操作系统lab2：RV64 内核线程调度

教师：李环、柳晴
学号：3220104519
姓名：蔡佳伟

一、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，并完成简单调度算法的实现

二、实验过程

2.1 准备工程

全部完成，lab1代码可以正常运行

```
Boot HART MEDELEG         : 0x0000000000f0b509
2024 ZJU Operating System
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
kernel is running!
kernel is running!
```

2.2 线程调度功能实现

2.2.1 线程初始化

实现task_init()函数：依照提示文字来实现

```
void task_init() {
    srand(2024);

    // 1. 调用 kalloc() 为 idle 分配一个物理页
    // 2. 设置 state 为 TASK_RUNNING;
```

```
    // 3. 由于 idle 不参与调度，可以将其 counter / priority 设置为 0
    // 4. 设置 idle 的 pid 为 0
    // 5. 将 current 和 task[0] 指向 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, 此外，counter 和 priority 进行如下赋
值：
    //     - counter  = 0;
    //     - priority = rand() 产生的随机数（控制范围在 [PRIORITY_MIN, PRIORITY_MAX]
之间）
    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中的 ra 和 sp
    //     - ra 设置为 __dummy（见 4.2.2）的地址
    //     - sp 设置为该线程申请的物理页的高地址

    idle = (struct task_struct *)kalloc();  // 分配一页物理内存
    idle->state = TASK_RUNNING;             // 设置状态为运行
    idle->counter = 0;                      // idle 不参与调度，设置 counter 为 0
    idle->priority = 0;                     // idle 的优先级为 0
    idle->pid = 0;                          // idle 的进程 ID 为 0
    task[0] = idle;                         // task[0] 指向 idle
    current = idle;                         // current 指向当前的 idle 线程

    for (int i = 1; i < NR_TASKS; i++) {
        task[i] = (struct task_struct *)kalloc();  // 分配物理页
        task[i]->state = TASK_RUNNING;             // 设为运行状态
        task[i]->counter = 0;                      // 初始化时 counter 为 0
        task[i]->priority = PRIORITY_MIN + rand() % (PRIORITY_MAX - PRIORITY_MIN
+ 1);  // 随机优先级
        task[i]->pid = i;

        // 设置 thread_struct 中的返回地址和栈指针
        task[i]->thread.ra = (uint64_t)__dummy;    // ra 设置为 __dummy 的地址
        // task[i]->thread.sp = (uint64_t)task[i] + 4096;  // sp 设置为物理页的高地
址
        task[i]->thread.sp = (uint64_t)PGROUNDUP((uint64_t)(task[i]));
    }

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

在head.S中声明并且调用如下：

```
arch > riscv > kernel > ASM head.S
1   .extern start_kernel
2   .extern mm_init
3   .extern task_init
4       .section .text.init
5       .globl _start
6   _start:
7       # ------------------
8       # - your code here -
9
10      # set sp move to top of boot_stack
11      la sp, boot_stack_top
12
13      call mm_init
14
15      call task_init
16
17      # set stvec = _traps
18      la t0, _traps
19      csrw stvec, t0
20
21      # set sie[STIE] = 1
22      li t0, 0x20
23      csrs sie, t0
```

需要在初始化完栈指针之后就调用

2.2.2 __dummy与dummy的实现

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

注意需要在一开始声明外部函数dummy(extern)和全局函数__dummy(globl)。

```
__dummy:
    la t0, dummy        # 将 dummy 的地址加载到 t0 中
    csrw sepc, t0       # 将 sepc 设置为 dummy 的地址

    # 从 S 模式返回并开始执行 dummy() 函数
    sret                # 使用 sret 指令返回到用户模式或下一个特权级
```

2.2.3 实现进程切换

判断下一个执行的线程next与当前的线程是否为同一个线程，如果不是，调用_switch_to函数进行线程切换。

```
void switch_to(struct task_struct *next) {
    if (current != next) {  // 如果 current 和 next 不是同一个线程，才需要进行切换
        struct task_struct *prev = current;  // 保存当前线程
        current = next;                      // 切换到下一个线程
        printk("prepare to switch_to_222\n");
        __switch_to(prev, next);             // 执行上下文切换
    }
}
```

这是在entry.S中实现线程上下文切换_switch_to，因为前面的结构有state,counter,priority,pid四个信息，每个信息是uint64_t（8位），所以offset从32开始。注意ra和sp和s0-s11的保存顺序也不能更换，按照数据结构的顺序来，否则会出错。

