

浙江大学

本科实验报告

课程名称：操作系统

姓 名：徐文皓

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：软件工程

学 号：3210102377

指导教师：夏莹杰

2023 年 11 月 04 日

浙江大学操作系统实验报告

实验名称：_____RV64 内核线程调度_____

电子邮件地址：_____手机：_____

实验地点：_____线上_____实验日期：2023 年 11 月 04 日


一、实验目的和要求

本实验的目的有：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。

本实验的要求是：独立完成作业；在 lab1 的基础上，完成初始化工作，主要为分配内存、设置 task_struct、和补全__dummy；实现线程切换_switch_to；调度函数，实现 do_timer；实现具体调度算法：短作业优先调度算法和优先级调度算法。

二、实验过程

将本实验新增的文件添加到相应位置，修改 arch/riscv/Makefile 第 3 行的链接路径，增加“../test/*.o”。使用“make run”命令启动内核，与 lab1 现象相同。



```
1 all:
2     ${MAKE} -C kernel all
3     ${LD} -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o ../../test/*.o -o ../../vmlinux
4     $(shell test -d boot || mkdir -p boot)
5     ${OBJCOPY} -O binary ../../vmlinux ./boot/Image
6     nm ../../vmlinux > ../../System.map
7
8 clean:
9     ${MAKE} -C kernel clean
10    $(shell test -d boot && rm -rf boot)
11
```

完善 task_init(), 对 idle(task[0])和 task[1]~task[NR_TASK-1]进行初始化。为方便测试，我们在任务初始化时输出任务总数 NR_TASKS，在初始化任务时以 SET [pid counter priority]格式输出相关信息。

```

1 void task_init() {
2     printk("NR_TASKS = ", NR_TASKS);
3     test_init(NR_TASKS);
4     // 对idle进行初始化
5     idle = (struct task_struct*)kalloc(); // 1. 调用 kalloc() 为 idle 分配一个物理页
6     idle->state = TASK_RUNNING;          // 2. 设置 state 为 TASK_RUNNING;
7     idle->counter = 0;                    // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
8     idle->priority = 0;
9     idle->pid = 0;                        // 4. 设置 idle 的 pid 为 0
10    current = idle;                      // 5. 将 current 和 task[0] 指向 idle
11    task[0] = idle;
12
13    for (int i = 1; i < NR_TASKS; i++){ // 对task[1]~task[NR_TASKS-1]进行初始化
14        task[i] = (struct task_struct*)kalloc(); // 1. 参考 idle 的设置, 为 task[1] ~ task[NR_TASKS - 1] 进行初始化
15        task[i]->pid = i;
16        task[i]->state = TASK_RUNNING;          // 2. 其中每个线程的 state 为 TASK_RUNNING, 此外, 为了单元测试的需要,
17        task[i]->counter = task_test_counter[i]; // counter 和 priority 进行如下赋值:
18        task[i]->priority = task_test_priority[i];
19        task[i]->thread.ra = (uint64)__dummy;    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 'thread_struct' 中的 'ra' 和 'sp',
20        task[i]->thread.sp = (uint64)task[i]+PGSIZE; // 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
21
22        printk("\nSET [pid=%d counter=%d priority=%d]\n", task[i]->pid, task[i]->counter, task[i]->priority);
23    }
24    printk("...proc_init done!\n");
25 }

```

补充 head.S, 在主程序开始前调用 mm_init()和 task_init(), 实现内存管理接口和任务的初始化。

```

1 _start:
2     la sp, boot_stack_top
3     call mm_init
4     call task_init
5
6     ...
7     # set the first time interrupt
8     ...
9
10    call start_kernel

```

当线程在运行时, 由于时钟中断的触发, 会将当前运行线程的上下文环境保存在栈上。当线程再次被调度时, 会将上下文从栈上恢复, 但是当我们创建一个新的线程, 此时线程的栈为空, 当这个线程被调度时, 是没有上下文需要被恢复的, 所以我们为线程第一次调度提供一个特殊的返回函数__dummy。在 entry.S 中添加__dummy 段和__dummy 函数, 用于不作上下文处理地将 sepc 设置为 dummy()的地址并直接结束中断。

```

1     .global __dummy
2     __dummy:
3     la a1, dummy
4     cswr sepc, a1
5     sret

