浙江大学

本科实验报告

课程名称： 操作系统

姓 名： 周歆仪

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： 3220106418

指导教师： 李环

2024年10月8日

浙江大学操作系统实验报告

实验名称：Lab2 RV64 内核线程调度

电子邮件地址： CindyZhou2003@outlook.com

手机： 13917199624

实验地点：曹光彪西楼503

实验日期：2024年 10月 5日

一、实验目的和要求

- 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

实验环境

- Environment in previous labs

二、实验过程

1. 准备工程

根据实验手册同步并修改一些代码，最后得到代码结构如下：

```
.
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       ├── include
│       │   ├── defs.h
```

```
8   │   │   ├── mm.h
9   │   │   ├── proc.h
10  │   │   └── sbi.h
11  │   └── kernel
12  │       ├── Makefile
13  │       ├── clock.c
14  │       ├── entry.S
15  │       ├── head.S
16  │       ├── mm.c
17  │       ├── proc.c
18  │       ├── sbi.c
19  │       ├── trap.c
20  │       └── vmlinux.lds
21  ├── include
22  │   ├── printk.h
23  │   ├── stddef.h
24  │   ├── stdint.h
25  │   ├── stdlib.h
26  │   ├── string.h
27  │   └── types.h
28  ├── init
29  │   ├── Makefile
30  │   ├── main.c
31  │   └── test.c
32  └── lib
33      ├── Makefile
34      ├── printk.c
35      ├── rand.c
36      └── string.c
```

在 `head.S` 中，我们在 `_start` 初始化栈后调用了两个初始化函数，用来初始化内存管理系统：

```
_start:
    # ———————————————————
    # -  initialize stack  -
    # ———————————————————
    # Set stack pointer to .bss.stack
    la sp, boot_stack_top  # Load the top of the stack into sp (stack pointer)

    # 初始化
    call mm_init
```

2. 线程调度功能实现

1）线程初始化

```c
void task_init() {
    srand(2024);  // 设置随机种子

    // 1. 为 idle 线程分配物理页
    idle = (struct task_struct *)kalloc();
    if (!idle) {
        printk("Failed to allocate memory for idle task.\n");
        return;
    }

    // 2. 初始化 idle 线程的状态
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 12345;
    idle->pid = 0; // 将 idle 线程的 pid 设为 0

    // 3. 设置当前线程和任务数组
    current = idle;
    task[0] = idle;

    // 为 task[1] ~ task[NR_TASKS - 1] 进行初始化
    for (int i = 1; i < NR_TASKS; ++i) {
        // 1. 为每个线程分配物理页
        task[i] = (struct task_struct *)kalloc();
        if (!task[i]) {
            printk("Failed to allocate memory for task %d.\n", i);
            return;
        }

        // 2. 初始化线程状态
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority =  rand() % 10 + 1;  // 设置随机优先级
        task[i]->pid = i;

        // 3. 设置线程的 ra 和 sp
        task[i]->thread.ra = (uint64_t)__dummy;
        task[i]->thread.sp = (uint64_t)task[i] + 4096;  // 物理页的高地址
    }

```

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

同时，我们在 `arch/riscv/kernel/head.S` 中 `call mm_init` 后调用 `task_init()` 进行线程初始化

2) `__dummy` 与 `dummy` 的实现

在 `arch/riscv/kernel/entry.S` 中添加函数 `__dummy`：在 `__dummy` 中将 `sepc` 设置为 `dummy()` 的地址，并使用 `sret` 从 S 模式中返回

```
__dummy:
    # 设置 sepc 寄存器为 dummy 函数的地址
    la      t0, dummy     // 将 dummy 地址加载到 t0 寄存器中
    csrw    sepc, t0      // 将 t0 中的地址写入到 sepc (Supervisor Exception Program Counter)
    # 使用 sret 返回到 S 模式，让新的线程从 sepc 中的地址（即 dummy 的地址）开始执行
    sret
```

3）实现线程切换

在 `arch/riscv/kernel/proc.c` 中添加 `switch_to` 函数：

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：

```
extern void __switch_to(struct task_struct *prev, struct task_struct *next);

void switch_to(struct task_struct *next) {
    if (current == next) {
        printk(RED"Already running task %d, no switch needed.\n", next->pid);
        return;
    }

    struct task_struct *prev = current;
    printk(RED"Switching from task %d to task %d :[PID = %d, COUNTER = %d, PRIORITY = %d]\n",
        prev->pid, next->pid, next->counter, next->priority);
    current = next;  // 更新 current 指向

    // 传递 prev 和 next
    __switch_to(prev, next);  // 切换上下文
```

```
}
```

在 `arch/riscv/kernel/entry.S` 中实现线程上下文切换 `__switch_to`：

- `__switch_to` 接受两个 `task_struct` 指针作为参数；
- 保存当前线程的 `ra`，`sp`，`s0~s11` 到当前线程的 `thread_struct` 中；
- 将下一个线程的 `thread_struct` 中的相关数据载入到 `ra`，`sp`，`s0~s11` 中进行恢复：

  由于线程的数据结构为

  ```
  struct task_struct {
      uint64_t state;      // 线程状态 8字节
      uint64_t counter;    // 运行剩余时间 8字节
      uint64_t priority;   // 运行优先级 1 最低 10 最高 8字节
      uint64_t pid;        // 线程 id 8字节

      struct thread_struct thread;
  };
  ```

  所以 `thread_struct` 所存放的地址是当前 `task_struct` 地址+32

```
    .globl __switch_to
__switch_to:
    sd ra, 32(a0)          # 保存 ra 到 prev->thread_struct.ra
    sd sp, 40(a0)          # 保存 sp 到 prev->thread_struct.sp

    # 保存 s0 ~ s11 到 prev->thread_struct.s[0] ~ s[11]
    sd s0, 48(a0)
    sd s1, 56(a0)
    sd s2, 64(a0)
    sd s3, 72(a0)
    sd s4, 80(a0)
    sd s5, 88(a0)
    sd s6, 96(a0)
    sd s7, 104(a0)
    sd s8, 112(a0)
    sd s9, 120(a0)
    sd s10, 128(a0)
    sd s11, 136(a0)

    # 加载下一个线程的上下文
    ld ra, 32(a1)          # 恢复 ra (从 next->thread_struct.ra 加载)
    ld sp, 40(a1)          # 恢复 sp (从 next->thread_struct.sp 加载)

    # 恢复 s0 ~ s11
```

```
    ld s0, 48(a1)
    ld s1, 56(a1)
    ld s2, 64(a1)
    ld s3, 72(a1)
    ld s4, 80(a1)
    ld s5, 88(a1)
    ld s6, 96(a1)
    ld s7, 104(a1)
    ld s8, 112(a1)
    ld s9, 120(a1)
    ld s10, 128(a1)
    ld s11, 136(a1)

    ret
```

