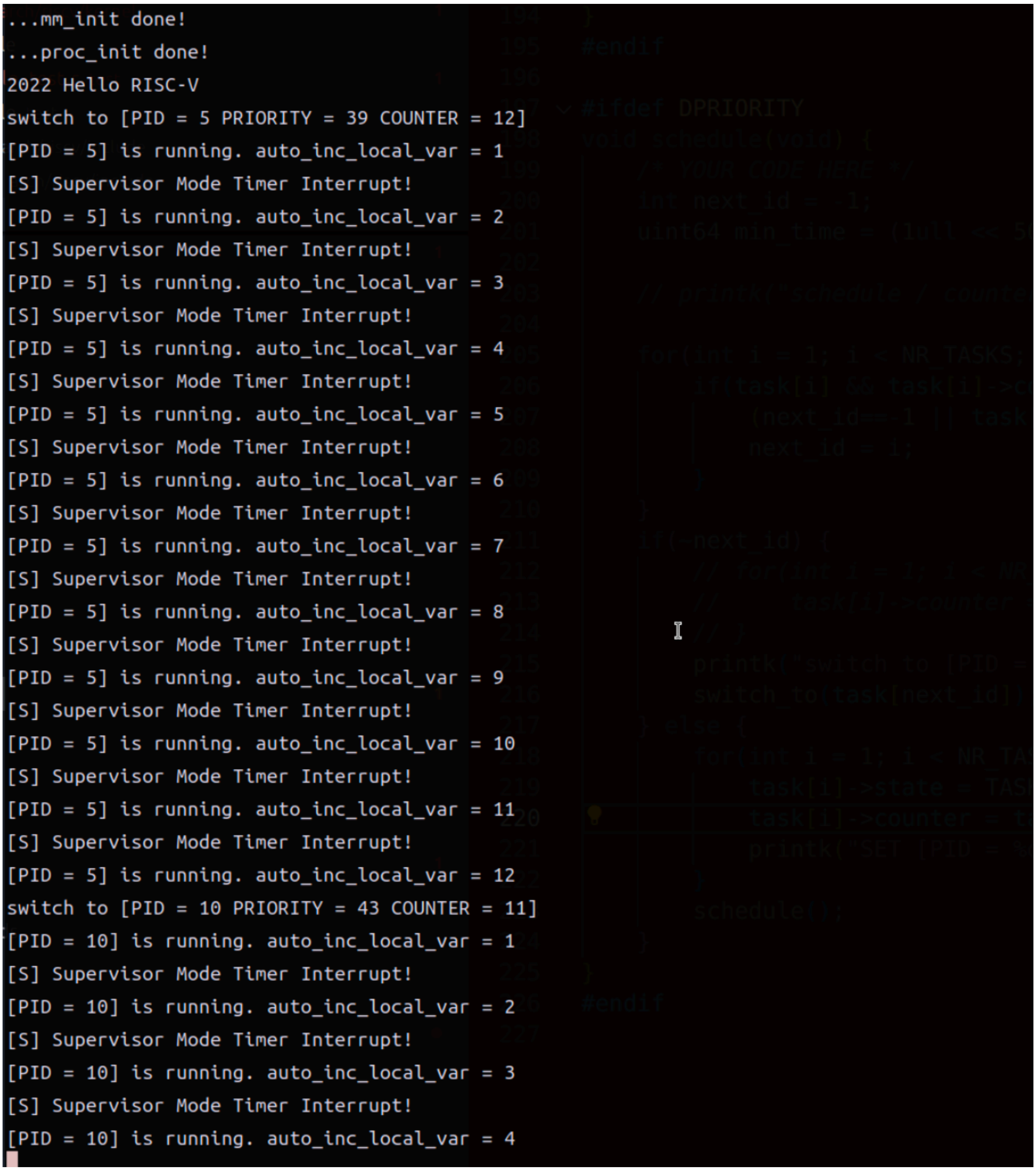
```
#ifdef DPRIORITY
void schedule(void) {
    int next_id = -1;
    for(int i = 1; i < NR_TASKS; ++i) {
        if(task[i] && task[i]->counter > 0 && task[i]->state == TASK_RUNNING &&
            (next_id==-1 || task[i]->counter >= task[next_id]->counter)) {
            next_id = i;
        }
    }
    if(~next_id) {
        printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n", next_id,
task[next_id]->priority, task[next_id]->counter);
        switch_to(task[next_id]);
    } else {
        for(int i = 1; i < NR_TASKS; ++i) if (task[i]) {
            task[i]->state = TASK_RUNNING;
            task[i]->counter = task[i]->priority + 1;
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", i, task[i]-
>priority, task[i]->counter);
        }
        schedule();
    }
}
#endif
```