```

完善 switch_to(), 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。如果两者是同一个线程, 则无需做任何处理, 否则调用__switch_to 进行线程切换。

```

1 void switch_to(struct task_struct* next) {
2     if(current!=next){
3         struct task_struct *prev = current;
4         current = next;
5         printk("\nSWITCH TO [pid=%d counter=%d priority=%d]\n", next->pid, next->counter, next->priority);
6         __switch_to(prev, next);
7     }
8 }

```

`__switch_to` 在 `entry.S` 中实现，将当前进程的 `ra`、`sp` 以及 `s0~s11` 保存到其 `thread` 成员上，并将这些寄存器从下一个进程的 `thread` 成员上读出。考虑到 `struct task_struct` 的定义，在 `thread` 之前定义了 7 个 `uint64` 类型成员，因此 `thread` 中各成员的地址分布从 `a0+48` 开始。

```
1  .globl __switch_to
2  __switch_to:
3      # save state to prev process
4      sd ra, 48(a0)
5      sd sp, 56(a0)
6      sd s0, 64(a0)
7      sd s1, 72(a0)
8      sd s2, 80(a0)
9      sd s3, 88(a0)
10     sd s4, 96(a0)
11     sd s5, 104(a0)
12     sd s6, 112(a0)
13     sd s7, 120(a0)
14     sd s8, 128(a0)
15     sd s9, 136(a0)
16     sd s10, 144(a0)
17     sd s11, 152(a0)
18
19     # restore state from next process
20     ld ra, 48(a1)
21     ld sp, 56(a1)
22     ld s0, 64(a1)
23     ld s1, 72(a1)
24     ld s2, 80(a1)
25     ld s3, 88(a1)
26     ld s4, 96(a1)
27     ld s5, 104(a1)
28     ld s6, 112(a1)
29     ld s7, 120(a1)
30     ld s8, 128(a1)
31     ld s9, 136(a1)
32     ld s10, 144(a1)
33     ld s11, 152(a1)
34
35     ret
```

实现调度入口函数 `do_timer()`，并在时钟中断函数 `trap_handler()` 中调用。

```
1 void do_timer(void) {
2     if(current==task[0]) // 1. 如果当前线程是 idle 线程 直接进行调度
3         schedule();
4     else{ // 2. 如果当前线程不是 idle
5         if(current->counter>0) // 对当前线程的运行剩余时间减1
6             current->counter--;
7         if(current->counter==0) // 若剩余时间仍然大于0 则直接返回 否则进行调度
8             schedule();
9     }
10 }
```

```
1 void trap_handler(unsigned long scause, unsigned long sepc){
2     unsigned long interrupt = scause & 0x8000000000000000;
3     unsigned long cause = scause & 0x7fffffffffffffff;
4     if(interrupt&&cause==5){ // timer interrupt
5         printk("[S] Supervisor Mode Timer Interrupt\n");
6         clock_set_next_event();
7         do_timer();
8     }
9     else{
10         printk("[Error] Not Timer Interrupt with scause = 0x%16x\n", scause);
11     }
12     return;
13 }
```

根据 `SJF` 和 `PRIORITY` 的宏定义情况，引入条件编译机制，实现短作业优先调度算法和优先级调度算法。算法的总体策略是，根据某种特定标准（如剩余时间、优先级）选择状态为 `TASK_RUNNING` 的剩余时间不为 0 的进程，如果选择到这样的进程则进行切换，否则为进程补充剩余时间后重新进行调度。

```

1 #ifdef SJF
2 void schedule(void) {
3     int minIndex;
4     while (1){
5         minIndex = -1;
6         int minValue = 2147483647; // max int
7         for (int i = 1; i < NR_TASKS; i++){
8             if(task[i]->state!=TASK_RUNNING)
9                 continue;
10            if(task[i]->counter<=0)
11                continue;
12            if(task[i]->counter<minValue){
13                minIndex = i;
14                minValue = task[i]->counter;
15            }
16        }
17        if(minIndex!=-1)
18            break;
19        else
20            for (int i = 1; i < NR_TASKS; i++){
21                task[i]->counter = rand();
22            }
23        switch_to(task[minIndex]);
24    }
25 #endif
26
27 #ifdef PRIORITY
28 void schedule(void) {
29     int next = 0;
30     while (1) {
31         int maxCounter = 0;
32         next = 0;
33         for (int i = NR_TASKS - 1; i >= 0; i--){
34             if(task[i]==NULL)
35                 continue;
36             if(task[i]->state==TASK_RUNNING&&task[i]->counter > maxCounter)
37                 maxCounter = task[i]->counter, next = i;
38         }
39         if(maxCounter>0)
40             break;
41         for (int i = NR_TASKS - 1; i > 0; i--){
42             if(task[i]!=NULL)
43                 task[i]->counter = (task[i]->counter >> 1) + task[i]->priority;
44         }
45         switch_to(task[next]);
46     }
47 #endif