4）实现调度入口函数

在 `arch/riscv/kernel/proc.c` 中实现 `do_timer()` 函数

```c
void do_timer(void) {
   /* 如果当前线程是 idle 线程 直接进行调度 */
    if (current == idle || current->pid == 0) {
        printk(YELLOW"idle process is running!\n");
        schedule();
        return;
    }

    /* 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 */
    if(current->counter > 0) {
        current->counter -- ;
        printk(YELLOW"do_timer: Task %d has counter = %d\n", current->pid, current->counter);
        return;
    }

    if (current->counter == 0) {
        printk(YELLOW"Task %d has counter 0, forcing switch ...\n", current->pid);
        schedule();  // Trigger rescheduling
        return;
    }

}
```

并在 `arch/riscv/kernel/trap.c` 时钟中断处理函数中调用：

```c
// Trap处理函数
void trap_handler(unsigned long scause, unsigned long sepc) {
    unsigned long stval, sstatus;

    // 获取 stval 和 sstatus 的值，用于进一步分析异常信息
    __asm__ volatile("csrr %0, stval" : "=r"(stval));
    __asm__ volatile("csrr %0, sstatus" : "=r"(sstatus));

    if (scause >> 63) {  // 中断
        if ((scause & 0xFF) == 5) { // Exception Code == 5, 判断为 timer interrupt
            // printk("%s", "[Trap Handler] S-Mode Timer Interrupt!\n");

            // 处理定时器中断
            clock_set_next_event();  // 设置下一个时钟中断事件
            do_timer();              // 调用 do_timer 处理调度逻辑
        } else {
            printk("Other interrupt");
        }
    } else {  // 异常
        printk("Exception occurred, type: %lx at SEPC: %lx, STVAL: %lx, SSTATUS: %lx\n", scause, sepc, stval, sstatus);
    }
}
```

5）线程调度算法实现

参考 Linux v0.11 调度算法代码 实现一个优先级调度算法，具体逻辑如下：

- `task_init` 的时候随机为各个线程赋予了优先级
- 调度时选择 `counter` 最大的线程运行
- 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
  - 设置完后需要重新进行调度
- 最后通过 `switch_to` 切换到下一个线程

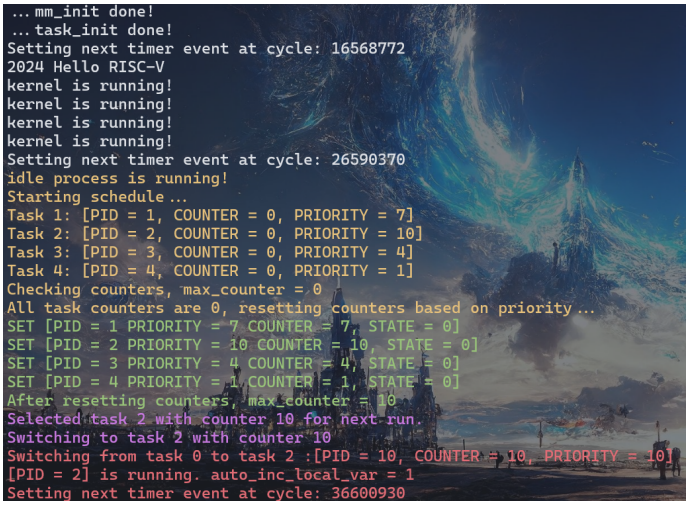
```c
void schedule() {
    uint64_t max_counter = 0;
    struct task_struct *next_task = NULL;

    printk("Starting schedule ... \n");

```

```c
    // 检查所有任务状态
    for (int i = 1; i < NR_TASKS; ++i) {
        printk("Task %d: [PID = %d, COUNTER = %d, PRIORITY = %d]\n",
               i, task[i]->pid, task[i]->counter, task[i]->priority);

        if (task[i]->state == TASK_RUNNING && task[i]->counter ≥ max_counter && task[i]->counter > 0) {
            max_counter = task[i]->counter;
            next_task = task[i];
        }
    }
    printk("Checking counters, max_counter = %d\n", max_counter);

    // 如果所有任务的 counter 都为 0，则重新分配 counter = priority
    if (max_counter == 0 || next_task == NULL) {
        printk("All task counters are 0, resetting counters based on priority...\n");
        for (int i = 1; i < NR_TASKS; ++i) {
            task[i]->counter = task[i]->priority;
            // task[i]->state = TASK_RUNNING;  // 强制将所有任务状态设置为 TASK_RUNNING
            printk(GREEN"SET [PID = %d PRIORITY = %d COUNTER = %d, STATE = %d]\n",
                   task[i]->pid, task[i]->priority, task[i]->counter, task[i]->state);
        }

        // 重新选择调度的线程
        for (int i = 1; i < NR_TASKS; ++i) {
            if (task[i]->state == TASK_RUNNING && task[i]->counter ≥ max_counter && task[i]->counter > 0) {
                max_counter = task[i]->counter;
                next_task = task[i];
            }
        }
        printk("After resetting counters, max_counter = %d\n", max_counter);
    }


    if (next_task) {
        printk(PURPLE"Selected task %d with counter %d for next run.\n", next_task->pid, next_task->counter);
    }

    // 调用 switch_to 切换到下一个线程
    if (next_task && next_task ≠ current) {
```

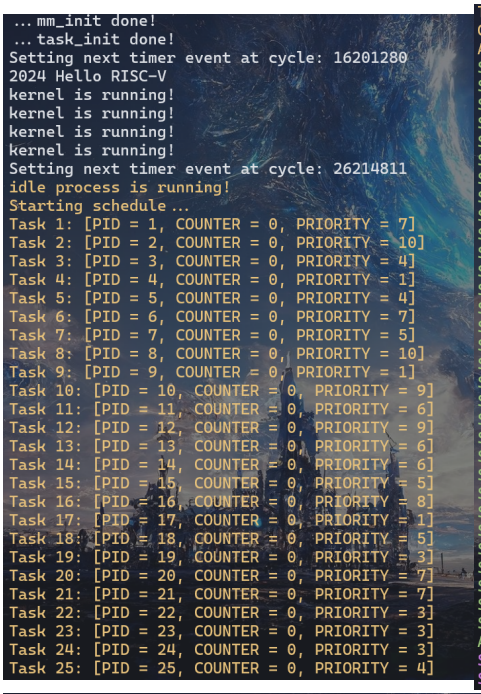
```
        printk(PURPLE"Switching to task %d with counter %d\n", next_task->pid, next_task->counter);
        switch_to(next_task);
    } else {
        printk(PURPLE"No task switch needed. Current task: %d, Counter: %d\n", current->pid, current->counter);
    }
}
```