3.2 运行效果

3.2.1 SJF 短作业优先调度

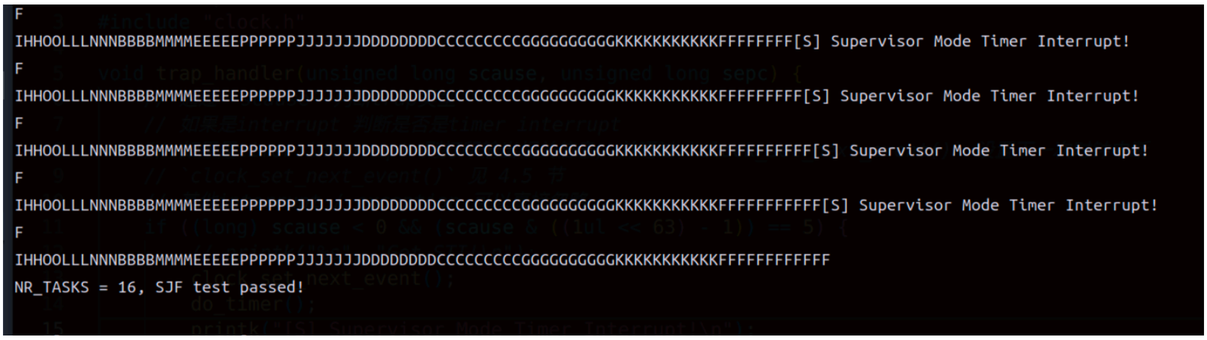
```
2022 Hello RISC-V
switch to [PID = 8 COUNTER = 1]
[PID = 8] is running. auto_inc_local_var = 1
switch to [PID = 7 COUNTER = 2]
[PID = 7] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 7] is running. auto_inc_local_var = 2
switch to [PID = 14 COUNTER = 2]
[PID = 14] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 14] is running. auto_inc_local_var = 2
switch to [PID = 11 COUNTER = 3]
[PID = 11] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 11] is running. auto_inc_local_var = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 11] is running. auto_inc_local_var = 3
switch to [PID = 13 COUNTER = 3]
[PID = 13] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 13] is running. auto_inc_local_var = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 13] is running. auto_inc_local_var = 3
switch to [PID = 1 COUNTER = 4]
[PID = 1] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. auto_inc_local_var = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. auto_inc_local_var = 3
[S] Supervisor Mode Timer Interrupt!
[PID = 1] is running. auto_inc_local_var = 4
switch to [PID = 12 COUNTER = 4]
[PID = 12] is running. auto_inc_local_var = 1
```

3.2.2 Priority 优先级调度

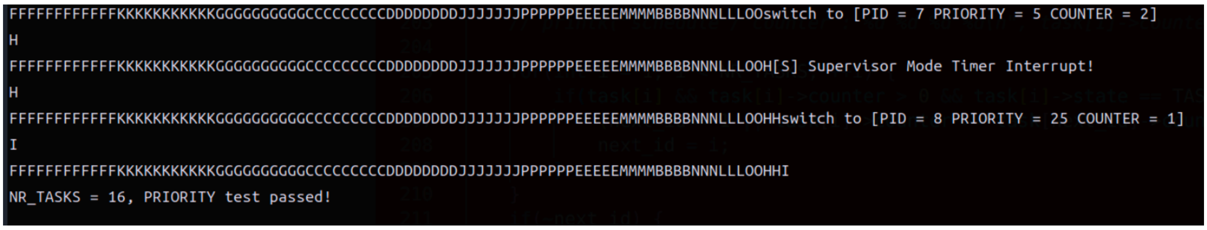


3.3 测试结果

3.3.1 SJF 短作业优先调度



3.3.2 Priority 优先级调度



3.4 思考题

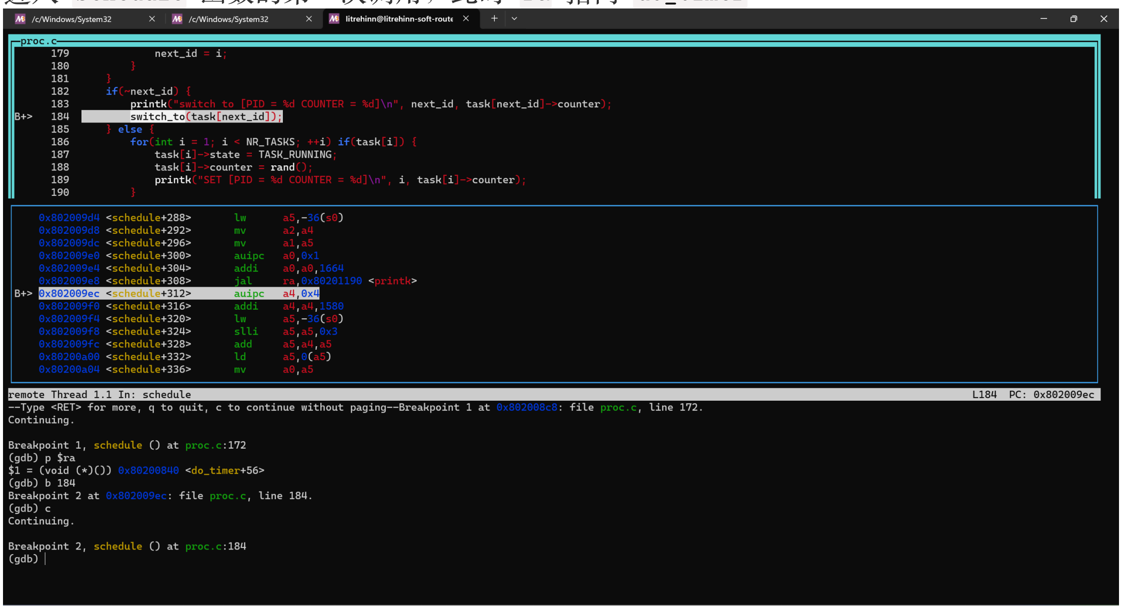
1. **在 RV64 中一共用 32 个通用寄存器，为什么 `context_switch` 中只保存了14个？**