```c
/* 线程数据结构 */
struct task_struct {
    uint64_t state;     // 线程状态
    uint64_t counter;   // 运行剩余时间
    uint64_t priority;  // 运行优先级 1 最低 10 最高
    uint64_t pid;       // 线程 id

    struct thread_struct thread;
};
```

```c
/* 线程状态段数据结构 */
struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];
};
```

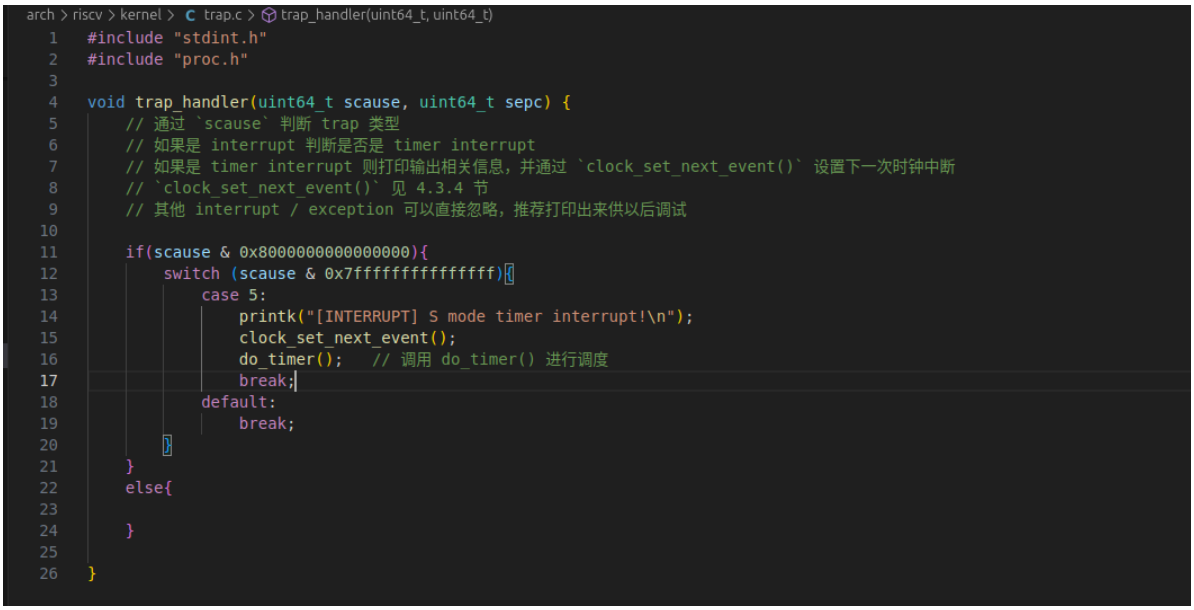
```
97    __switch_to:
98      # save state to prev process a0 保存当前线程的 ra, sp, s0~s11
99      # task_struct 存放在该页的低地址部分，将线程的栈指针 sp 指向该页的高地址
100     sd ra, 32(a0)
101     sd sp, 40(a0)
102     sd s0, 48(a0)
103     sd s1, 56(a0)
104     sd s2, 64(a0)
105     sd s3, 72(a0)
106     sd s4, 80(a0)
107     sd s5, 88(a0)
108     sd s6, 96(a0)
109     sd s7, 104(a0)
110     sd s8, 112(a0)
111     sd s9, 120(a0)
112     sd s10, 128(a0)
113     sd s11, 136(a0)
114
115     # restore state from next process
116     ld ra, 32(a1)
117     ld sp, 40(a1)
118     ld s0, 48(a1)
119     ld s1, 56(a1)
120     ld s2, 64(a1)
121     ld s3, 72(a1)
122     ld s4, 80(a1)
123     ld s5, 88(a1)
124     ld s6, 96(a1)
125     ld s7, 104(a1)
126     ld s8, 112(a1)
127     ld s9, 120(a1)
128     ld s10, 128(a1)
129     ld s11, 136(a1)
130
131     ret
```

2.2.4 实现调度入口函数

实现do_timer()函数

```
void do_timer() {
    // 如果当前线程是 idle 线程或时间片耗尽，则进行调度
    if (current->pid == 0 || current->counter == 0) {  // idle 线程的 pid 假设为 0
        schedule();  // 调用调度函数切换到下一个线程
    } else {
        // 减少当前线程的剩余时间片
        (current->counter)--;
        if (current->counter == 0) {
            // schedule();  // 如果时间片耗尽，进行调度
        }
    }
}
```

在trap.c中调用，从而进行调度



2.2.5 线程调度算法实现

先遍历所有线程，选择counter最大的线程运行，在这里我用next指针记录最大线程。如果所有线程 counter 都为 0，则令所有线程 counter = priority，即优先级越高，运行的时间越长，且越先运行，设置完后重新进行调度，最后通过switch_to切换到下一个线程。

```
void schedule(){
    printk("Schedule\n");
    int max_counter = 0;
    struct task_struct *next = NULL;

    for (int i = 0; i < NR_TASKS; i++) {
        printk("i = %d task[i]->counter = %d\n",i,task[i]->counter);
        if (task[i] && task[i]->counter > 0){
            if(task[i]->counter > max_counter){
                max_counter = task[i]->counter;
                next = task[i];
```