3. 编译及测试

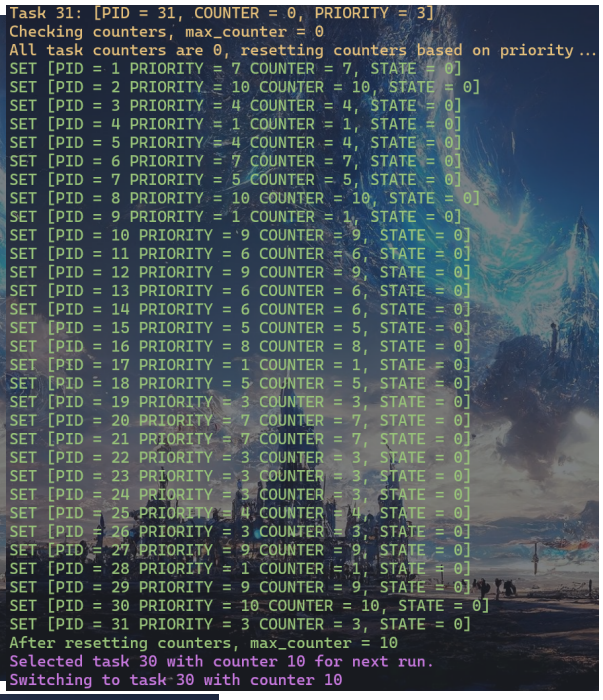
`make TEST_SCHED=1 run` 的结果输出：



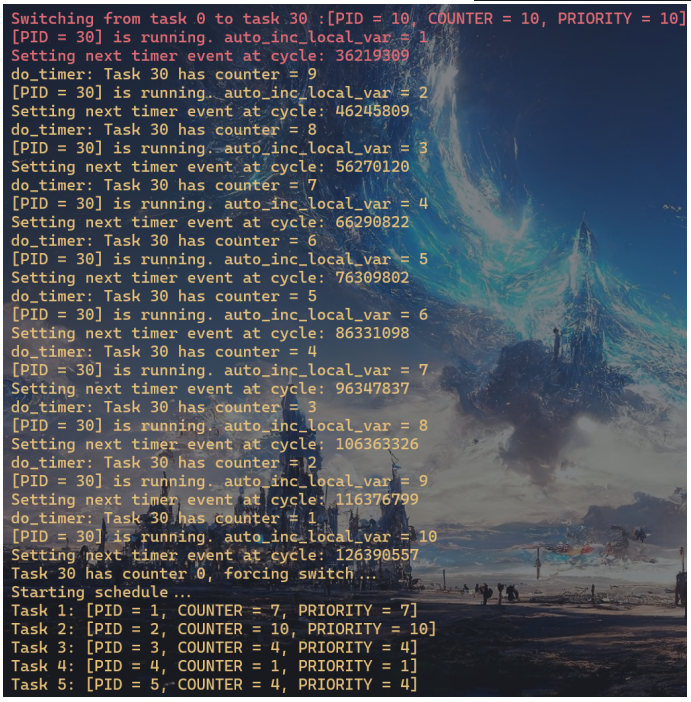
`make run` 的输出：



```
...mm_init done!
...task_init done!
Setting next timer event at cycle: 16201280
2024 Hello RISC-V
kernel is running!
kernel is running!
kernel is running!
kernel is running!
Setting next timer event at cycle: 26214811
idle process is running!
Starting schedule...
Task 1: [PID = 1, COUNTER = 0, PRIORITY = 7]
Task 2: [PID = 2, COUNTER = 0, PRIORITY = 10]
Task 3: [PID = 3, COUNTER = 0, PRIORITY = 4]
Task 4: [PID = 4, COUNTER = 0, PRIORITY = 1]
Task 5: [PID = 5, COUNTER = 0, PRIORITY = 4]
Task 6: [PID = 6, COUNTER = 0, PRIORITY = 7]
Task 7: [PID = 7, COUNTER = 0, PRIORITY = 5]
Task 8: [PID = 8, COUNTER = 0, PRIORITY = 10]
Task 9: [PID = 9, COUNTER = 0, PRIORITY = 1]
Task 10: [PID = 10, COUNTER = 0, PRIORITY = 9]
Task 11: [PID = 11, COUNTER = 0, PRIORITY = 6]
Task 12: [PID = 12, COUNTER = 0, PRIORITY = 9]
Task 13: [PID = 13, COUNTER = 0, PRIORITY = 6]
Task 14: [PID = 14, COUNTER = 0, PRIORITY = 6]
Task 15: [PID = 15, COUNTER = 0, PRIORITY = 5]
Task 16: [PID = 16, COUNTER = 0, PRIORITY = 8]
Task 17: [PID = 17, COUNTER = 0, PRIORITY = 1]
Task 18: [PID = 18, COUNTER = 0, PRIORITY = 5]
Task 19: [PID = 19, COUNTER = 0, PRIORITY = 3]
Task 20: [PID = 20, COUNTER = 0, PRIORITY = 7]
Task 21: [PID = 21, COUNTER = 0, PRIORITY = 7]
Task 22: [PID = 22, COUNTER = 0, PRIORITY = 3]
Task 23: [PID = 23, COUNTER = 0, PRIORITY = 3]
Task 24: [PID = 24, COUNTER = 0, PRIORITY = 3]
Task 25: [PID = 25, COUNTER = 0, PRIORITY = 4]
```



```
Task 31: [PID = 31, COUNTER = 0, PRIORITY = 3]
Checking counters, max_counter = 0
All task counters are 0, resetting counters based on priority...
SET [PID = 1 PRIORITY = 7 COUNTER = 7, STATE = 0]
SET [PID = 2 PRIORITY = 10 COUNTER = 10, STATE = 0]
SET [PID = 3 PRIORITY = 4 COUNTER = 4, STATE = 0]
SET [PID = 4 PRIORITY = 1 COUNTER = 1, STATE = 0]
SET [PID = 5 PRIORITY = 4 COUNTER = 4, STATE = 0]
SET [PID = 6 PRIORITY = 7 COUNTER = 7, STATE = 0]
SET [PID = 7 PRIORITY = 5 COUNTER = 5, STATE = 0]
SET [PID = 8 PRIORITY = 10 COUNTER = 10, STATE = 0]
SET [PID = 9 PRIORITY = 1 COUNTER = 1, STATE = 0]
SET [PID = 10 PRIORITY = 9 COUNTER = 9, STATE = 0]
SET [PID = 11 PRIORITY = 6 COUNTER = 6, STATE = 0]
SET [PID = 12 PRIORITY = 9 COUNTER = 9, STATE = 0]
SET [PID = 13 PRIORITY = 6 COUNTER = 6, STATE = 0]
SET [PID = 14 PRIORITY = 6 COUNTER = 6, STATE = 0]
SET [PID = 15 PRIORITY = 5 COUNTER = 5, STATE = 0]
SET [PID = 16 PRIORITY = 8 COUNTER = 8, STATE = 0]
SET [PID = 17 PRIORITY = 1 COUNTER = 1, STATE = 0]
SET [PID = 18 PRIORITY = 5 COUNTER = 5, STATE = 0]
SET [PID = 19 PRIORITY = 3 COUNTER = 3, STATE = 0]
SET [PID = 20 PRIORITY = 7 COUNTER = 7, STATE = 0]
SET [PID = 21 PRIORITY = 7 COUNTER = 7, STATE = 0]
SET [PID = 22 PRIORITY = 3 COUNTER = 3, STATE = 0]
SET [PID = 23 PRIORITY = 3 COUNTER = 3, STATE = 0]
SET [PID = 24 PRIORITY = 3 COUNTER = 3, STATE = 0]
SET [PID = 25 PRIORITY = 4 COUNTER = 4, STATE = 0]
SET [PID = 26 PRIORITY = 3 COUNTER = 3, STATE = 0]
SET [PID = 27 PRIORITY = 9 COUNTER = 9, STATE = 0]
SET [PID = 28 PRIORITY = 1 COUNTER = 1, STATE = 0]
SET [PID = 29 PRIORITY = 9 COUNTER = 9, STATE = 0]
SET [PID = 30 PRIORITY = 10 COUNTER = 10, STATE = 0]
SET [PID = 31 PRIORITY = 3 COUNTER = 3, STATE = 0]
After resetting counters, max_counter = 10
Selected task 30 with counter 10 for next run.
Switching to task 30 with counter 10
```