   进行切换时，只需要保存 Callee Saved Register，因为 Caller Saved Register 会被 C 编译器保存在当前栈上，`context_switch` 函数只需要处理那些 C 编译器不会保存的寄存器。这样可以减少上下文切换的开销，提高性能。

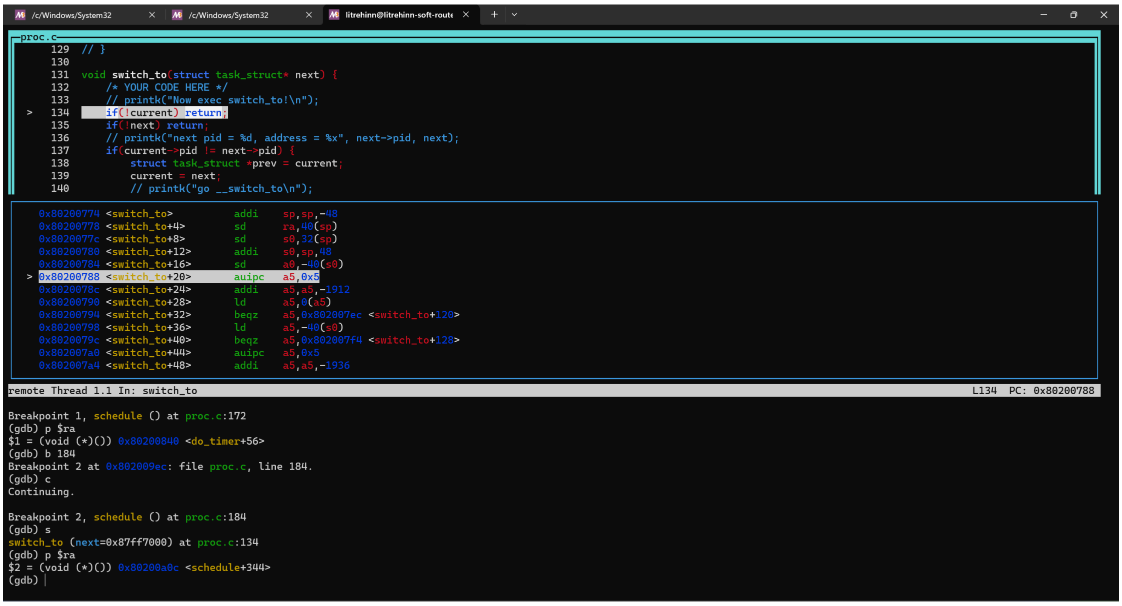
2. **当线程第一次调用时，其 `ra` 所代表的返回点是 `__dummy`。那么在之后的线程调用中 `context_switch` 中，`ra` 保存/恢复的函数返回点是什么呢？请同学用 gdb 尝试追踪一次完整的线程切换流程，并关注每一次 `ra` 的变换。**

