

Lab 2: RV64 内核线程调度

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

```

```

7  ISA=rv64imafd
8  ABI=lp64
9
10 INCLUDE = -I $(shell pwd)
11 CF = -march=$(ISA) -mabi=$(ABI)
12 CFLAG = ${CF} ${INCLUDE}
13
14 .PHONY:all TEST run debug
15 all:
16     ${MAKE} -C lib clean
17     ${MAKE} -C test clean
18     ${MAKE} -C init clean
19     ${MAKE} -C arch/riscv clean
20     $(shell test -f vmlinux) || \
21     $(shell test -f System.map) || \
22     @echo -e '\n'Clean F
23     ${MAKE} -C lib all
24     ${MAKE} -C test all
25     ${MAKE} -C init all
26     ${MAKE} -C arch/riscv all
27     @echo -e '\n'Build F
28
29 TEST:
30     ${MAKE} -C lib all
31     ${MAKE} -C test test
32     ${MAKE} -C init all
33     ${MAKE} -C arch/riscv all
34     @echo -e '\n'Build F
35
36 run: all
37     @echo Launch the qemu
38     @qemu-system-riscv64
39
40 test-run: TEST
41     @echo Launch the qemu
42     @qemu-system-riscv64

```

3.2.2 Priority 优先级调度

```

...mm_init done!
...proc_init done!
2022 Hello RISC-V
switch to [PID = 5 PRIORITY = 39 COUNTER = 12]
[PID = 5] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 3
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 4
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 5
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 6
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 7
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 8
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 9
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 10
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 11
[S] Supervisor Mode Timer Interrupt!
[PID = 5] is running. auto_inc_local_var = 12
switch to [PID = 10 PRIORITY = 43 COUNTER = 11]
[PID = 10] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt!
[PID = 10] is running. auto_inc_local_var = 2
[S] Supervisor Mode Timer Interrupt!
[PID = 10] is running. auto_inc_local_var = 3
[S] Supervisor Mode Timer Interrupt!
[PID = 10] is running. auto_inc_local_var = 4

```

```

194 }
195 #endif
196
197 #ifdef DPRIORITY
198 void schedule(void) {
199     /* YOUR CODE HERE */
200     int next_id = -1;
201     uint64 min_time = (1ull << 5);
202
203     // printk("schedule / counter\n");
204
205     for(int i = 1; i < NR_TASKS; i++) {
206         if(task[i] && task[i]->counter < min_time) {
207             (next_id == -1 || task[i]->priority < task[next_id]->priority) ? next_id = i : ;
208         }
209     }
210
211     if(-next_id) {
212         // for(int i = 1; i < NR_TASKS; i++) {
213         //     task[i]->counter = task[i]->priority;
214         // }
215         printk("switch to [PID = %d]\n", task[next_id]->pid);
216         switch_to(task[next_id]);
217     } else {
218         for(int i = 1; i < NR_TASKS; i++) {
219             task[i]->state = TASK_RUNNING;
220             task[i]->counter = task[i]->priority;
221             printk("SET [PID = %d] counter = %d\n", task[i]->pid, task[i]->counter);
222         }
223         schedule();
224     }
225 }
226 #endif

```

3.3 测试结果

3.3.1 SJF 短作业优先调度

```

F
IHOOALLNNBBBMMMMEEEEPPPPPPJJJJJJDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFF[S] Supervisor Mode Timer Interrupt!
F
void trap_handler(unsigned long cause, unsigned long sepc) {
IHOOALLNNBBBMMMMEEEEPPPPPPJJJJJJDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFF[S] Supervisor Mode Timer Interrupt!
F
void timer_interrupt_handler() {
IHOOALLNNBBBMMMMEEEEPPPPPPJJJJJJDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFF[S] Supervisor Mode Timer Interrupt!
F
void timer_interrupt_handler() {
IHOOALLNNBBBMMMMEEEEPPPPPPJJJJJJDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFF[S] Supervisor Mode Timer Interrupt!
F
while (1) {
    if (cause < 0) { // (cause < 0) == 03 == 1) == 57
IHOOALLNNBBBMMMMEEEEPPPPPPJJJJJJDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKFFFFFFFFFF
NR_TASKS = 16, SJF test passed!
next_event();
do_timer();
}
}
}

```

3.3.2 Priority 优先级调度