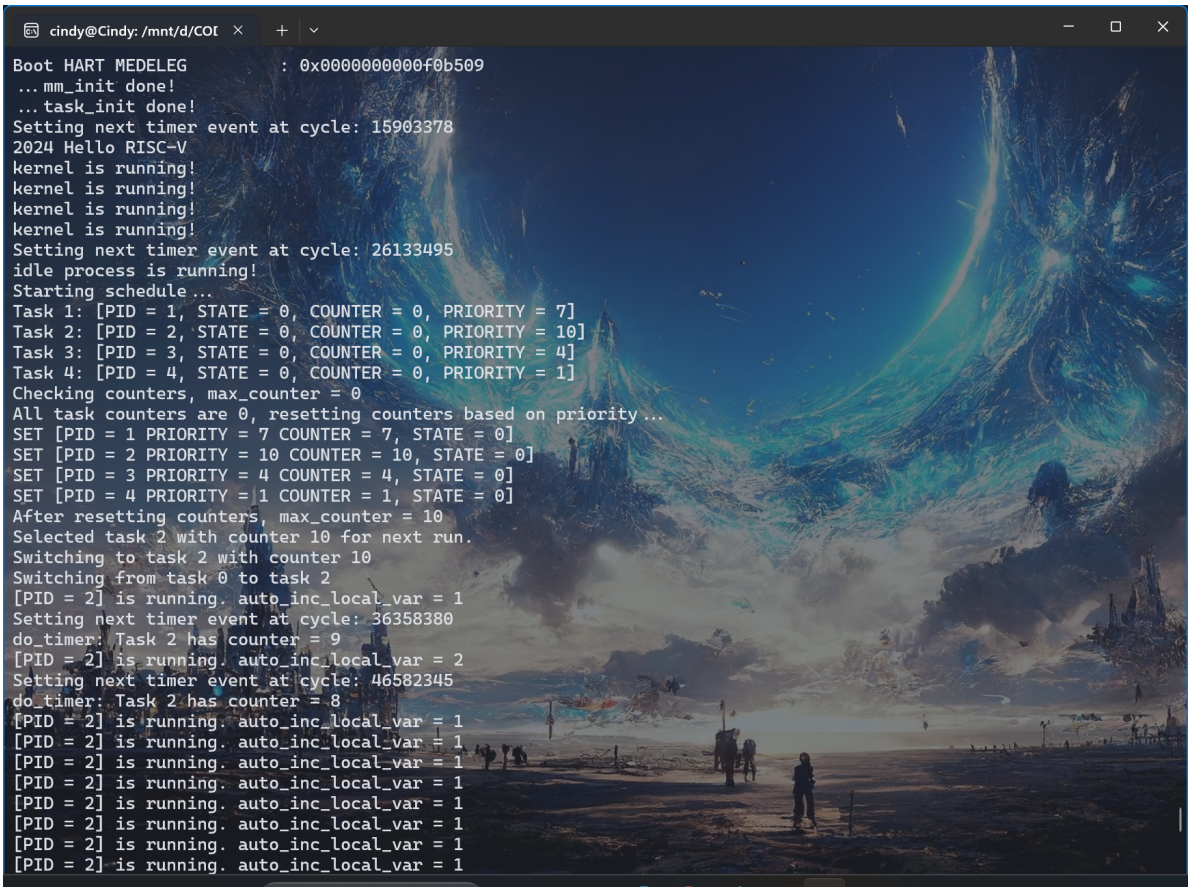


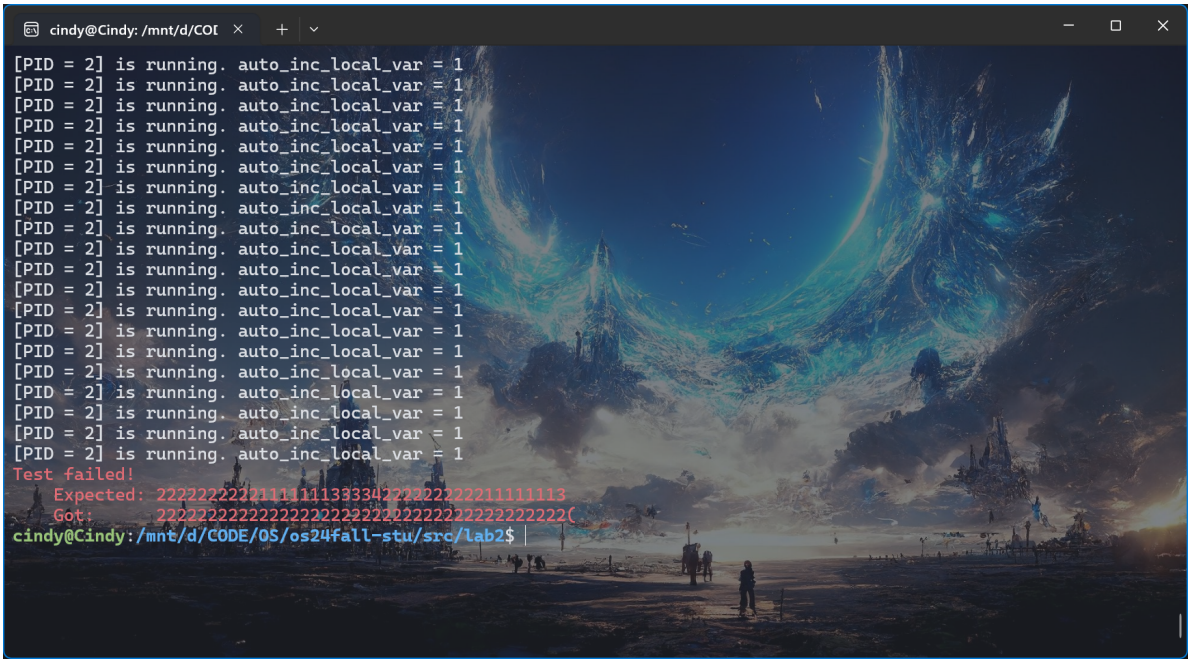
```
Switching from task 0 to task 30 :[PID = 10, COUNTER = 10, PRIORITY = 10]
[PID = 30] is running. auto_inc_local_var = 1
Setting next timer event at cycle: 36219309
do_timer: Task 30 has counter = 9
[PID = 30] is running. auto_inc_local_var = 2
Setting next timer event at cycle: 46245809
do_timer: Task 30 has counter = 8
[PID = 30] is running. auto_inc_local_var = 3
Setting next timer event at cycle: 56270120
do_timer: Task 30 has counter = 7
[PID = 30] is running. auto_inc_local_var = 4
Setting next timer event at cycle: 66290822
do_timer: Task 30 has counter = 6
[PID = 30] is running. auto_inc_local_var = 5
Setting next timer event at cycle: 76309802
do_timer: Task 30 has counter = 5
[PID = 30] is running. auto_inc_local_var = 6
Setting next timer event at cycle: 86331098
do_timer: Task 30 has counter = 4
[PID = 30] is running. auto_inc_local_var = 7
Setting next timer event at cycle: 96347837
do_timer: Task 30 has counter = 3
[PID = 30] is running. auto_inc_local_var = 8
Setting next timer event at cycle: 106363326
do_timer: Task 30 has counter = 2
[PID = 30] is running. auto_inc_local_var = 9
Setting next timer event at cycle: 116376799
do_timer: Task 30 has counter = 1
[PID = 30] is running. auto_inc_local_var = 10
Setting next timer event at cycle: 126390557
Task 30 has counter 0, forcing switch...
Starting schedule...
Task 1: [PID = 1, COUNTER = 7, PRIORITY = 7]