```
            }
        }
    }

    printk("max_counter = %d\n",max_counter);

    if (max_counter == 0) { // 如果没有找到 counter > 0 的线程
        for (int i = 0; i < NR_TASKS; i++) {
            if (task[i]) {   // 重置所有线程的 counter 为其优先级
                task[i]->counter = task[i]->priority;
            }
        }

        schedule(); // 重新进行调度
        return;
    }

    if (next && next != current) { // 切换到选定的线程
        printk("prepare to switch_to\n");
        switch_to(next);
    }
}
```

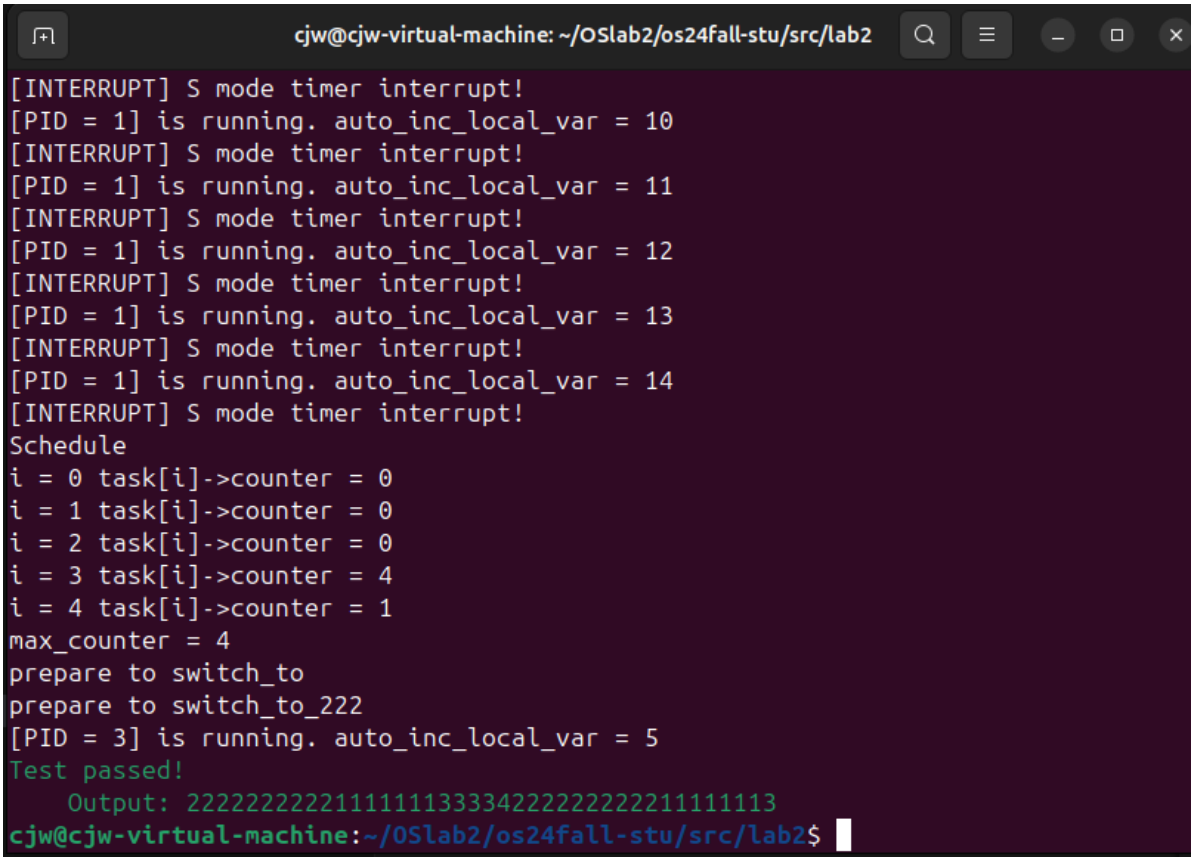
2.2.6 测试结果

注意修改最外层的makefile，添加`-DTEST_SCHED=$(TEST_SCHED)`，这样就可以进行测试用例的测试。修改如下：

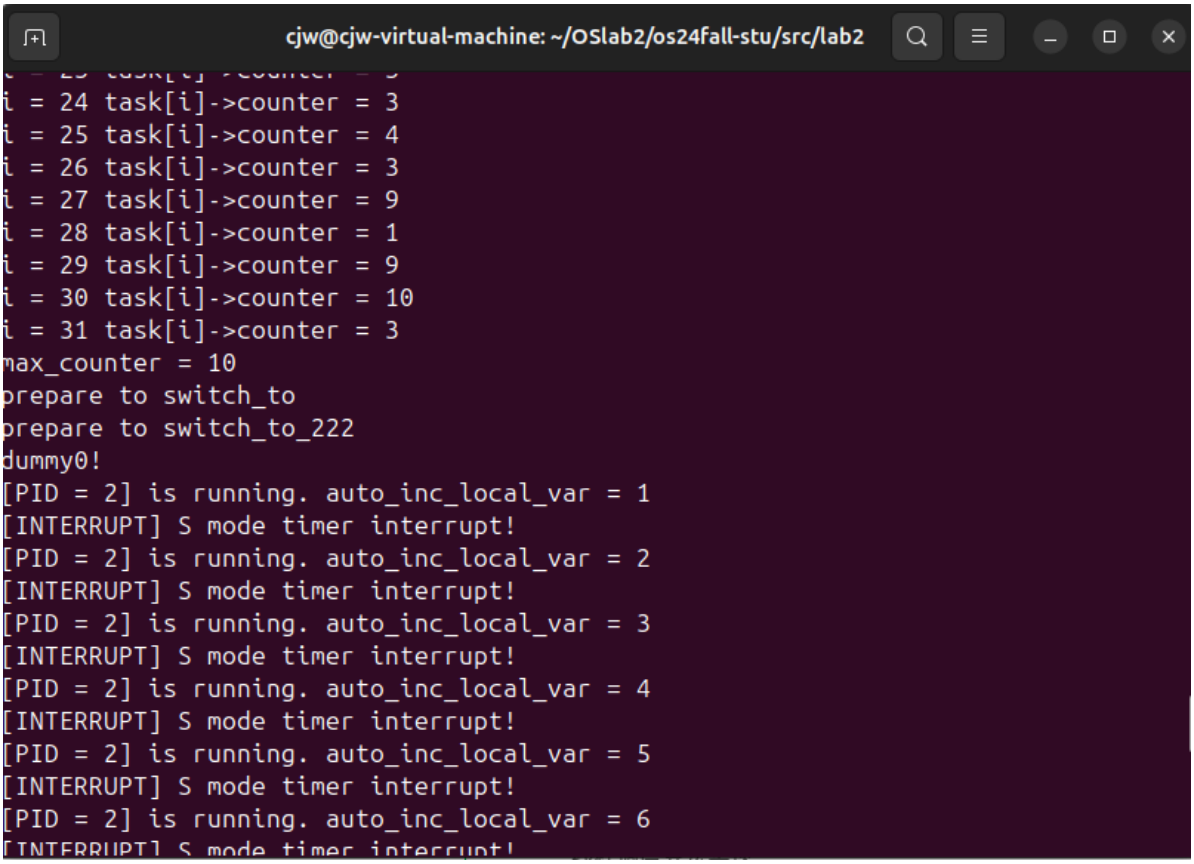
```
export
CROSS   :=  riscv64-linux-gnu-
GCC     :=  $(CROSS)gcc
LD      :=  $(CROSS)ld
OBJCOPY :=  $(CROSS)objcopy
OBJDUMP :=  $(CROSS)objdump
TEST_SCHED  :=  0
CROSS_=riscv64-linux-gnu-
GCC=${CROSS_}gcc
# 链接器
LD=${CROSS_}ld
# 对象复制工具和对象转储工具
OBJCOPY=${CROSS_}objcopy
OBJDUMP=${CROSS_}objdump