```

FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDJJJJJJPPPPPEEEEEMMMMBBBNNLLLOOswitch to [PID = 7 PRIORITY = 5 COUNTER = 2]
H
FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDJJJJJJPPPPPEEEEEMMMMBBBNNLLLOOH[S] Supervisor Mode Timer Interrupt!
H
FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDJJJJJJPPPPPEEEEEMMMMBBBNNLLLOOHswitch to [PID = 8 PRIORITY = 25 COUNTER = 1]
I
FFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDJJJJJJPPPPPEEEEEMMMMBBBNNLLLOOHI
NR_TASKS = 16, PRIORITY test passed!

```

3.4 思考题

1. 在 RV64 中共用 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`

```

proc.c
179     next_id = i;
180 }
181 }
182 if (!next_id) {
183     printk("switch to [PID = %d COUNTER = %d]\n", next_id, task[next_id]->counter);
184     switch_to(task[next_id]);
185 } else {
186     for (int i = 1; i < NR_TASKS; ++i) if (task[i]) {
187         task[i]->state = TASK_RUNNING;
188         task[i]->counter = rand();
189         printk("SET [PID = %d COUNTER = %d]\n", i, task[i]->counter);
190     }
}

0x802009d4 <schedule+288> lw a5,-36(a0)
0x802009d8 <schedule+292> mv a2,a5
0x802009dc <schedule+296> mv a1,a5
0x802009e0 <schedule+300> auipc a0,0x1
0x802009e4 <schedule+304> addi a0,a0,1664
0x802009e8 <schedule+308> jal ra,0x80201190 <printk>
0x802009ec <schedule+312> auipc a4,0x0
0x802009f0 <schedule+316> addi a0,a0,1580
0x802009f4 <schedule+320> lw a5,-36(a0)
0x802009f8 <schedule+324> slli a5,a5,0x3
0x802009fc <schedule+328> add a5,a4,a5
0x80200a00 <schedule+332> ld a5,(a3)
0x80200a04 <schedule+336> mv a0,a5

Remote Thread t1 in: schedule
--Type <RE> for more, q to quit, c to continue without paging--Breakpoint 1 at 0x80200808: file proc.c, line 172.
Continuing.

Breakpoint 1, schedule () at proc.c:172
(gdb) p $ra
$1 = (void (*)(void)) 0x80200808 <do_timer+56>
(gdb) b 184
Breakpoint 2 at 0x802009ec: file proc.c, line 184.
(gdb) c
Continuing.

Breakpoint 2, schedule () at proc.c:184
(gdb)

```

进入 `switch_to`，`ra` 指向 `schedule`。

```
proc.c
129 // }
130
131 void switch_to(struct task_struct* next) {
132     /* YOUR CODE HERE */
133     // printk("Now exec switch_to!\n");
134     if(!current) return;
135     if(!next) return;
136     // printk("next pid = %d, address = %x", next->pid, next);
137     if(current->pid != next->pid) {
138         struct task_struct *prev = current;
139         current = next;
140         // printk("go __switch_to\n");
141     }
142 }

0x80200774 <switch_to>      addi    sp,sp,-48
0x80200778 <switch_to+4>    sd      ra,48(sp)
0x8020077c <switch_to+8>    sd      s0,32(sp)
0x80200780 <switch_to+12>   addi    s0,s0,48
0x80200784 <switch_to+16>   sd      a0,-16(s0)
> 0x80200788 <switch_to+20> auipc    a5,0x5
0x8020078c <switch_to+24>   addi    a5,a5,-1912
0x80200790 <switch_to+28>   ld      a5,0(a5)
0x80200794 <switch_to+32>   beqz    a5,0x8020079c <switch_to+128>
0x80200798 <switch_to+36>   ld      a5,-48(s0)
0x8020079c <switch_to+40>   beqz    a5,0x802007f4 <switch_to+128>
0x802007a0 <switch_to+44>   auipc    a5,0x5
0x802007a4 <switch_to+48>   addi    a5,a5,-1936

remote Thread 1.1 In: switch_to
L134 PC: 0x80200788

(gdb) p $ra
$1 = (void (*)(void)) 0x80200840 <do_timer+56>
(gdb) b 184
Breakpoint 2 at 0x802009ec: file proc.c, line 184.
(gdb) c
Continuing.