Task 2: [PID = 2, COUNTER = 10, PRIORITY = 10]
Task 3: [PID = 3, COUNTER = 4, PRIORITY = 4]
Task 4: [PID = 4, COUNTER = 1, PRIORITY = 1]
Task 5: [PID = 5, COUNTER = 4, PRIORITY = 4]
```

三、讨论和心得

按实验指导操作时的问题

第一次编译运行test的时候出现了以下情况，线程成功切换但运行两次后就不断重复输出，测试失败





自己针对该问题debug许久无果后询问了助教老师，在修改了一些小问题之后仍然错误。最后通过助教老师的debug，发现前一次lab写的 `__traps` 有问题，导致爆栈。

以下是修改后的代码，主要修改在对于x2寄存器的保存和 `sp` 栈指针变化上：

```
_traps:
    # 1. 保存 31 个寄存器和 sepc 到栈

    addi sp, sp, -264           # 分配栈空间，32 * 8 + 8 = 264 字节

```

```
    sd x0, 0(sp)
    sd x1, 8(sp)

    sd x3, 24(sp)
    sd x4, 32(sp)
    sd x5, 40(sp)
    sd x6, 48(sp)
     ... 省略
    sd x30, 240(sp)
    sd x31, 248(sp)

    csrr t0, sepc              # 读取 sepc
    sd t0, 256(sp)             # 保存 sepc 到栈上

    sd x2, 16(sp)              # 保存 x2

    # 2. 调用 trap_handler
    csrr a0, scause            # scause → a0
    csrr a1, sepc              # sepc → a1
    call trap_handler          # 调用 C 语言中的 trap_handler

    # 3. 恢复 sepc 和 31 个寄存器
    ld t0, 256(sp)             # 恢复 sepc 的值
    csrw sepc, t0              # 将恢复的 sepc 写回 sepc

    ld x0, 0(sp)
    ld x1, 8(sp)
    ld x3, 24(sp)
    ld x4, 32(sp)
     ... 省略
    ld x30, 240(sp)
    ld x31, 248(sp)

    ld x2, 16(sp)              # 恢复原来的栈顶指针

    addi sp, sp, 264           # 调整栈指针恢复

    # 4. 从 trap 返回
    sret
```

心得

1. 编写汇编指令的时候要注意保存/传入参数的地址
2. 面对比较复杂的工程，我们不能太相信自己以前写的代码，bug可能隐藏其中

其他

`sepc`：程序异常计数器，是存储终端异常发生的PC值

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?