   切换到一个还未被切换到的线程时：

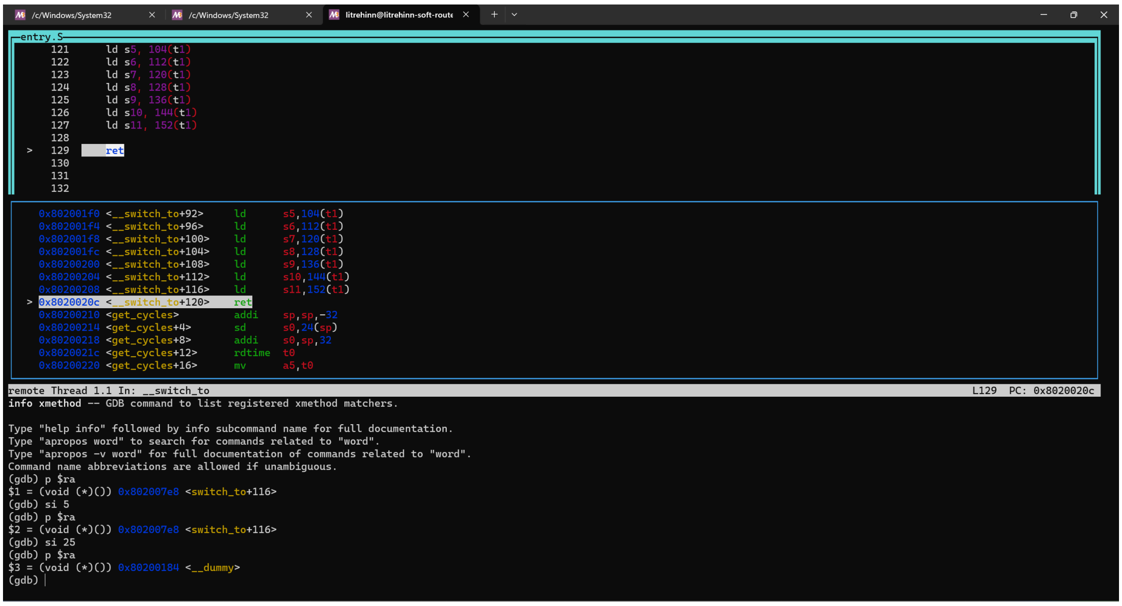
   进入 `schedule` 函数的第一次调用，此时 `ra` 指向 `do_timer`



进入 `switch_to`， `ra` 指向 `schedule`。

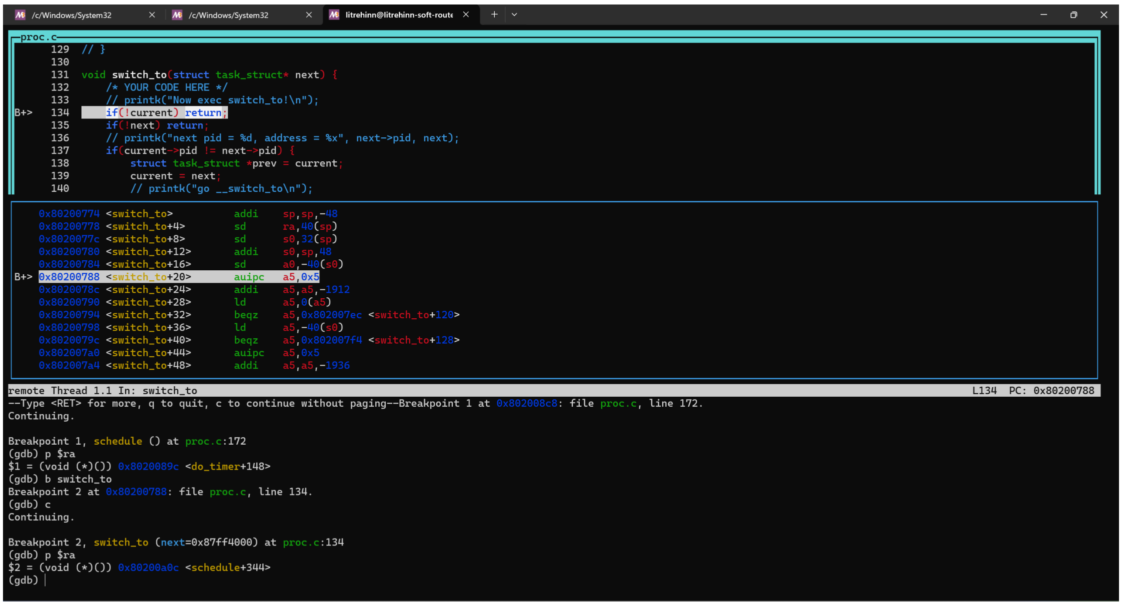


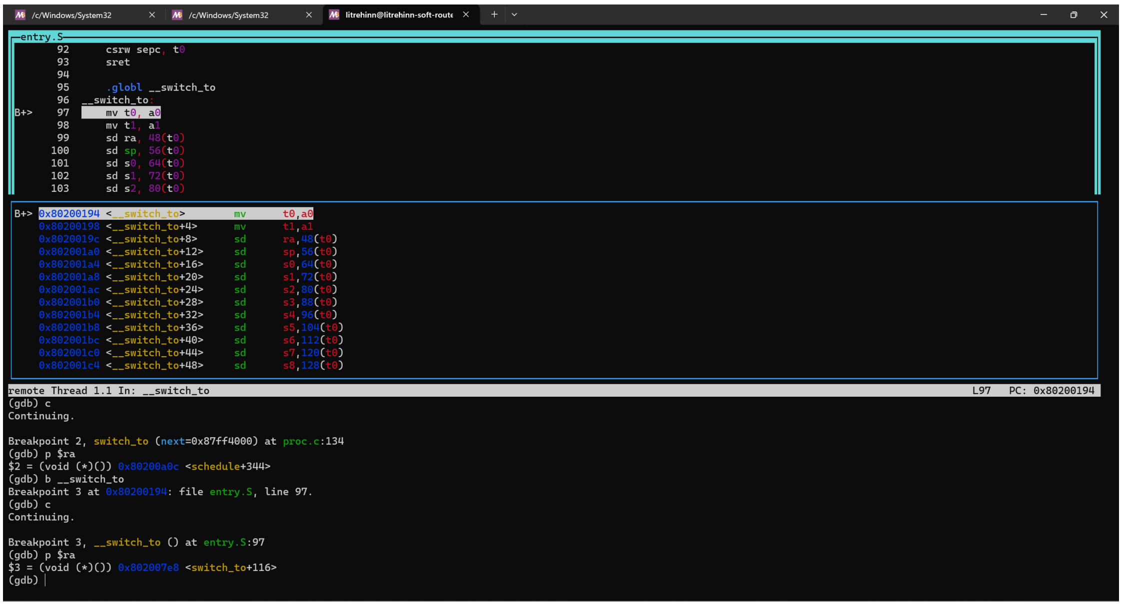
进入 `__switch_to`， `ra` 指向 `switch_to`，读取目标线程状态体的 `ra`，指向 `__dummy`，由于此时该线程没有运行，没有上下文结构体，于是跳到 `__dummy` 后让其帮助初始化后用更改 `sepc` 的方法跳至 `dummy`。