Breakpoint 2, schedule () at proc.c:184
(gdb) s
switch_to (next=0x87ff7000) at proc.c:134
(gdb) p $ra
$2 = (void (*)(void)) 0x80200a0c <schedule+344>
(gdb) |
```

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`。

```
-entry.S
121 ld s5, 104(t1)
122 ld s6, 112(t1)
123 ld s7, 120(t1)
124 ld s8, 128(t1)
125 ld s9, 136(t1)
126 ld s10, 144(t1)
127 ld s11, 152(t1)
128
> 129 ret
130
131
132

0x802001f0 <__switch_to+92> ld      s5,104(t1)
0x802001f4 <__switch_to+96> ld      s6,112(t1)
0x802001f8 <__switch_to+100> ld      s7,120(t1)
0x802001fc <__switch_to+104> ld      s8,128(t1)
0x80200200 <__switch_to+108> ld      s9,136(t1)
0x80200204 <__switch_to+112> ld      s10,144(t1)
0x80200208 <__switch_to+116> ld      s11,152(t1)
> 0x8020020c <__switch_to+120> ret
0x80200210 <get_cycles>    addi    sp,sp,-32
0x80200214 <get_cycles+4>  sd      s0,24(sp)
0x80200218 <get_cycles+8>  addi    s0,s0,32
0x8020021c <get_cycles+12> rdtime   t0
0x80200220 <get_cycles+16> mv      a5,t0

remote Thread 1.1 In: __switch_to
L129 PC: 0x8020020c

Info xmethod -- GDB command to list registered xmethod matchers.

Type "help info" followed by info subcommand name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) p $ra
$1 = (void (*)(void)) 0x802007e8 <switch_to+116>
(gdb) si 5
(gdb) p $ra
$2 = (void (*)(void)) 0x802007e8 <switch_to+116>
(gdb) si 25
(gdb) p $ra
$3 = (void (*)(void)) 0x80200184 <__dummy>
(gdb) |
```

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。


```

proc.c
129 // }
130
131 void switch_to(struct task_struct* next) {
132     /* YOUR CODE HERE */
133     // printk("Now exit switch_to!\n");
134     if(!current) return;
135     if(!next) return;
136     // printk("next pid = %d, address = %x", next->pid, next);
137     if(current->pid != next->pid) {
138         struct task_struct* prev = current;
139         current = next;
140         // printk("go __switch_to!\n");
    }

0x80200774 <switch_to>      addi    sp,sp,-48
0x80200778 <switch_to+4>    sd      ra,40(sp)
0x8020077c <switch_to+8>    sd      s0,32(sp)
0x80200780 <switch_to+12>   addi    s0,sp,48
0x80200784 <switch_to+16>   sd      a0,-40(s0)
B> 0x80200788 <switch_to+20> auipc    a5,0x1912
0x8020078c <switch_to+24>   addi    a5,a5,-1912
0x80200790 <switch_to+28>   ld      a5,0(a5)
0x80200794 <switch_to+32>   beqz    a5,0x802007ec <switch_to+128>
0x80200798 <switch_to+36>   ld      a5,-40(s0)
0x8020079c <switch_to+40>   beqz    a5,0x802007f4 <switch_to+128>
0x802007a0 <switch_to+44>   auipc    a5,0x1915
0x802007a4 <switch_to+48>   addi    a5,a5,-1915

Remote Thread 1.1 In: switch_to
--Type <RET> for more, q to quit, c to continue without paging--Breakpoint 1 at 0x80200788: file proc.c, line 172.
Continuing.

Breakpoint 1, schedule () at proc.c:172
(gdb) p $ra
$1 = (void (*)(void)) 0x8020089c <do_timer+148>
(gdb) b switch_to
Breakpoint 2 at 0x80200788: file proc.c, line 134.
(gdb) c
Continuing.