```

在 Makefile 中通过修改 CFLAG，可以向 gcc 命令中添加参数，通过-D 添加宏定义，可以对调度算法进行条件编译。如-DPRIORITY 参数选择对优先级调度算法进行编译。

```

1 CFLAG = ${CF} ${INCLUDE} -DPRIORITY

```

至此，通过 make run 正常启动内核，可以观察到如下输出。

```

[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
[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=10 counter=11 priority=43]
[PID = 10] is running. auto_inc_local_var = 1
[S] Supervisor Mode Timer Interrupt
[PID = 10] is running. auto_inc_local_var = 2

```

通过 make test-run，运行单元测试。

以下为在不同线程数下两种调度算法的测试结果：

```

D
CCCCCCCCCCCCCCCC[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=1 counter=4 priority=37]
B
CCCCCCCCCCCCCCCCB[S] Supervisor Mode Timer Interrupt
B
CCCCCCCCCCCCCCCCBB[S] Supervisor Mode Timer Interrupt
B
CCCCCCCCCCCCCCCCBBB[S] Supervisor Mode Timer Interrupt
B
CCCCCCCCCCCCCCCCBBBB
NR_TASKS = 4, PRIORITY test passed!

[S] Supervisor Mode Timer Interrupt

```

```
B
FFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDDEEEEBB[S] Supervisor Mode Timer Interrupt
B
FFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDDEEEEBB[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=7 counter=2 priority=5]
H
FFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDDEEEEBB[BH[S] Supervisor Mode Timer Interrupt
H
FFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDDEEEEBB[BH
NR_TASKS = 8, PRIORITY test passed!

[S] Supervisor Mode Timer Interrupt
```

```
SWITCH TO [pid=7 counter=2 priority=5]
H
FFFFFFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPPEEEEEMMMBBBBBNNNLLLOOH[S] Supervisor Mode Timer Interrupt
H
FFFFFFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPPEEEEEMMMBBBBBNNNLLLOOH[S] Supervisor Mode Timer Interrupt

SWITCH TO [pid=8 counter=1 priority=25]
I
FFFFFFFFFFFFFFFFKKKKKKKKKKGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPPEEEEEMMMBBBBBNNNLLLOOHHI
NR_TASKS = 16, PRIORITY test passed!

[S] Supervisor Mode Timer Interrupt
```

```
C
BBBBBBBBBBBBBBBBCCCCCCCC[S] Supervisor Mode Timer Interrupt
C
BBBBBBBBBBBBBBBBCCCCCCCC
NR_TASKS = 4, SJF test passed!

[S] Supervisor Mode Timer Interrupt
```

```
F HBBBBEEEEEDDDDDDDCCCCCCCCGGGGGGGGGFFFFFFFFF[S] Supervisor Mode Timer Interrupt  
F  
HBBBBEEEEEDDDDDDDCCCCCCCCGGGGGGGGGFFFFFFFFF  
NR_TASKS = 8, SJF test passed!  
  
[S] Supervisor Mode Timer Interrupt
```

```

IH00LLNNNNBBBBMMMMEEEEEEEEPPPPPPJJJJJJDDDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKKKFFFFFFFFFFF[S] Supervisor Mode Timer Interrupt
F
IH00LLNNNNBBBBMMMMEEEEEEEEPPPPPPJJJJJJDDDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKKKFFFFFFFFFFF[S] Supervisor Mode Timer Interrupt
F
IH00LLNNNNBBBBMMMMEEEEEEEEPPPPPPJJJJJJDDDDDDDDDDCCCCCCCCGGGGGGGGGGKKKKKKKKKKKKFFFFFFFFFFF
NR_TASKS = 16, SJF test passed!

[S] Supervisor Mode Timer Interrupt