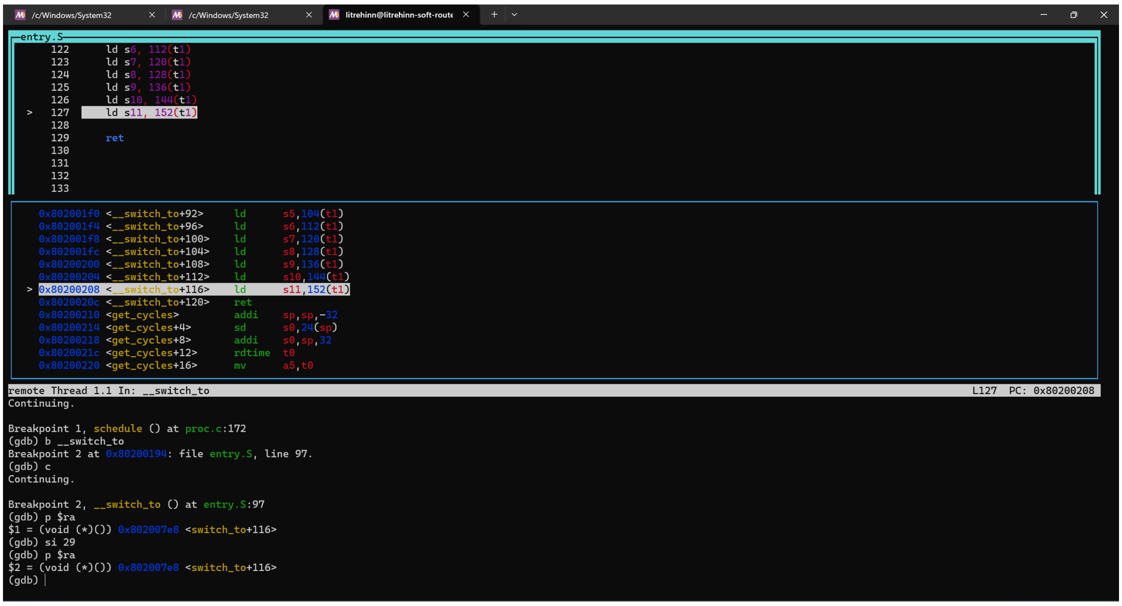
切换到一个已经运行过的线程时：

直到执行至 `__switch_to` 时，流程和上部分一致。

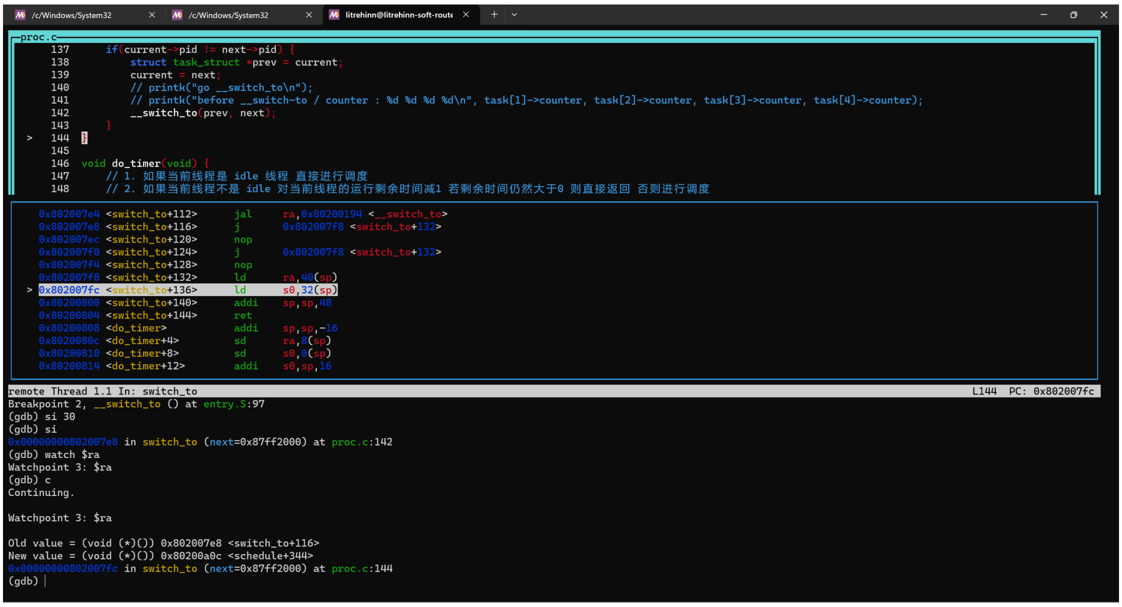




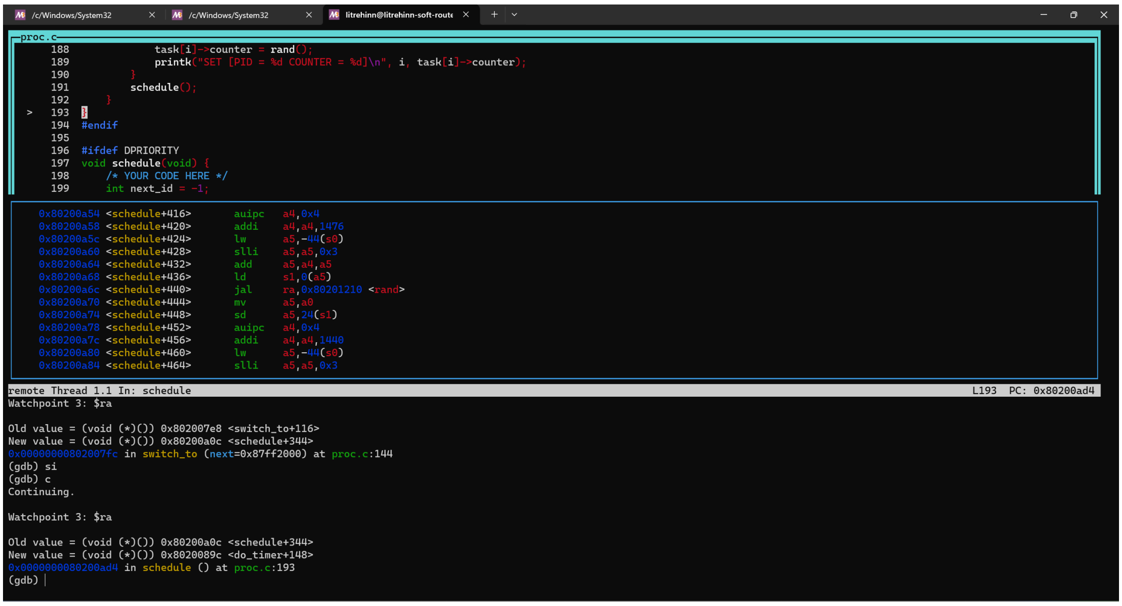