Breakpoint 2, switch_to (next=0x87ff4000) at proc.c:134
(gdb) p $ra
$2 = (void (*)(void)) 0x80200a0c <schedu+344>
(gdb)

```

```

entry.S
92     csrr sepc, t0
93     sret
94
95     .globl __switch_to
96     __switch_to:
97     mv t0, a0
98     mv t1, a1
99     sd ra, 40(t0)
100    sd sp, 56(t0)
101    sd s0, 64(t0)
102    sd s1, 72(t0)
103    sd s2, 80(t0)

B> 0x80200194 <__switch_to> mv      t0,a0
0x80200198 <__switch_to+4> mv      t1,a1
0x8020019c <__switch_to+8> sd      ra,40(t0)
0x802001a0 <__switch_to+12> sd      sp,56(t0)
0x802001a4 <__switch_to+16> sd      s0,64(t0)
0x802001a8 <__switch_to+20> sd      s1,72(t0)
0x802001ac <__switch_to+24> sd      s2,80(t0)
0x802001b0 <__switch_to+28> sd      s3,88(t0)
0x802001b4 <__switch_to+32> sd      s4,96(t0)
0x802001b8 <__switch_to+36> sd      s5,104(t0)
0x802001bc <__switch_to+40> sd      s6,112(t0)
0x802001c0 <__switch_to+44> sd      s7,120(t0)
0x802001c4 <__switch_to+48> sd      s8,128(t0)

Remote Thread 1.1 In: __switch_to
(gdb) c
Continuing.

Breakpoint 2, switch_to (next=0x87ff4000) at proc.c:134
(gdb) p $ra
$2 = (void (*)(void)) 0x80200a0c <schedu+344>
(gdb) b __switch_to
Breakpoint 3 at 0x80200194: file entry.S, line 97.
(gdb) c
Continuing.

Breakpoint 3, __switch_to () at entry.S:97
(gdb) p $ra
$3 = (void (*)(void)) 0x802007e8 <switch_to+116>
(gdb)

```

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。

```

entry.S
122 ld s6, 112(t1)
123 ld s7, 120(t1)
124 ld s8, 128(t1)
125 ld s9, 136(t1)
126 ld s10, 144(t1)
> 127 ld s11, 152(t1)
128
129 ret
130
131
132
133

0x802001f0 <__switch_to+92> ld s5,104(t1)
0x802001f4 <__switch_to+96> ld s6,112(t1)
0x802001f8 <__switch_to+100> ld s7,120(t1)
0x802001fc <__switch_to+104> ld s8,128(t1)
0x80200200 <__switch_to+108> ld s9,136(t1)
0x80200204 <__switch_to+112> ld s10,144(t1)
> 0x80200208 <__switch_to+116> ld s11,152(t1)
0x8020020c <__switch_to+120> ret
0x80200210 <get_cycles> addi sp,sp,-32
0x80200214 <get_cycles+4> sd s0,24(sp)
0x80200218 <get_cycles+8> addi s0,sp,32
0x8020021c <get_cycles+12> rtime t0
0x80200220 <get_cycles+16> mv a5,t0

Remote Thread 1.1 In: __switch_to
Continuing.

Breakpoint 1, schedule () at proc.c:172
(gdb) b __switch_to
Breakpoint 2 at 0x80200194: file entry.S, line 97.
(gdb) c
Continuing.

Breakpoint 2, __switch_to () at entry.S:97
(gdb) p $ra
$1 = (void (*)(void)) 0x802007e8 <switch_to+116>
(gdb) si 29
(gdb) p $ra
$2 = (void (*)(void)) 0x802007e8 <switch_to+116>
(gdb) |

```

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

```

proc.c
137 if(current->pid != next->pid) {
138     struct task_struct *prev = current;
139     current = next;
140     // printk("go __switch_to\n");
141     // printk("before __switch_to / counter : %d %d %d %d\n", task[1]->counter, task[2]->counter, task[3]->counter, task[4]->counter);
142     __switch_to(prev, next);
143 }
> 144
145
146 void do_timer(void) {
147     // 1. 如果当前线程是 idle 线程 直接进行调度
148     // 2. 如果当前线程不是 idle 对当前线程的运行剩余时间减1 若剩余时间仍然大于0 则直接返回 否则进行调度
149 }

0x802007e4 <switch_to+112> jal ra,0x80200194 <__switch_to>
0x802007e8 <switch_to+116> j 0x802007f8 <switch_to+132>
0x802007ec <switch_to+120> nop
0x802007f0 <switch_to+124> j 0x802007f8 <switch_to+132>
0x802007f4 <switch_to+128> nop
0x802007f8 <switch_to+132> ld ra,40(sp)
> 0x802007fc <switch_to+136> ld s0,32(sp)
0x80200800 <switch_to+140> addi sp,sp,48
0x80200804 <switch_to+144> ret
0x80200808 <do_timer> addi sp,sp,-16
0x8020080c <do_timer+4> sd ra,0(sp)
0x80200810 <do_timer+8> sd s0,0(sp)
0x80200814 <do_timer+12> addi s0,sp,16

Remote Thread 1.1 In: switch_to
Breakpoint 2, __switch_to () at entry.S:97
(gdb) si 30
(gdb) si
0x8020080802007e8 in switch_to (next=0x87ff2000) at proc.c:142
(gdb) watch $ra
Watchpoint 3: $ra
(gdb) c
Continuing.