```

三、讨论和心得

本次实验流程相对简单，过程中遇到了以下问题。

其一，由于之前看错了指导书，使用了包含 `puti()`和 `puts()`的 `print.h(2022)`版本，目前已经顺利迁移到 `printk.h(2023)`版本。

其二，`test/Makefile` 中的链接部分是对所有 `.o` 文件进行链接，而虽然 `make all` 和 `make test` 设置了不同依赖文件，但是 `schedule_null` 和 `schedule_test` 的同名函数仍会存在冲突。因此，在两种 `make` 模式之间进行切换时，必须执行 `make clean`。

其三，有时在启动内核后会抛出 `scause=0x5` 异常，即 `Load Access Fault`。经过排查和请教，`head.S` 中 `mm_init` 的调用必须在 `#set first time interrupt` 前完成，因为时钟中断会调用 `do_timer()` 函数，进而在调度中访问相关线程，这决定了在第一次时钟中断发生前任务初始化必须完成。

四、思考题

1. 在 RV64 中共有 32 个通用寄存器，为什么 `context_switch` 中只保存了 14 个？

在回答这个问题之前，我们回顾 RISC-V 中的寄存器。除了部分寄存器如 `zero`、`gp`、`tp` 等，在发生函数调用时需要保存相关寄存器。除了 `sp`、`s0~s11` 为 Callee Saved Register 外，其余寄存器均为 Caller Saved Register。

```
1 _traps:
2     # 1. save 32 registers and sepc to stack
3     ...
4     csrr a0, sepc
5     sd a0, 256(sp)
6
7     # 2. call trap_handler
8     csrr a0, scause
9     csrr a1, sepc
10    call trap_handler
11
12    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
13    ld a0, 256(sp)
14    csrw sepc, a0
15    ...
16
17    # 4. return from trap
18    sret
```

```
1 void switch_to(struct task_struct* next) {
2     if(current != next){
3         struct task_struct *prev = current;
4         current = next;
5         printk("\nSWITCH TO [pid=%d counter=%d priority=%d]\n", next->pid, next->counter, next->priority);
6         __switch_to(prev, next);
7     }
8 }
```

```
1 __switch_to:
2     # save state to prev process (ra, sp, s0~s11)
3     ...
4     # restore state from next process (ra, sp, s0~s11)
5     ...
6     ret
```

我们考虑以上的 `__traps`，`switch_to()` 和 `__switch_to` 函数。

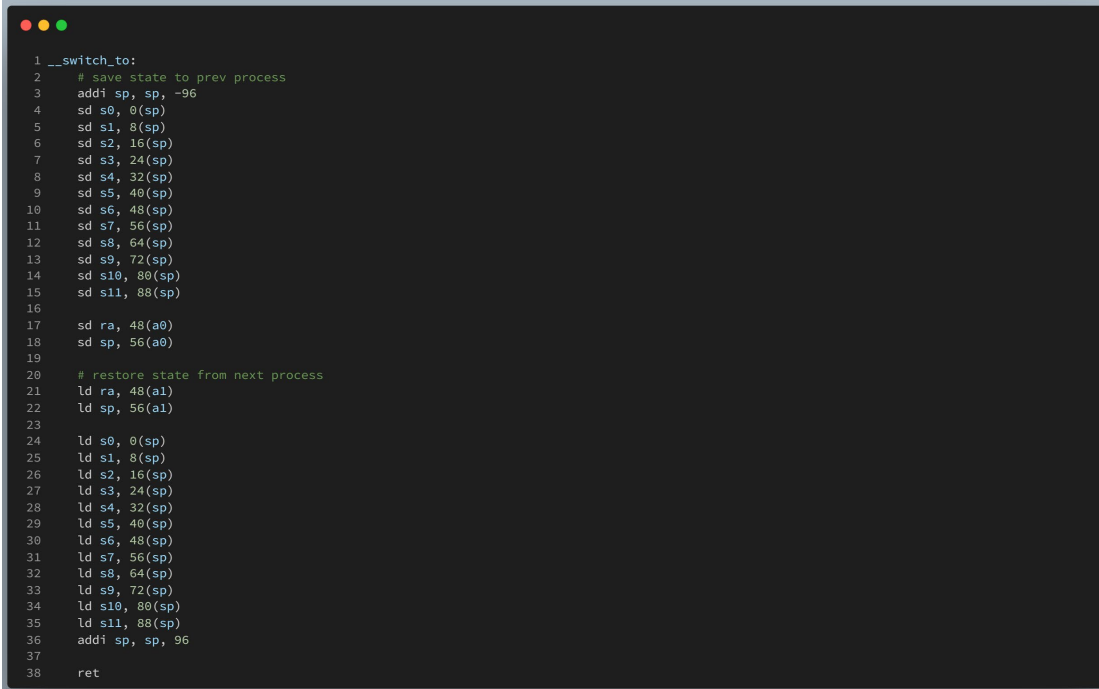
在一次中断处理前后，如果没有发生线程切换，从 `trap_handler()` 返回后，`_traps` 会自然恢复当前进程的 32 个通用寄存器，我们无需展开。因此只需要考虑发生了线程切换的情况，我们假设这次线程切换由线程 A 切换至线程 B。