在上下文切换中，我们通常只保存 **线程切换前和切换后所需要用到的最小寄存器集**。在 RISC-V 的上下文切换实现中，保存的寄存器通常包括：

- `ra`：返回地址。
- `sp`：栈指针。
- `s0` - `s11`：保存寄存器（`s0` 到 `s11`，共 12 个）。

这些寄存器在上下文切换时需要保存和恢复，以确保线程执行时的上下文状态保持一致。这样做不仅能确保上下文切换的完整性，还能减少不必要的寄存器保存和恢复，提高上下文切换的效率。

其他寄存器：`t0` - `t6` 是临时寄存器（caller-saved），它们通常在函数调用中可以被随意使用，因此在线程上下文切换时不需要保存；`a0` - `a7` 这些寄存器用于传递函数参数和返回值，因此在上下文切换时不需要保存。

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()` 函数解析

`mm_init()` 函数用于内存管理模块的初始化。该函数的作用是通过 `kfreerange` 函数，将操作系统内核结束地址 `_ekernel` 到物理内存结束地址 `PHY_END` 之间的所有内存标记为可用的物理页，并将这些物理页加入到空闲内存链表 `kmem.freelist` 中。

```c
void mm_init(void) {
    kfreerange(_ekernel, (char *)PHY_END);
    printk("...mm_init done!\n");
}
```

其中：

- `_ekernel`：表示内核结束的地址（全局符号）。内核代码、数据和栈的内存空间使用完之后，`_ekernel` 就是内核空间的结束地址，通常内核初始化之后，这个地址指向内核之外的第一个未使用的物理页的起始地址。
- `PHY_END`：表示物理内存的结束地址。该地址指定了系统中可用的最大物理内存地址（例如 64 MiB 或 128 MiB）。
- `kfreerange` 函数：批量释放内存页

```c
void kfreerange(char *start, char *end) {
    char *addr = (char *)PGROUNDUP((uintptr_t)start);
    for (; (uintptr_t)(addr) + PGSIZE ≤ (uintptr_t)end; addr += PGSIZE) {
        kfree((void *)addr);
    }
}
```

1. 使用 `PGROUNDUP` 对 `start` 地址（内存起始地址）进行对齐，确保 `start` 地址是一个页对齐的地址。
2. 遍历 `start` 到 `end` （内存结束地址）之间的每个物理页地址，并调用 `kfree()` 将其释放。
3. 每次循环，`addr += PGSIZE`，将指针移动到下一个物理页的起始地址，直到超出 `end` 地址。

`mm_init()` 调用了 `kfreerange` 来将 `_ekernel` 到 `PHY_END` 之间的内存初始化为可用内存块。该函数作用是将指定地址范围（对齐到页边界）内的所有页依次调用 `kfree()` 函数，将它们加入到 `kmem.freelist` （即内存空闲链表）中。

内存管理的核心操作函数：`kalloc()` 和 `kfree()`

1. `kalloc()` 函数：分配一个空闲物理页

```
void *kalloc() {
    struct run *r;

    r = kmem.freelist;
    kmem.freelist = r->next;

    memset((void *)r, 0x0, PGSIZE);
    return (void *)r;
}
```

- **分配过程：**
  1. 从 `kmem.freelist` 中取出第一个空闲物理页（`kmem.freelist` 指向的是一个 `run` 结构体的链表）。
  2. 将 `kmem.freelist` 指向下一个空闲页（`kmem.freelist = r->next`）。
  3. 将所分配的页清零（`memset((void *)r, 0x0, PGSIZE)`），确保返回的内存页是干净的（初始化为 0）。
  4. 返回该页的地址（`return (void *)r`）。
- **返回值：**
  返回一个指向 4 KiB（`PGSIZE`）大小的内存页的指针。

2. `kfree()` 函数：释放一个物理页

```
void kfree(void *addr) {
    struct run *r;

    // PGSIZE align
    *(uintptr_t *)&addr = (uintptr_t)addr & ~(PGSIZE - 1);

    memset(addr, 0x0, (uint64_t)PGSIZE);

    r = (struct run *)addr;
    r->next = kmem.freelist;
    kmem.freelist = r;

    return;
```

```
}
```

- **释放过程：**
  1. **对齐地址**：对传入的 `addr` 进行页对齐（`addr & ~(PGSIZE - 1)`）。页对齐是为了确保释放的地址是一个合法的页起始地址（例如，释放操作仅能释放 4 KiB 对齐的地址）。
  2. **清零内存页**：将整个内存页置零（`memset(addr, 0x0, PGSIZE)`），确保释放的内存页不会残留数据。
  3. **将页加入到空闲链表中**：将该页的地址转换为 `struct run *` 类型，并将它插入到 `kmem.freelist` 链表头部。
- **空闲链表的更新：**
  - 通过 `r->next = kmem.freelist;` 将当前页的 `next` 指向原链表的头结点。
  - 然后通过 `kmem.freelist = r;` 将链表头部更新为当前页。

3. `struct run` 数据结构

`struct run` 是一个简单的链表结构体，用于表示一个空闲内存页。

```
struct run {
    struct run *next;
};
```

- `next` 指针用于将各个空闲页链接起来，形成一个单向链表。

内存管理的整体流程

1. **内存初始化（`mm_init()`）**
   - 通过 `kfreerange()` 函数，将内核结束地址 `_ekernel` 到物理内存结束地址 `PHY_END` 之间的所有物理页加入到空闲链表 `kmem.freelist` 中，形成一个单向链表。
2. **内存分配（`kalloc()`）**
   - 从 `kmem.freelist` 中取出链表头部的页，并将 `kmem.freelist` 更新为下一个节点（`r->next`）。
   - 将分配的页清零，并返回该页的地址。
3. **内存释放（`kfree()`）**

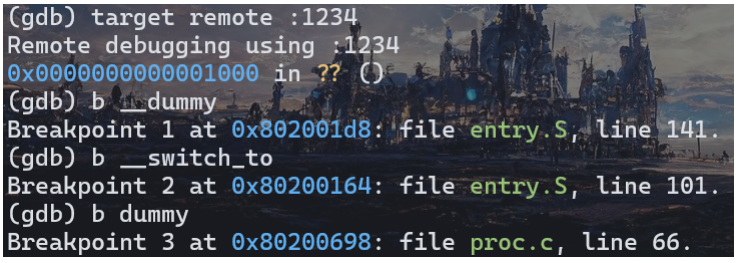
- 将要释放的页进行页对齐，并置零该页。
- 将其加入到 `kmem.freelist` 链表头部，更新空闲链表。