由于此时线程已经有上下文结构体，所以线程状态体的 `ra` 应该是正常指向 `switch_to` 的返回地址（即读取前后不改变，如图），但是栈已经切换到了目标线程的栈（`sp`），后续执行流接续目标线程执行切换前的执行流。



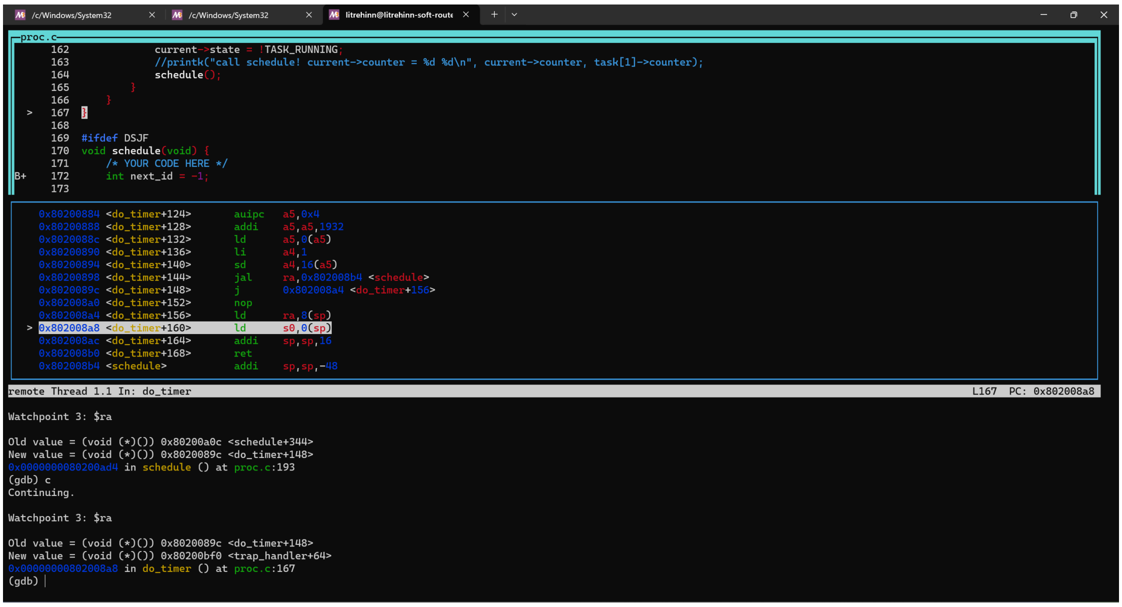
退出 `` `__switch_to ``，从栈上恢复 `ra`，指向 `schedule`