ISA     :=  rv64imafd
ABI     :=  lp64

INCLUDE :=  -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include
CF      :=  -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builtin -ffunction-
sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--
nmagic -Wl,--gc-sections -g
CFLAG   :=  $(CF) $(INCLUDE) -DTEST_SCHED=$(TEST_SCHED)

.PHONY:all run debug clean
```

```
all: clean
    $(MAKE) -C lib all
    $(MAKE) -C init all
    $(MAKE) -C arch/riscv all
    @echo -e '\n'Build Finished OK

run: all
    @echo Launch qemu...
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default

debug: all
    @echo Launch qemu for debug...
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default
-S -s

clean:
    $(MAKE) -C lib clean
    $(MAKE) -C init clean
    $(MAKE) -C arch/riscv clean
    $(shell test -f vmlinux && rm vmlinux)
    $(shell test -f vmlinux.asm && rm vmlinux.asm)
    $(shell test -f System.map && rm System.map)
    @echo -e '\n'Clean Finished
```

`make TEST_SCHED=1 run` 结果如下：



`make run` 结果如下：



可以看出均正确。

三、讨论和心得

这次的实验在理解了每一个小实验的文字部分以后，还是比较清晰的。过程中遇到的几个问题，第一个是makefile不会修改，也不知道要修改哪里，这会导致测试用例不能正常运行，最后也是照着文字一点点尝试改对；第二个是entry.S中的__switch_to函数从0开始，没有从32开始，从而导致访问内存出现错误；第三个还是__switch_to函数没有按顺序访问，也会导致访问内存出错，因而无法正常跳转到dummy函数中。这次实验我也学到了很多知识，比如进程的调度，有关操作系统的实践的更多概念，甚至实验一中的部分代码也更深理解了。

四、思考题

1. 在 RV64 中一共有 32 个通用寄存器，为什么 __switch_to 中只保存了 14 个?

每个进程中ra和sp的值必须被保存，s0-s11寄存器的值也必须被保存。剩下的寄存器gp、tp在本实验未涉及，故不需要保存，其他寄存器为caller saved寄存器，也不需要保存。

2. 阅读并理解 arch/riscv/kernel/mm.c 代码，尝试说明 mm_init 函数都做了什么，以及在 kalloc 和 kfree 的时候内存是如何被管理的。

`mm_init` 函数是整个内存管理系统的初始化函数，它调用了 `kfreerange`，将从内核结束位置 `_ekernel` 到物理内存的结束位置 `PHY_END` 这一段区域标记为可分配内存。

具体操作：

- `_ekernel`：这是内核结束的地址（通常是链接器脚本中定义的）。

- `PHY_END`：表示物理内存的结束地址。
- `kfreerange`：这个函数通过对指定的内存范围进行 `PGSIZE`（页面大小）对齐，然后逐页调用 `kfree` 函数，将这段内存加入到空闲列表中，以便后续分配。

通过 `mm_init` 的初始化，内核会把从 `_ekernel` 到 `PHY_END` 的所有空闲内存页面逐步加入到内存分配系统的空闲列表中。

`kalloc` 用于从空闲内存列表中分配一个物理页面，并返回页面的地址。

具体步骤：

- **空闲列表**：`kmem.freelist` 指向当前空闲的物理页面。`struct run` 是一个链表结构，每个节点代表一个物理页面，`freelist` 是链表的头部。
- **从空闲列表取出页面**：`r = kmem.freelist` 将当前空闲列表的第一个页面取出，并将 `kmem.freelist` 指向下一个空闲页面。
- **清空页面内容**：`memset((void *)r, 0x0, PGSIZE)` 将分配的页面内容清零，以确保返回的页面内容是干净的。
- **返回页面地址**：最终返回该页面的地址。

因此，`kalloc` 相当于从空闲内存池中分配一个 4KB（假设 `PGSIZE` 为 4KB）的内存块，并返回其起始地址。

`kfree` 用于将一个已分配的页面重新释放回空闲列表中，以便以后可以再次使用。

具体步骤：

- **对齐地址**：`addr & ~(PGSIZE - 1)` 保证传入的地址按页面大小（4KB）对齐。这样可以避免传入未对齐的地址带来的问题。
- **清空页面内容**：`memset(addr, 0x0, PGSIZE)` 清空该页面的内容，以防止未来使用时泄露旧的数据。
- **将页面重新加入空闲列表**：通过将这个页面包装成一个 `struct run` 节点，然后将其插入到 `kmem.freelist` 链表的头部，实现了内存的回收。释放的页面会重新成为下一个可分配的空闲页面。

3. 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 switch_to 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（需要截图）。

第一次调用时，由于我们在task_init中设置了task[i]->thread.ra = (uint64_t)dummy，所以会返回到ra存储的_dummy函数中去，之后在这个线程被中断时，traps函数会保存目前的运行情况地址（即switch_to函数中调用__switch_to的位置，如下图所示。

```
(gdb) n
112         sd s10, 128(a0)
(gdb) n
113         sd s11, 136(a0)
(gdb) n
116         ld ra, 32(a1)
(gdb) i r ra
ra             0x80200778       0x80200778 <switch_to+96>
(gdb) n
117         ld sp, 40(a1)
(gdb) i r ra
ra             0x8020016c       0x8020016c <__dummy>
(gdb) c
Continuing.
```

如果下一个要切换的线程是第一次运行，则会跳转到_dummy，如果已经运行过至少一次，则是从上一次保存的地址开始运行，如图所示

```
void switch_to(struct task_struct *next) {
    if (current != next) {  // 如果 current 和 next 不是同一个线程，才需要进行切换
        struct task_struct *prev = current;  // 保存当前线程
        current = next;                      // 切换到下一个线程
        printk("prepare to switch_to_222\n");
        __switch_to(prev, next);             // 执行上下文切换
    }
}
```

gdb追踪：

进入_traps：

```
(gdb) target remote: 1234
Remote debugging using : 1234
0x0000000000001000 in ?? ()
(gdb) b dummy
Breakpoint 1 at 0x80200638: file proc.c, line 63.
(gdb) b _traps
Breakpoint 2 at 0x80200054: file entry.S, line 12.
(gdb) c
Continuing.