- **内存管理的核心机制：**
  - 利用单向链表 `kmem.freelist` 作为空闲页管理数据结构，每个空闲页由一个 `struct run` 结构体表示。
  - 内存初始化时，将内核结束地址 `_ekernel` 到物理内存结束地址 `PHY_END` 之间的物理页都标记为空闲，并加入到 `kmem.freelist` 中。
  - `kalloc()` 函数从链表中取出一个空闲页，并将其返回给用户。
  - `kfree()` 函数将释放的页重新加入到空闲链表中，以便后续分配时再次使用。

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）。

在一个终端中运行 `make debug`，在另一个终端运行以下命令：

```
# 第二个终端
gdb-multiarch /mnt/d/CODE/OS/os24fall-stu/src/lab2/vmlinux

# 连接
(gdb) target remote :1234
# 设置断点
(gdb) b __dummy
(gdb) b __switch_to
(gdb) b dummy
```



第一次运行 `c` ，碰到断点2，内核执行到 `__switch_to` 。我们使用以下命令查看 `ra` 的值： `(gdb) info registers ra`

```
(gdb) c
Continuing.

Breakpoint 2, __switch_to () at entry.S:101
101         sd ra, 32(a0)          # 保存 ra 到 prev->thread_struct.ra
(gdb) info registers ra
ra             0x80200800       0x80200800 <switch_to+148>
```

此时当前 `ra` 寄存器指向的是 `switch_to` 函数的返回地址（0x80200800）。在 `__switch_to` 执行时，它会保存当前线程的 `ra` 寄存器值到 `prev->thread.ra` 中。

第二次运行，碰到断点1，内核执行到 `__dummy` ，查看当前 `ra` 的值：

```
(gdb) c
Continuing.

Breakpoint 1, __dummy () at entry.S:141
141         la      t0, dummy     // 将 dummy 地址加载到 t0 寄存器中
(gdb) info registers ra
ra             0x802001d8       0x802001d8 <__dummy>
```

此时 `ra` 寄存器被设置为 `dummy` 函数的地址，以便 `sret` 指令返回时，能够跳转到 `dummy` 函数开始执行。这次调度的 `ra` 是一个特殊的返回地址，用于引导新线程进入 `dummy` 函数。

第三次运行，碰到断点3，内核执行到 `dummy` ：

```
(gdb) c
Continuing.

Breakpoint 3, dummy () at proc.c:66
66          uint64_t MOD = 1000000007;
(gdb) info registers ra
ra             0x802001d8       0x802001d8 <__dummy>
```

`ra` 仍然保持为 `0x802001d8` ，表示当前线程被恢复时，如果该线程再次返回，会回到 `__dummy` 。当 `__dummy` 中的 `sret` 指令被执行时，程序计数器跳转到 `dummy` 函数。

第四次运行，再次碰到断点2，内核执行到 `__switch_to` ，至此已经完成一次完整的上下文切换。

```
Breakpoint 3, dummy () at proc.c:66
66          uint64_t MOD = 1000000007;
(gdb) info registers ra
ra             0x802001d8       0x802001d8 <__dummy>
(gdb) c
Continuing.

Breakpoint 2, __switch_to () at entry.S:101
101         sd ra, 32(a0)          # 保存 ra 到 prev->thread_struct.ra
(gdb) info registers ra
ra             0x80200800       0x80200800 <switch_to+148>
```

`ra` 在 `__switch_to` 中被保存到上一个线程的 `prev->thread.ra` 中。这里的 `ra` 值指向 `switch_to` 的返回地址（即从 `switch_to` 返回到调度器的地方）。这表示，当 `__switch_to` 执行完上下文切换后，返回地址会跳回到 `switch_to` 函数中继续执行。

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

```
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default -S -s
```

设置断点：

```
(gdb) target remote: 1234
Remote debugging using : 1234
0x0000000000001000 in ?? ()
(gdb) b __switch_to
Breakpoint 1 at 0x80200164: file entry.S, line 101.
(gdb) b dummy
Breakpoint 2 at 0x80200698: file proc.c, line 66.
(gdb) b do_timer
Breakpoint 3 at 0x8020082c: file proc.c, line 117.
(gdb) b schedule
Breakpoint 4 at 0x80200930: file proc.c, line 142.
(gdb) b trap_handler
Breakpoint 5 at 0x80200f34: file trap.c, line 16.
```

当第一个时钟中断触发时，`trap_handler` 会被执行，并跳转到 `do_timer` 中：

```
(gdb) info registers sp pc   # 查看当前栈指针和程序计数器
```

```
(gdb) c
Continuing.

Breakpoint 5, trap_handler (scause=9223372036854775813, sepc=2149585116) at trap.c:16
16          __asm__ volatile("csrr %0, stval" : "=r"(stval));
(gdb) info registers sp pc
sp             0x80205e98       0x80205e98
pc             0x80200f34       0x80200f34 <trap_handler+24>
```

继续运行，直到 `schedule()` 被调用，使用 `backtrace` 命令查看当前栈中的调用关系：

```
(gdb) c
Continuing.

Breakpoint 3, do_timer () at proc.c:117
117         if (current == idle || current->pid == 0) {
(gdb) c
Continuing.

Breakpoint 4, schedule () at proc.c:142
142         uint64_t max_counter = 0;
(gdb) bt
#0  schedule () at proc.c:142
#1  0x000000008020086c in do_timer () at proc.c:119
#2  0x0000000080200f64 in trap_handler (scause=9223372036854775813, sepc=2149585116) at trap.c:25
#3  0x00000000802000d4 in _traps () at entry.S:54
Backtrace stopped: frame did not save the PC
(gdb) info registers sp pc
sp             0x80205e58       0x80205e58
pc             0x80200930       0x80200930 <schedule+16>
```

说明：

- 当前 `schedule` 运行在 `proc.c` 的第 141 行。
- `schedule` 是从 `do_timer` 中被调用的，而 `do_timer` 则是从 `trap_handler` 中被调用的。
- `trap_handler` 又是通过 `_traps` （`entry.S` 中的汇编处理程序）调用的。
- `_traps` 中的栈帧没有保存 `PC` ，因此无法进一步回溯到 `trap_handler` 之前的调用（这通常是由于 `_traps` 是一个手动控制的汇编中断处理程序）。