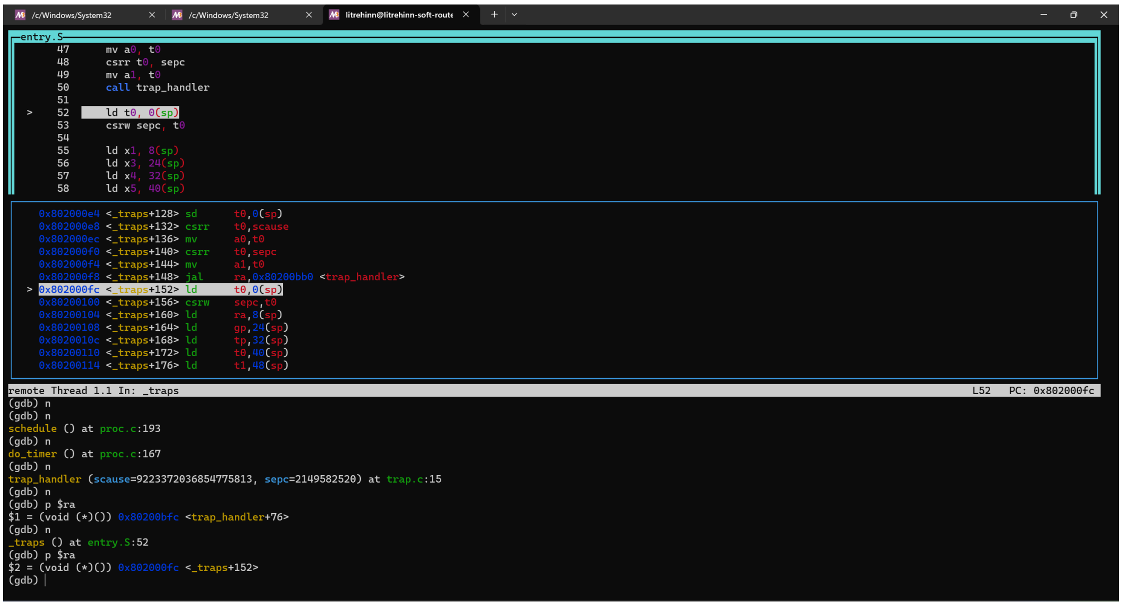
退出 `schedule`，从栈上恢复 `ra`，指向 `do_timer`



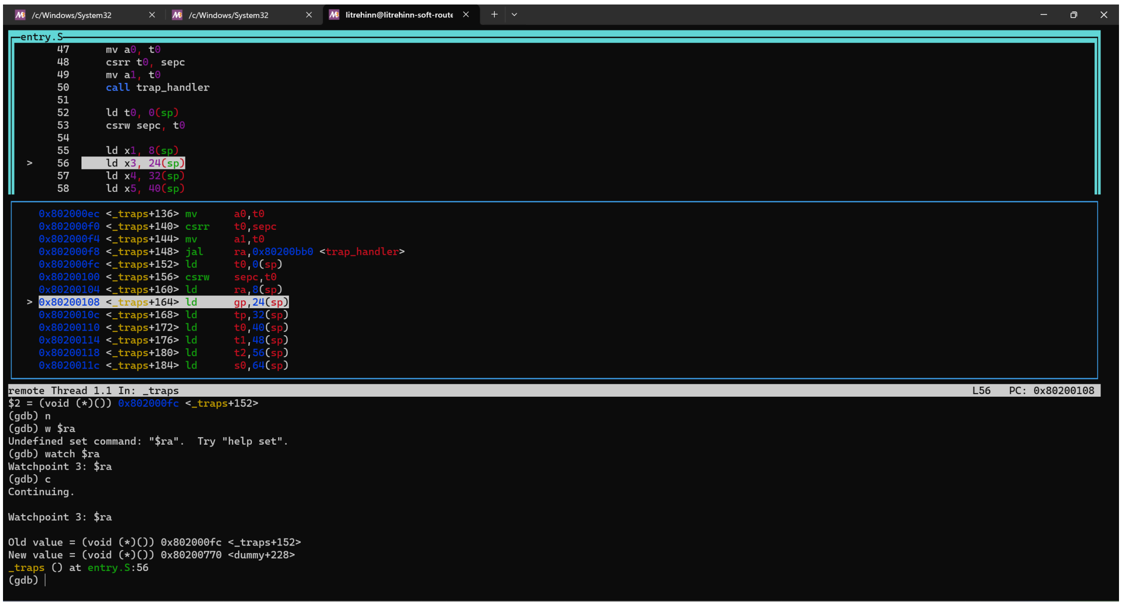
退出 `do_timer`，从栈上恢复 `ra`，指向 `trap_handler`