在线程 A 的运行过程中，发生时钟中断，`_traps` 在调用中断处理函数 `trap_handler()` 之前，将线程 A 的 32 个通用寄存器存储在线程 A 的栈上。进入中断处理函数之后，不久后调用 `switch_to()`，进而调用 `__switch_to`，至此都在线程 A 之中。

在调用 `__switch_to` 之前，即 `switch_to()` 中，线程 A 将此时的 Caller Saved Register 保存到自身的栈上。在 `__switch_to` 中，作为一次函数调用，我们需要将 Callee Saved Register 进行保存，即 `sp` 和 `s0~s11`。同时，由于我们无法确定线程

B 的返回地址是 `dummy()` 还是 `__switch_to` 的返回地址，因此单独维护 `ra`。

事实上，类似的代码也符合需求：



```
1 __switch_to:
2     # save state to prev process
3     addi sp, sp, -96
4     sd s0, 0(sp)
5     sd s1, 8(sp)
6     sd s2, 16(sp)
7     sd s3, 24(sp)
8     sd s4, 32(sp)
9     sd s5, 40(sp)
10    sd s6, 48(sp)
11    sd s7, 56(sp)
12    sd s8, 64(sp)
13    sd s9, 72(sp)
14    sd s10, 80(sp)
15    sd s11, 88(sp)
16
17    sd ra, 48(a0)
18    sd sp, 56(a0)
19
20    # restore state from next process
21    ld ra, 48(a1)
22    ld sp, 56(a1)
23
24    ld s0, 0(sp)
25    ld s1, 8(sp)
26    ld s2, 16(sp)
27    ld s3, 24(sp)
28    ld s4, 32(sp)
29    ld s5, 40(sp)
30    ld s6, 48(sp)
31    ld s7, 56(sp)
32    ld s8, 64(sp)
33    ld s9, 72(sp)
34    ld s10, 80(sp)
35    ld s11, 88(sp)
36    addi sp, sp, 96
37
38    ret
```

切换到线程 B 后，B 将其 `s0~s11` 恢复。如果线程 B 是首次被调度，则从 `__dummy` 继续执行。否则，从 `__switch_to` 返回到 `switch_to()`，C 编译器自动为其恢复 Caller Saved Register，之后从中断处理函数 `trap_handler()` 中返回，在 `_traps` 后续部分恢复 32 个寄存器。

因此，除 `ra` 具有特殊用途之外，在 `__switch_to` 中对 Caller Saved Register 进行的操作不会被外部注意到，我们无需保存。

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。

进入第一次中断 _traps, ra 被设置为中断返回地址。

```
entry.S
3      .globl _traps
4      _traps:
5      # 1. save 32 registers and sepc to stack
6      addi sp, sp, -8*33
7      sd x0, 0(sp)
8      sd x1, 8(sp)
9      sd x2, 16(sp)
10     sd x3, 24(sp)
11     sd x4, 32(sp)

0x80200014 <_stext+20> ld      t0,48(t0)
0x80200018 <_stext+24> csrwr  stvec,t0
0x8020001c <_stext+28> li      t0,32
0x80200020 <_stext+32> csrrs  sie,t0
0x80200024 <_stext+36> rdtime  t0
0x80200028 <_stext+40> lui     t1,0x989
0x8020002c <_stext+44> addiw  t1,t1,1664
0x80200030 <_stext+48> li      a0,0
0x80200034 <_stext+52> li      a1,0
0x80200038 <_stext+56> add     a2,t0,t1

remote Thread 1.1 In: _traps L6 PC: 0x8020005c
(gdb) b _traps
Breakpoint 1 at 0x8020005c: file entry.S, line 6.
(gdb) c
Continuing.

Breakpoint 1, _traps () at entry.S:6
1: /x $ra = 0x80200c20
(gdb)
```