继续运行到 `__switch_to` 的断点，使用 `x/20x $sp` 命令查看栈中的数据（当前栈指针指向的内存内容）：

```
(gdb) c
Continuing.

Breakpoint 1, __switch_to () at entry.S:101
101         sd ra, 32(a0)           # 保存 ra 到 prev->thread_struct.ra
(gdb) info registers sp pc
sp             0x80205e28       0x80205e28
pc             0x80200164       0x80200164 <__switch_to>
(gdb) x/20x $sp
0x80205e28:     0x0000000a      0x00000000      0x87fe1000      0x00000000
0x80205e38:     0x0000001e      0x00000000      0x87fff000      0x00000000
0x80205e48:     0x80205e88      0x00000000      0x80200ccc      0x00000000
0x80205e58:     0x00000000      0x00000020      0x00000020      0x00000020
0x80205e68:     0x87fe1000      0x00000000      0x0000000a      0x00000000
```

接着继续运行，经过一个 `dummy` 的断点和多个 `trap_handler` 和 `do_timer` 的断点，直到再次遇到schedule的断点

```
(gdb) c
Continuing.

Breakpoint 4, schedule () at proc.c:142
142         uint64_t max_counter = 0;
(gdb) bt
#0  schedule () at proc.c:142
#1  0x000000008020090c in do_timer () at proc.c:134
#2  0x0000000080200f64 in trap_handler (scause=9223372036854775813, sepc=2149582520) at trap.c:25
#3  0x00000000802000d4 in _traps () at entry.S:54
Backtrace stopped: frame did not save the PC
(gdb) info register sp pc
sp             0x87fe1e58       0x87fe1e58
pc             0x80200930       0x80200930 <schedule+16>
```

再接着运行到 `__switch_to` 的断点，使用 `x/20x $sp` 命令查看栈中的数据（当前栈指针指向的内存内容）：

```
(gdb) c
Continuing.

Breakpoint 1, __switch_to () at entry.S:101
101         sd ra, 32(a0)           # 保存 ra 到 prev->thread_struct.ra
(gdb) x/20x $sp
0x87fe1e28:     0x0000000a      0x00000000      0x87ff7000      0x00000000
0x87fe1e38:     0x00000008      0x00000000      0x87fe1000      0x00000000
0x87fe1e48:     0x87fe1e88      0x00000000      0x80200ccc      0x00000000
0x87fe1e58:     0x00000000      0x00000000      0x00000000      0x00000020
0x87fe1e68:     0x87ff7000      0x00000000      0x0000000a      0x00000000
```

使用 `info frame` 和 `info register` 查看当前帧的详细信息和所有寄存器值：

```
(gdb) info frame
Stack level 0, frame at 0x87fe1e28:
 pc = 0x80200164 in __switch_to (entry.S:101); saved pc = 0x8020080c
 called by frame at 0x87fe1e58
 source language asm.
 Arglist at 0x87fe1e28, args:
 Locals at 0x87fe1e28, Previous frame's sp is 0x87fe1e28
(gdb) info registers
ra             0x8020080c       0x8020080c <switch_to+160>
sp             0x87fe1e28       0x87fe1e28
gp             0x0      0x0
tp             0x80047000       0x80047000
t0             0x0      0
t1             0x0      0
t2             0x0      0
fp             0x87fe1e58       0x87fe1e58
s1             0x0      0
a0             0x87fe1000       2281574400
a1             0x87ff7000       2281664512
a2             0x0      0
a3             0x0      0
a4             0x87ff7000       2281664512
a5             0x80206010       2149605392
a6             0x2      2
a7             0x4442434e       1145193294
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
s8             0x0      0
s9             0x0      0
s10            0x0      0
s11            0x0      0
t3             0x0      0
t4             0x0      0
```

以下是一个栈布局图示例：

```
+-----------------+    ← 上一个线程的栈顶（task[0]）
|       ...       |
| saved ra        |    ← task[0] 的返回地址
| saved s0        |    ← 保存的 s0
| saved s1        |    ← 保存的 s1
|       ...       |    ← 更多保存寄存器 s2 ~ s11
| saved s11       |
| prev_sp         |    ← task[0] 的 sp（恢复时使用）
+-----------------+    ← __switch_to() 中的栈帧（保存 prev 线程上下文）
| saved ra (schedule)|  ← schedule 中调用 __switch_to 的返回地址
| saved sp        |    ← 当前调度器的 sp
+-----------------+    ← 当前线程的栈帧起始位置（task[1]）
|       ...       |
| dummy() frame   |    ← dummy() 中的栈帧
| saved ra        |    ← ra = 0x802001d8 (__dummy)
| task[1] sp      |    ← 新线程的 sp
+-----------------+
```

根据上述得到的 `__switch_to` 中的栈帧布局，以及调用栈的详细信息，我们可以画出一个简单的栈帧布局图：

```
栈地址（从上到下增加）
+-------------------------------+  ← 上一个线程（task[0]）的栈顶 `sp = 0x80205e28`
| saved ra = 0x80200ccc         |  ← 返回地址：`task[0]` 的返回地址（某函数）
| saved s0                      |  ← 保存的 `s0` 寄存器值
| saved s1                      |  ← 保存的 `s1` 寄存器值
| ...                           |  ← 更多保存的寄存器（s2 ~ s11）
| saved s11                     |
| prev sp = 0x87fe1000          |  ← `task[0]` 的 `sp`（用于恢复上下文）
+-------------------------------+
| __switch_to() 栈帧            |  ← `__switch_to` 中保存的上下文
| saved ra (0x8020080c)         |   ← 保存的 `ra`，`__switch_to` 的返回地址
| current sp = 0x87fe1e28       |   ← `__switch_to` 执行时的 `sp` 值
+-------------------------------+
| task[1] 栈帧                  |   ← `task[1]` 的栈帧起始位置 `sp = 0x87fe1e28`
| dummy() frame                 |   ← `dummy` 函数栈帧，记录 `dummy` 函数上下文
| saved ra (0x802001d8)         |   ← `dummy` 函数返回地址（指向 `__dummy`）
| task[1] sp = 0x87ff7000       |   ← 新线程的 `sp`（`task[1]` 的栈顶）
```

```
+-------------------------------+
```

五、附录

参考 实验指导二 - 浙江大学24年秋冬操作系统实验 (zju-sec.github.io)

有所帮助：助教老师以及ChatGPT