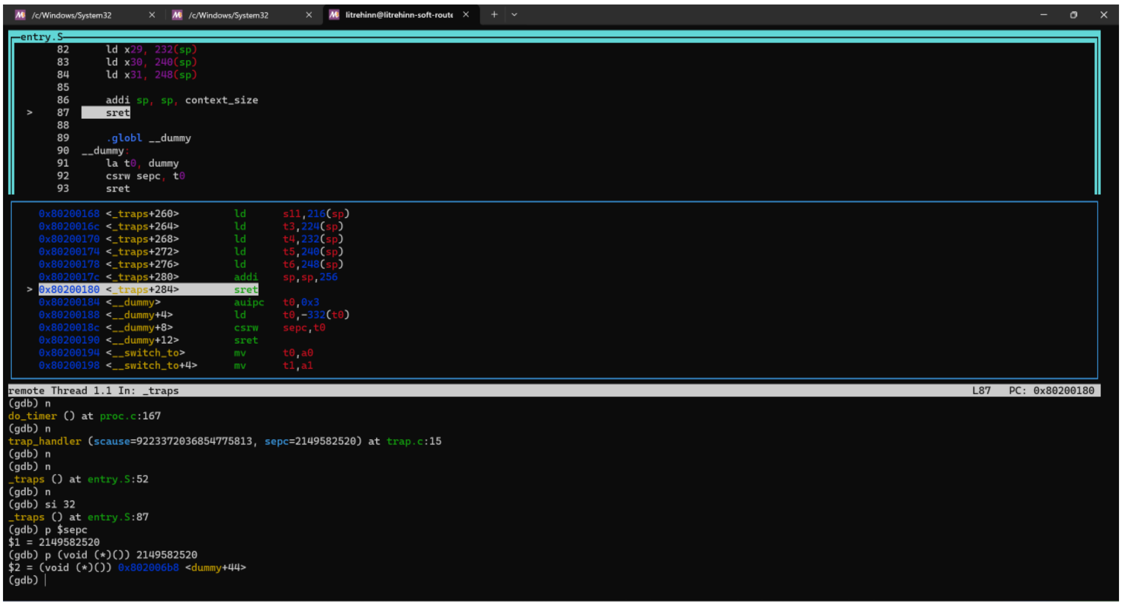
退出 `trap_handler`，从栈上恢复 `ra`，指向 `_traps`



从目标线程的栈上恢复 `ra`，指向目标线程触发中断前的 `ra`，即 `<dummy+228>`，实际要返回到的地址不是 `ra`。



可以看出，实际要返回到的地址是从上下文结构体恢复到 `sepc` 中的，即 `<dummy+44>`.


/c/Windows/System32
/c/Windows/System32
litrehinn@litrehinn-soft-route
entry.S
82 ld x29, 232(sp)
83 ld x30, 240(sp)
84 ld x31, 248(sp)
85
86 addi sp, sp, context_size
> 87 sret
88
89 .globl __dummy
90 __dummy:
91 la t0, dummy
92 csrw sepc, t0
93 sret
0x80200168 <_traps+260> ld s11,216(sp)
0x8020016c <_traps+264> ld t3,224(sp)
0x80200170 <_traps+268> ld t4,232(sp)
0x80200174 <_traps+272> ld t5,240(sp)
0x80200178 <_traps+276> ld t6,248(sp)
0x8020017c <_traps+280> addi sp,sp,256
> 0x80200180 <_traps+284> sret
0x80200184 <__dummy> auipc t0,0x3
0x80200188 <__dummy+4> ld t0,-332(t0)
0x8020018c <__dummy+8> csrw sepc,t0
0x80200190 <__dummy+12> sret
0x80200194 <__switch_to> mv t0,a0
0x80200198 <__switch_to+4> mv t1,a1
remote Thread 1.1 In: _traps L87 PC: 0x80200180
(gdb) n
do_timer () at proc.c:167
(gdb) n
trap_handler (scause=9223372036854775813, sepc=2149582520) at trap.c:15
(gdb) n
(gdb) n
_traps () at entry.S:52
(gdb) n
(gdb) si 32
_traps () at entry.S:87
(gdb) p $sepc
$1 = 2149582520
(gdb) p (void (*)()) 2149582520
$2 = (void (*)()) 0x802006b8 <dummy+44>
(gdb)