调用 trap_handler 这一中断处理函数后, 被设置为 trap_handler 的返回地址, 即 _traps 中下一条指令的地址。

```
trap.c
3  #include "proc.h"
4  extern void clock_set_next_event();
5  void trap_handler(unsigned long scause, unsigned long sepc){
6      unsigned long interrupt = scause & 0x8000000000000000;
7      unsigned long cause = scause & 0x7fffffff;
8      if(interrupt&&cause==5){ // timer interrupt
9          printk("[S] Supervisor Mode Timer Interrupt\n");
10         clock_set_next_event();
11         do_timer();
12     }
13 }

0x80200b68 <trap_handler+4> sd      ra,40(sp)
0x80200b6c <trap_handler+8> sd      s0,32(sp)
0x80200b70 <trap_handler+12> addi    s0,sp,48
0x80200b74 <trap_handler+16> sd      a0,-40(s0)
0x80200b78 <trap_handler+20> sd      a1,-48(s0)
> 0x80200b7c <trap_handler+24> ld      a4,-40(s0)
0x80200b80 <trap_handler+28> li      a5,-1
0x80200b84 <trap_handler+32> slli    a5,a5,0x3f
0x80200b88 <trap_handler+36> and     a5,a4,a5
0x80200b8c <trap_handler+40> sd      a5,-24(s0)

remote Thread 1.1 In: trap_handler L6 PC: 0x80200b7c
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
(gdb) s
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
1: /x $ra = 0x80200c20
trap_handler (scause=9223372036854775813, sepc=2149583940) at trap.c:6
1: /x $ra = 0x802000f4
(gdb)
```

由于 printk()等函数与本问题无关, 我们不做讨论。进入 do_timer()后, ra 被设置为 do_timer()的返回地址, 即 trap_handler 下一条指令的地址。

```
proc.c
144     switch_to(task[next]);
145 }
146 #endif
147
148 void do_timer(void) {
149     // 1. 如果当前线程是 idle 线程 直接进行调度
150     // 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
151
152     /* YOUR CODE HERE */
}

0x80200998 <do_timer>      addi    sp,sp,-16
0x8020099c <do_timer+4>     sd      ra,8(sp)
0x802009a0 <do_timer+8>     sd      s0,0(sp)
0x802009a4 <do_timer+12>    addi    s0,sp,16
> 0x802009a8 <do_timer+16>   auipc   a5,0x4
0x802009ac <do_timer+20>    addi    a5,a5,1648
0x802009b0 <do_timer+24>    ld      a4,0(a5)
0x802009b4 <do_timer+28>    auipc   a5,0x4
0x802009b8 <do_timer+32>    addi    a5,a5,1628
0x802009bc <do_timer+36>    ld      a5,0(a5)

remote Thread 1.1 In: do_timer                                L153  PC: 0x802009a8
1: /x $ra = 0x80200bc4
(gdb) finish
Run till exit from #0  printk (s=0x802020c8 "[S] Supervisor Mode Timer Interrupt\n") at printk.c:107
trap_handler (scause=9223372036854775813, sepc=2149583940) at trap.c:10
1: /x $ra = 0x80200bc4
Value returned is $1 = 36
(gdb) n
1: /x $ra = 0x80200bc8
(gdb) s
do_timer () at proc.c:153
1: /x $ra = 0x80200bcc
```