Breakpoint 2, _traps () at entry.S:12
12          addi sp, sp, -256
(gdb)
```

进入trap_handler函数处理中断：

```
(gdb) n
13          sd tp, 0(sp)
(gdb) b trap_handler
Breakpoint 3 at 0x80200cb0: file trap.c, line 11.
(gdb) c
Continuing.

Breakpoint 3, trap_handler (scause=9223372036854775813, sepc=2149584240)
    at trap.c:11
11              if(scause & 0x8000000000000000){
(gdb)
```

因为是时钟中断，所以需要调用clock_set_next_event，并且进入do_timer

```
Breakpoint 3, trap_handler (scause=9223372036854775813, sepc=2149584240)
    at trap.c:11
11              if(scause & 0x8000000000000000){
(gdb) n
12                      switch (scause & 0x7fffffffffffffff){
(gdb) n
14                                      printk("[INTERRUPT] S mode timer interru
pt!\n");
(gdb) n
15                                      clock_set_next_event();
(gdb) n
16                                      do_timer();   // 调用 do_timer() 进行调
度
(gdb) b do_timer
Breakpoint 4 at 0x8020079c: file proc.c, line 106.
(gdb)
```

do_timer调用schedule函数

```
(gdb) b do_timer
Breakpoint 4 at 0x8020079c: file proc.c, line 106.
(gdb) b schedule
Breakpoint 5 at 0x8020080c: file proc.c, line 118.
(gdb) n

Breakpoint 4, do_timer () at proc.c:106
106         if (current->pid == 0 || current->counter == 0) {  // idle 线程的 pi
d 假设为 0
(gdb) c
Continuing.

Breakpoint 5, schedule () at proc.c:118
118         printk("Schedule\n");
(gdb)
```

```
(gdb) c
Continuing.

Breakpoint 5, schedule () at proc.c:118
118         printk("Schedule\n");
(gdb)
```

进入switch_to函数

```
(gdb) c
Continuing.

Breakpoint 6, switch_to (next=0x87ffd000) at proc.c:96
96          if (current != next) {  // 如果 current 和 next 不是同一个线程，才需
要进行切换
(gdb)
```

switch_to调用第一次进入__switch_to

```
(gdb) b __switch_to
Breakpoint 7 at 0x8020017c: file entry.S, line 100.
(gdb) c
Continuing.

Breakpoint 7, __switch_to () at entry.S:100
100         sd ra, 32(a0)
(gdb)
```

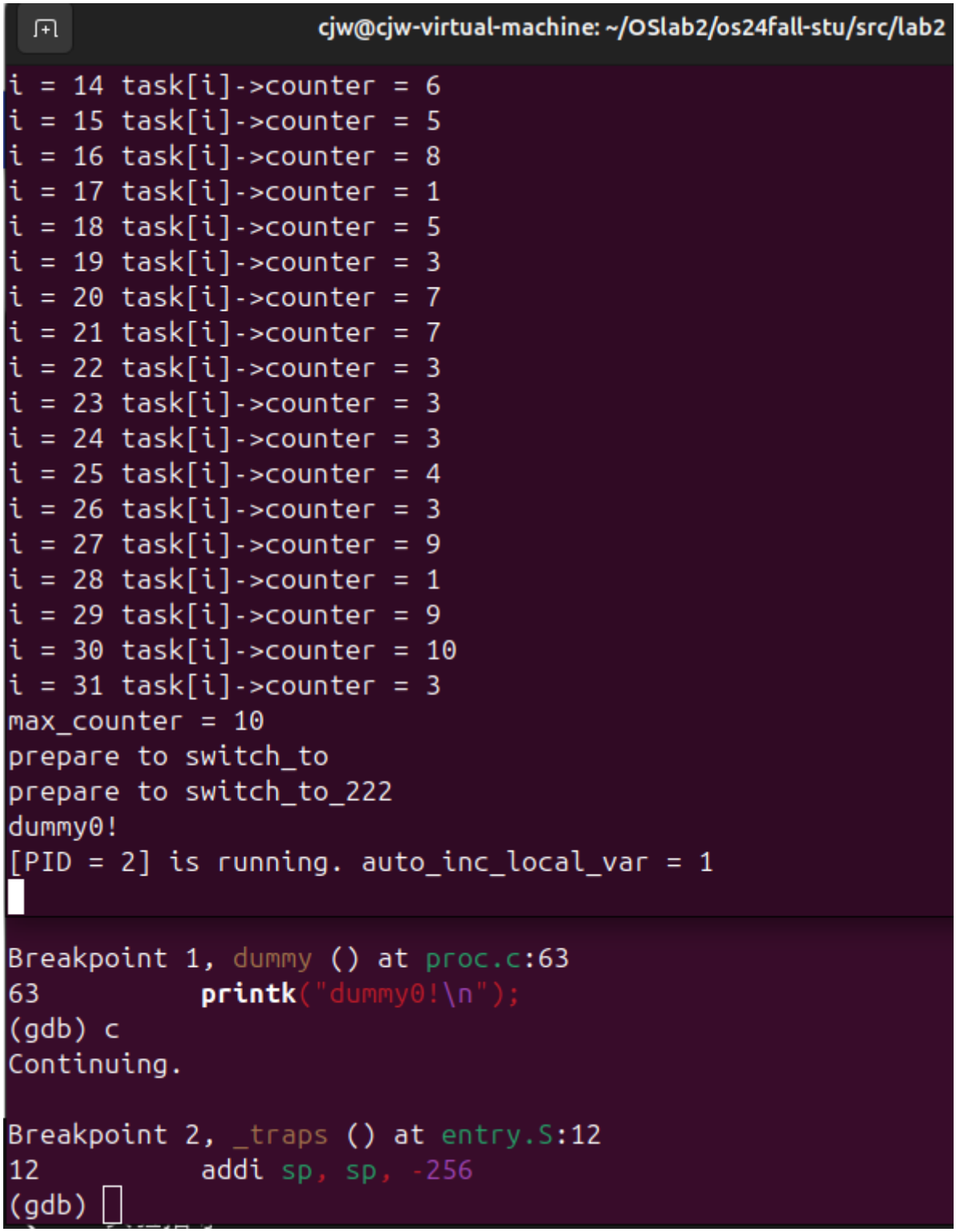
此时ra值为switch_to+96，即调用_switch_to语句的下一条指令