Watchpoint 3: $ra

Old value = (void (*)(void)) 0x802007e8 <switch_to+116>
New value = (void (*)(void)) 0x8020080c <schedule+340>
0x8020080802007fc in switch_to (next=0x87ff2000) at proc.c:144
(gdb) |

```

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

```

proc.c
188     task[i]--counter = rand();
189     printf("SET [PID = %d] COUNTER = %d\n", i, task[i]--counter);
190 }
191     schedule();
192 }
> 193 #endif
194 #endif
195
196 #ifdef DPCRIORITY
197 void schedule(void) {
198     /* YOUR CODE HERE */
199     int next_id = -1;

0x80200a54 <schedule+416> auipc a4,0x4
0x80200a58 <schedule+420> addi a4,a4,1076
0x80200a5c <schedule+424> lw a5,-4(a4)
0x80200a60 <schedule+428> slli a5,a5,0x3
0x80200a64 <schedule+432> add a5,a4,a5
0x80200a68 <schedule+436> ld s1,0(a5)
0x80200a6c <schedule+440> jal ra,0x80201210 <rand>
0x80200a70 <schedule+444> mv a5,a0
0x80200a74 <schedule+448> sd a5,24(s1)
0x80200a78 <schedule+452> auipc a4,0x4
0x80200a7c <schedule+456> addi a4,a4,1080
0x80200a80 <schedule+460> lw a5,-4(a4)
0x80200a84 <schedule+464> slli a5,a5,0x3

Remote Thread 1.1 In: schedule
Watchpoint 3: $ra

Old value = (void (*)(void)) 0x802007e8 <switch_to+116>
New value = (void (*)(void)) 0x80200a0c <schedule+344>
0x0000000000802007fc in switch_to (next=0x87ff2000) at proc.c:144
(gdb) si
(gdb) c
Continuing.

Watchpoint 3: $ra

Old value = (void (*)(void)) 0x80200a0c <schedule+344>
New value = (void (*)(void)) 0x8020089c <do_timer+148>
0x000000000080200ad4 in schedule () at proc.c:193
(gdb) |

```

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

```

proc.c
162     current->state = TASK_RUNNING;
163     //printf("call schedule! current->counter = %d\n", current->counter, task[i]--counter);
164     schedule();
165 }
166 }
> 167 #endif
168 #endif
169 #ifdef DSJF
170 void schedule(void) {
171     /* YOUR CODE HERE */
172     int next_id = -1;
173 }

0x80200884 <do_timer+124> auipc a5,0x4
0x80200888 <do_timer+128> addi a5,a5,1932
0x8020088c <do_timer+132> ld a5,0(a5)
0x80200890 <do_timer+136> li a4,1
0x80200894 <do_timer+140> sd a4,16(a5)
0x80200898 <do_timer+144> jal ra,0x802008b4 <schedule>
0x8020089c <do_timer+148> j 0x802008a4 <do_timer+156>
0x802008a0 <do_timer+152> nop
0x802008a4 <do_timer+156> ld ra,0(sp)
> 0x802008a8 <do_timer+160> ld s0,0(sp)
0x802008ac <do_timer+164> addi sp,sp,16
0x802008b0 <do_timer+168> ret
0x802008b4 <schedule> addi sp,sp,-48

Remote Thread 1.1 In: do_timer
Watchpoint 3: $ra

Old value = (void (*)(void)) 0x80200a0c <schedule+344>
New value = (void (*)(void)) 0x8020089c <do_timer+148>
0x000000000080200ad4 in schedule () at proc.c:193
(gdb) c
Continuing.