进入 schedule()后, ra 被设置成 schedule()的返回地址, 即 do_timer()下一条指令的地址。

```
proc.c
> 101     minIndex = -1;
102     int minValue = 2147483647; // max int
103     for (int i = 1; i < NR_TASKS; i++)
104     {
105         if(task[i]->state!=TASK_RUNNING)
106             continue;
107         if(task[i]->counter<=0)
108             continue;
109         if(task[i]->counter<minValue){
}

> 0x80200824 <schedule+20>   li      a5,-1
0x80200828 <schedule+24>   sw      a5,-36(s0)
0x8020082c <schedule+28>   lui     a5,0x80000
0x80200830 <schedule+32>   not     a5,a5
0x80200834 <schedule+36>   sw      a5,-40(s0)
0x80200838 <schedule+40>   li      a5,1
0x8020083c <schedule+44>   sw      a5,-44(s0)
0x80200840 <schedule+48>   j      0x802008ec <schedule+220>
0x80200844 <schedule+52>   auipc   a4,0x4
0x80200848 <schedule+56>   addi    a4,a4,2004

remote Thread 1.1 In: schedule                                L101  PC: 0x80200824
1: /x $ra = 0x80200bc4
Value returned is $1 = 36
(gdb) n
1: /x $ra = 0x80200bc8
(gdb) s
do_timer () at proc.c:153
1: /x $ra = 0x80200bcc
(gdb) s
1: /x $ra = 0x80200bcc
schedule () at proc.c:101
1: /x $ra = 0x802009c8
```

考虑到__traps, trap_handler(), do_timer()等距离__switch_to 中还存在其他函数调用, ra 会被不断变换, 因此对__switch_to 的变换不产生直接影响。因此, 我们从__switch_to 的主调函数_switch_to 开始观察。

```

proc.c
B+ 87     if(current!=next){
88         struct task_struct *prev = current;
89         current = next;
90         printk("\nSWITCH TO [pid=%d counter=%d priority=%d]\n", next->pid, next->counter, next->priority);
> 91         __switch_to(prev, next);
92     }
93 }
94
95 #ifdef SJF

> 0x802007f0 <switch_to+116> ld      a1,-40(s0)
0x802007f4 <switch_to+120> ld      a0,-24(s0)
0x802007f8 <switch_to+124> jal     ra,0x80200194 <__switch_to>
0x802007fc <switch_to+128> nop
0x80200800 <switch_to+132> ld      ra,40(sp)
0x80200804 <switch_to+136> ld      s0,32(sp)
0x80200808 <switch_to+140> addi    sp,sp,48
0x8020080c <switch_to+144> ret
0x80200810 <schedule>      addi    sp,sp,-48
0x80200814 <schedule+4>    sd      ra,40(sp)

```

在进入__switch_to之后，我们注意到 ra 被置为__switch_to 的返回地址，本例中为 0x802007fc。

```

__switch_to () at entry.S:98
1: /x $ra = 0x802007fc

```

在 ld ra, 48(a1)执行后，ra 被置为 0x80200184，与__dummy 的地址一致。

```

entry.S
111     sd s11, 152(a0)
112     # restore state from next process
113     # YOUR CODE HERE
114     ld ra, 48(a1)
> 115     ld sp, 56(a1)
116     ld s0, 64(a1)
117     ld s1, 72(a1)
118     ld s2, 80(a1)
119     ld s3, 88(a1)

0x802001b8 <__switch_to+36> sd      s7,120(a0)
0x802001bc <__switch_to+40> sd      s8,128(a0)
0x802001c0 <__switch_to+44> sd      s9,136(a0)
0x802001c4 <__switch_to+48> sd      s10,144(a0)
0x802001c8 <__switch_to+52> sd      s11,152(a0)
0x802001cc <__switch_to+56> ld      ra,48(a1)
> 0x802001d0 <__switch_to+60> ld      sp,56(a1)
0x802001d4 <__switch_to+64> ld      s0,64(a1)
0x802001d8 <__switch_to+68> ld      s1,72(a1)
0x802001dc <__switch_to+72> ld      s2,80(a1)

remote Thread 1.1 In: __switch_to L115 PC: 0x802001d0
1: /x $ra = 0x802007fc
1: /x $ra = 0x802007fc
1: /x $ra = 0x802007fc
1: /x $ra = 0x802007fc
1: /x $ra = 0x802007fc
1: /x $ra = 0x802007fc
1: /x $ra = 0x802007fc
(gdb) s
1: /x $ra = 0x80200184
(gdb) p/x &__dummy
$1 = 0x80200184

```

同时，我们关注到 ra 的值被存到了线程中，之后在这一线程再次被调度时，ra 将为__switch_to 的返回地址。

至此，在 ret 后将层层返回，ra 即被层层恢复。

因此，在之后的线程调用中 context_switch 中，ra 保存/恢复的函数返回点是__switch_to 的返回地址。