```
100         sd ra, 32(a0)
(gdb) i r ra
ra             0x80200778       0x80200778 <switch_to+96>
(gdb) n
101         sd sp, 40(a0)
(gdb)
```

运行到第一次切换ra，使用task_init中设置的另一个线程的ra，即_dummy

```
(gdb) n
112         sd s10, 128(a0)
(gdb) n
113         sd s11, 136(a0)
(gdb) n
116         ld ra, 32(a1)
(gdb) i r ra
ra             0x80200778       0x80200778 <switch_to+96>
(gdb) n
117         ld sp, 40(a1)
(gdb) i r ra
ra             0x8020016c       0x8020016c <__dummy>
(gdb) c
Continuing.
```

继续运行，切换到了线程2


cjw@cjw-virtual-machine: ~/OSlab2/os24fall-stu/src/lab2
i = 14 task[i]->counter = 6
i = 15 task[i]->counter = 5
i = 16 task[i]->counter = 8
i = 17 task[i]->counter = 1
i = 18 task[i]->counter = 5
i = 19 task[i]->counter = 3
i = 20 task[i]->counter = 7
i = 21 task[i]->counter = 7
i = 22 task[i]->counter = 3
i = 23 task[i]->counter = 3
i = 24 task[i]->counter = 3
i = 25 task[i]->counter = 4
i = 26 task[i]->counter = 3
i = 27 task[i]->counter = 9
i = 28 task[i]->counter = 1
i = 29 task[i]->counter = 9
i = 30 task[i]->counter = 10
i = 31 task[i]->counter = 3
max_counter = 10
prepare to switch_to
prepare to switch_to_222
dummy0!
[PID = 2] is running. auto_inc_local_var = 1
Breakpoint 1, dummy () at proc.c:63
63 printk("dummy0!\n");
(gdb) c
Continuing.
Breakpoint 2, _traps () at entry.S:12
12 addi sp, sp, -256
(gdb)

继续运行

```
Breakpoint 3, trap_handler (scause=9223372036854775813, sepc=2149582464)
    at trap.c:11
11              if(scause & 0x8000000000000000){
(gdb)
Continuing.

Breakpoint 4, do_timer () at proc.c:106
106         if (current->pid == 0 || current->counter == 0) {  // idle 线程的
d 假设为 0
(gdb)
Continuing.

Breakpoint 2, _traps () at entry.S:12
12          addi sp, sp, -256
(gdb)
```

继续运行，指导切换目标是一个运行过的线程，此时ra的值并不是_dummy，而是之前保存的switch_to+96

```
Breakpoint 6, switch_to (next=0x87ff7000) at proc.c:96
96          if (current != next) {  // 如果 current 和 next 不是同一个线程，才需
要进行切换
(gdb) c
Continuing.