Watchpoint 3: $ra

Old value = (void (*)(void)) 0x8020089c <do_timer+148>
New value = (void (*)(void)) 0x802008f0 <trap_handler+64>
0x0000000000802008a8 in do_timer () at proc.c:167
(gdb) |

```

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

```

entry.S
47      mv a0, t0
48      csrr t0, sepc
49      mv a1, t0
50      call trap_handler
51
> 52      ld t0, 0(sp)
53      csrrw sepc, t0
54
55      ld x1, 8(sp)
56      ld x3, 24(sp)
57      ld x4, 32(sp)
58      ld x5, 40(sp)

0x802000e4 <_traps+128> sd      t0, 0(sp)
0x802000e8 <_traps+132> csrr    t0, scause
0x802000ec <_traps+136> mv      a0, t0
0x802000f0 <_traps+140> csrr    t0, sepc
0x802000f4 <_traps+144> mv      a1, t0
0x802000f8 <_traps+148> jal     ra, 0x80200bb0 <trap_handler>
> 0x802000fc <_traps+152> ld      t0, 0(sp)
0x80200100 <_traps+156> csrrw   sepc, t0
0x80200104 <_traps+160> ld      ra, 8(sp)
0x80200108 <_traps+164> ld      gp, 24(sp)
0x8020010c <_traps+168> ld      tp, 32(sp)
0x80200110 <_traps+172> ld      t0, 40(sp)
0x80200114 <_traps+176> ld      t1, 48(sp)

Remote Thread 1.1 In: _traps
(gdb) n
(gdb) n
schedule () at proc.c:193
(gdb) n
do_timer () at proc.c:167
(gdb) n
trap_handler (scause=92233772036854775813, sepc=2149582520) at trap.c:15
(gdb) n
$1 = (void (*)(void)) 0x80200bfc <trap_handler+76>
(gdb) n
_trap () at entry.S:52
(gdb) p $ra
$2 = (void (*)(void)) 0x802000fc <_traps+152>
(gdb)

```

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。

```

entry.S
47      mv a0, t0
48      csrr t0, sepc
49      mv a1, t0
50      call trap_handler
51
> 52      ld t0, 0(sp)
53      csrrw sepc, t0
54
55      ld x1, 8(sp)
56      ld x3, 24(sp)
57      ld x4, 32(sp)
58      ld x5, 40(sp)

0x802000ec <_traps+136> mv      a0, t0
0x802000f0 <_traps+140> csrr    t0, sepc
0x802000f4 <_traps+144> mv      a1, t0
0x802000f8 <_traps+148> jal     ra, 0x80200bb0 <trap_handler>
0x802000fc <_traps+152> ld      t0, 0(sp)
0x80200100 <_traps+156> csrrw   sepc, t0
0x80200104 <_traps+160> ld      ra, 8(sp)
> 0x80200108 <_traps+164> ld      gp, 24(sp)
0x8020010c <_traps+168> ld      tp, 32(sp)
0x80200110 <_traps+172> ld      t0, 40(sp)
0x80200114 <_traps+176> ld      t1, 48(sp)
0x80200118 <_traps+180> ld      t2, 56(sp)
0x8020011c <_traps+184> ld      s0, 64(sp)

Remote Thread 1.1 In: _traps
$2 = (void (*)(void)) 0x802000fc <_traps+152>
(gdb) n
(gdb) w $ra
Undefined set command: "$ra". Try "help set".
(gdb) watch $ra
Watchpoint 3: $ra
(gdb) c
Continuing.

Watchpoint 3: $ra

Old value = (void (*)(void)) 0x802000fc <_traps+152>
New value = (void (*)(void)) 0x80200770 <dummy+228>
_trap () at entry.S:56
(gdb)

```

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>。

```
entry.S
82 ld x20, 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 csw sepc, t0
93 sret

0x80200168 <_traps+268> ld a11,216(sp)
0x8020016c <_traps+268> ld t3,t24(sp)
0x80200170 <_traps+268> ld t4,t20(sp)
0x80200174 <_traps+272> ld t5,t48(sp)
0x80200178 <_traps+276> ld t6,t48(sp)
0x8020017c <_traps+280> addi sp,sp,256
> 0x80200180 <_traps+284> sret
0x80200184 <__dummy> swlpc t0,0x3
0x80200188 <__dummy+4> ld t0,-332(t0)
0x8020018c <__dummy+8> csw 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
90_timer () at proc.c:167
(gdb) n
trap_handler (scause=9223372036854775813, sepc=2149582520) at trap.c:15
(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 (*)(void)) 2149582520
$2 = (void (*)(void)) 0x802006b8 <dummy+44>
(gdb) |
```