Breakpoint 7, __switch_to () at entry.S:100
100         sd ra, 32(a0)
(gdb) i r ra
ra             0x80200778       0x80200778 <switch_to+96>
(gdb)
```

4. 请尝试分析并画图说明 kernel 运行到输出第两次 switch to [PID ... 的时候内存中存在的全部函数帧栈布局。可通过 gdb 调试使用 backtrace 等指令辅助分析，注意分析第一次时钟中断触发后的 pc 和 sp 的变化。

图示分析：

```
时间点 1: 内核正在执行中 (PC: 0x0000xxxx, SP: 0x7ffffxxx)
    +--------------------+      +--------------------+
    |     栈帧 A         |      |    正常指令        |
    +--------------------+      +--------------------+
    |     栈帧 B         |      | PC-> 0x0000xxxx    |
    +--------------------+      +--------------------+
    |  栈顶指针 SP       |      |                    |
    +--------------------+      +--------------------+

时间点 2: 时钟中断发生 (PC: 0x0000yyyy, SP: 0x7fffyyy)
    +--------------------+      +--------------------+
    |  栈帧 A            |      | 保存 PC 到栈中     |
    +--------------------+      +--------------------+
    |  栈帧 B            |      | PC 跳转到中断处理  |
    +--------------------+      +--------------------+
    |  中断处理帧        |      | PC-> 0x0000yyyy    |
    +--------------------+      +--------------------+
    |  栈顶指针 SP       |      |                    |
    +--------------------+      +--------------------+

时间点 3: 上下文切换 (切换到新进程，PC: 0x0000zzzz, SP: 0x7fffzzzz)
    +--------------------+      +--------------------+
    |  栈帧 A            |      | 切换到新进程       |
    +--------------------+      +--------------------+
    |  新进程栈帧 C      |      | PC-> 0x0000zzzz    |
    +--------------------+      +--------------------+
    |  栈顶指针 SP       |      |                    |
    +--------------------+      +--------------------+
```

gdb调试：

```
(gdb) bt
#0  switch_to (next=0x87ffd000) at proc.c:96
#1  0x00000000802009e4 in schedule () at proc.c:147
#2  0x00000000802009b0 in schedule () at proc.c:141
#3  0x00000000802007c8 in do_timer () at proc.c:107
#4  0x0000000080200ce4 in trap_handler (scause=9223372036854775813,
    sepc=2149584240) at trap.c:16
#5  0x00000000802000e4 in _traps () at entry.S:50
Backtrace stopped: frame did not save the PC
(gdb) info registers
ra             0x802009e4       0x802009e4 <schedule+488>
sp             0x80204e10       0x80204e10
gp             0x0      0x0
tp             0x80047000       0x80047000
t0             0x80200d70       2149584240
t1             0x0      0
t2             0x0      0
fp             0x80204e40       0x80204e40
s1             0x1      1
```

```
a1             0x0      0
a2             0x0      0
a3             0xa      10
a4             0x0      0
a5             0x15     21
a6             0x2      2
a7             0x4442434e       1145193294
s2             0x0      0
s3             0x80200000       2149580800
s4             0x0      0
s5             0x87e00000       2279604224
s6             0x8000000a00006800       -9223371993905076224
s7             0x80040040       2147745856
--Type <RET> for more, q to quit, c to continue without paging--c
s8             0x2000   8192
s9             0x800426e0       2147755744
s10            0x0      0
s11            0x0      0
t3             0x10     16
t4             0x80046dda       2147773914
t5             0xf      15
t6             0x27     39
pc             0x8020072c       0x8020072c <switch_to+20>
```

可以看到，在第一次switch_to时，sp和pc的地址

```
Breakpoint 1, switch_to (next=0x87ff7000) at proc.c:96
96          if (current != next) {  // 如果 current 和 next 不是同一个线程，才需
要进行切换
(gdb) n
97              struct task_struct *prev = current;  // 保存当前线程
(gdb) info registers
ra             0x802009e4       0x802009e4 <schedule+488>
sp             0x87ffce40       0x87ffce40
gp             0x0      0x0
tp             0x80047000       0x80047000
t0             0x80200670       2149582448
t1             0x0      0
t2             0x0      0
```

```
a1             0x0      0
a2             0x0      0
a3             0xa      10
a4             0x87ff7000       2281664512
a5             0x87ffd000       2281689088
a6             0x2      2
a7             0x4442434e       1145193294
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
--Type <RET> for more, q to quit, c to continue without paging--c
s8             0x0      0
s9             0x0      0
s10            0x0      0
s11            0x0      0
t3             0x10     16
t4             0x80046dda       2147773914
t5             0xf      15
t6             0x27     39
pc             0x80200740       0x80200740 <switch_to+40>
(gdb)
```

在第二次switch_to调用时，可以看到SP 发生变化，SP 指向当前栈的顶部。当内核执行函数调用时，会在栈上创建新的栈帧，SP 会减少以分配新的栈空间。当函数返回时，SP 会恢复到函数调用前的状态。PC 也发生了变化，因为PC 存储了当前正在执行的指令地址，每次执行完一条指令后，PC 通常会递增，指向下一条指令。如果遇到函数调用，PC 会跳转到被调用函数的地址。