Lab 3: RV64 内核线程调度

姓名：巩德志　　学号：3200105088

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 背景知识

3.0 前言

在 lab2 中, 我们利用 trap 赋予了 OS 与软件, 硬件的交互能力。但是目前我们的 OS 还不具备多进程调度以及并发执行的能力。在本次实验中, 我们将利用时钟中断, 来实现多进程的调度以使得多个进程/线程并发执行。

3.1 进程与线程

`源代码`经编译器一系列处理（编译、链接、优化等）后得到的可执行文件, 我们称之为`程序 (Program)`。而通俗地说, `进程`就是`正在运行并使用计算机资源`的程序。`进程`与`程序`的不同之处在于, `进程`是一个动态的概念, 其不仅需要将其运行的程序的代码/数据等加载到内存空间中, 还需要拥有自己的`运行栈`。同时一个`进程`可以对应一个或多个`线程`, `线程`之间往往具有相同的代码, 共享一块内存, 但是却有不同的CPU执行状态。

在本次实验中, 为了简单起见, 我们采用 `single-threaded process` 模型, 即`一个进程`对应`一个线程`, 进程与线程不做明显区分。

3.1 线程相关属性

在不同的操作系统中, 为每个线程所保存的信息都不同。在这里, 我们提供一种基础的实现, 每个线程会包括：

- `线程ID`： 用于唯一确认一个线程。
- `运行栈`： 每个线程都必须有一个独立的运行栈, 保存运行时的数据。
- `执行上下文`： 当线程不在执行状态时, 我们需要保存其上下文（其实就是`状态寄存器`的值）, 这样之后才能够将其恢复, 继续运行。
- `运行时间片`： 为每个线程分配的运行时间。
- `优先级`： 在优先级相关调度时, 配合调度算法, 来选出下一个执行的线程。

3.2 线程切换流程图

- 在每次处理时钟中断时，操作系统首先会将当前线程的运行剩余时间减少一个单位。之后根据调度算法来确定是继续运行还是调度其他线程来执行。
- 在进程调度时，操作系统会遍历所有可运行的线程，按照一定的调度算法选出下一个执行的线程。最终将选择得到的线程与当前线程切换。
- 在切换的过程中，首先我们需要保存当前线程的执行上下文，再将将要执行线程的上下文载入到相关寄存器中，至此我们就完成了线程的调度与切换。

4 实验步骤

4.1 准备工程

- 此次实验基于 lab2 同学所实现的代码进行。
- 在 lab3 中需要同学需要添加并修改 `arch/riscv/include/proc.h` `arch/riscv/kernel/proc.c` 两个文件。

4.2 `proc.h` 数据结构定义

```
// arch/riscv/include/proc.h

#include "types.h"

#define NR_TASKS  (1 + 31) // 用于控制 最大线程数量 （idle 线程 + 31 内核线程）

#define TASK_RUNNING    0 // 为了简化实验，所有的线程都只有一种状态

#define PRIORITY_MIN 1
#define PRIORITY_MAX 10

/* 用于记录 `线程` 的 `内核栈与用户栈指针` */
/* (lab3中无需考虑，在这里引入是为了之后实验的使用) */
struct thread_info {
    uint64 kernel_sp;
    uint64 user_sp;
};

/* 线程状态段数据结构 */
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];
};

/* 线程数据结构 */
struct task_struct {
    struct thread_info* thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id

    struct thread_struct thread;
};
```

```
/* 线程初始化 创建 NR_TASKS 个线程 */
void task_init();

/* 在时钟中断处理中被调用 用于判断是否需要进行调度 */
void do_timer();

/* 调度程序 选择出下一个运行的线程 */
void schedule();

/* 线程切换入口函数*/
void switch_to(struct task_struct* next);

/* dummy funciton: 一个循环程序, 循环输出自己的 pid 以及一个自增的局部变量 */
void dummy();
```

4.3 线程调度功能实现

4.3.1 线程初始化

在初始化线程时以 4KB 为粒度，按照每个 Task 一帧的形式进行分配，并将 `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。

首先要为 `idle` 设置 `task_struct`，并将 `current` 和 `task[0]` 都指向 `idle`。

然后要将 `task[1]` ~ `task[NR_TASKS - 1]` 全部初始化，和 `idle` 设置的区别在于要为这些线程设置 `thread_struct` 中的 `ra` 和 `sp`。ra被初始化为dummy，sp指向页的最高处地址

```
void task_init() {
    // 1. 调用 kalloc() 为 idle 分配一个物理页
    idle=(struct task_struct*)kalloc();
    // 2. 设置 state 为 TASK_RUNNING;
    idle->state=TASK_RUNNING;
    // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
    idle->counter=0;
    idle->priority=0;
    // 4. 设置 idle 的 pid 为 0
    idle->pid=0;
    // 5. 将 current 和 task[0] 指向 idle
    current=idle;
    task[0]=idle;


    for (int i=1;i<=NR_TASKS-1;i++){
        task[i]=(struct task_struct*)kalloc();
        task[i]->state=TASK_RUNNING;
        task[i]->counter=0;
        task[i]->priority=rand()%PRIORITY_MAX+1;
        task[i]->pid=i;
        task[i]->thread.ra=(uint64)__dummy;
        task[i]->thread.sp=(uint64)task[i]+PGSIZE-1;
    }

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

4.3.2 `__dummy` 与 `dummy` 介绍

- `task[1]` ~ `task[NR_TASKS - 1]` 都运行同一段代码 `dummy()` 我们在 `proc.c` 添加 `dummy()`:

  ```
  void dummy() {
      uint64 MOD = 1000000007;
      uint64 auto_inc_local_var = 0;
      int last_counter = -1;
      while(1) {
          if (last_counter == -1 || current->counter != last_counter) {
              last_counter = current->counter;
              auto_inc_local_var = (auto_inc_local_var + 1) % MOD;
              printk("[PID = %d] is running. auto_inc_local_var = %d\n",
  current->pid, auto_inc_local_var);
          }
      }
  }
  ```

- 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。

    ```
    __dummy:
        # YOUR CODE HERE
        la t0, dummy
        csrw sepc, t0
        sret
    ```

4.3.3 实现线程切换

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。

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

void switch_to(struct task_struct* next) {
    /* YOUR CODE HERE */
    if (next==current)
        return;
    else {
#ifdef DSJF
        printk("\n");
        printk("switch to [PID = %d COUNTER = %d]\n", next->pid, next->counter);
#endif

#ifdef DPRIORITY
        printk("\n");
```

```
        printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n", next->pid,
next->priority, next->counter);
#endif
        struct task_struct* prev = current;
        current=next;
        __switch_to(prev,next);
    }
}
```

```
__switch_to:
    # save state to prev process
    # YOUR CODE HERE
    add t0, a0, 40  # offset of thread_struct in task_struct
    sd ra, 0(t0)
    sd sp, 8(t0)
    sd s0, 16(t0)
    sd s1, 24(t0)
    sd s2, 32(t0)
    sd s3, 40(t0)
    sd s4, 48(t0)
    sd s5, 56(t0)
    sd s6, 64(t0)
    sd s7, 72(t0)
    sd s8, 80(t0)
    sd s9, 88(t0)
    sd s10, 96(t0)
    sd s11, 104(t0)
    # restore state from next process
    # YOUR CODE HERE
    add t0, a1, 40  # offset of thread_struct next task_struct
    ld ra, 0(t0)
    ld sp, 8(t0)
    ld s0, 16(t0)
    ld s1, 24(t0)
    ld s2, 32(t0)
    ld s3, 40(t0)
    ld s4, 48(t0)
    ld s5, 56(t0)
    ld s6, 64(t0)
    ld s7, 72(t0)
    ld s8, 80(t0)
    ld s9, 88(t0)
    ld s10, 96(t0)
    ld s11, 104(t0)

    ret
```

注意在存取和恢复thread_struct之前，需要有8*5个字节的偏移，原因是task_struct中还有五个八字节的变量

```
struct task_struct {
    struct thread_info* thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id

    struct thread_struct thread;
};
```

4.3.4 实现调度入口函数

实现 `do_timer()`，并在 `时钟中断处理函数` 中调用。

```
void do_timer(void) {
    // 1. 如果当前线程是 idle 线程 直接进行调度
    // 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
    if (current==idle)
        schedule();
    else{
        current->counter--;
        if (current->counter>0)
            return;
        else
            schedule();
    }
    /* YOUR CODE HERE */
}
```

4.3.5 实现线程调度

本次实验我们需要实现两种调度算法：1.短作业优先调度算法，2.优先级调度算法。

4.3.5.1 短作业优先调度算法

- 当需要进行调度时按照一下规则进行调度：
    - 遍历线程指针数组 `task` (不包括 `idle`，即 `task[0]`)，在所有运行状态 (`TASK_RUNNING`) 下的线程运行剩余时间 `最少` 的线程作为下一个执行的线程。
    - 如果 `所有` 运行状态下的线程运行剩余时间都为0，则对 `task[1]` ~ `task[NR_TASKS-1]` 的运行剩余时间重新赋值 (使用 `rand()`)，之后再重新进行调度。

```
void schedule(void)
{
    struct task_struct *next = current;
    uint64 min_counter = 0xFFFFFFFFFFFFFFFF;
    while (1)
    {
        //遍历线程指针数组task(不包括 idle , 即 task[0] ), 在所有运行状态(TASK_RUNNING) 下的线程运行剩余时间最少的线程作为下一个执行的线程。
        for (int i = 1; i < NR_TASKS; i++)
        {
            if (task[i]->state == TASK_RUNNING && task[i]->counter > 0)
            {
                if (task[i]->counter < min_counter)
                {
                    min_counter = task[i]->counter;
                    next = task[i];
                }
            }
        }
        //如果所有运行状态下的线程运行剩余时间都为0, 则对 task[1] ~ task[NR_TASKS-1]
的运行剩余时间重新赋值 (使用 rand()) , 之后再重新进行调度。
        if (min_counter == 0xFFFFFFFFFFFFFFFF)
        {
            for (int i = 1; i < NR_TASKS; i++)
            {
                printk("\n");
                task[i]->counter = rand() % 10 + 2;
                printk("SET [PID = %d COUNTER = %d]\n", task[i]->pid,
task[i]->counter);
            }
        }
        else
            break;
    }
    switch_to(next);
}
```

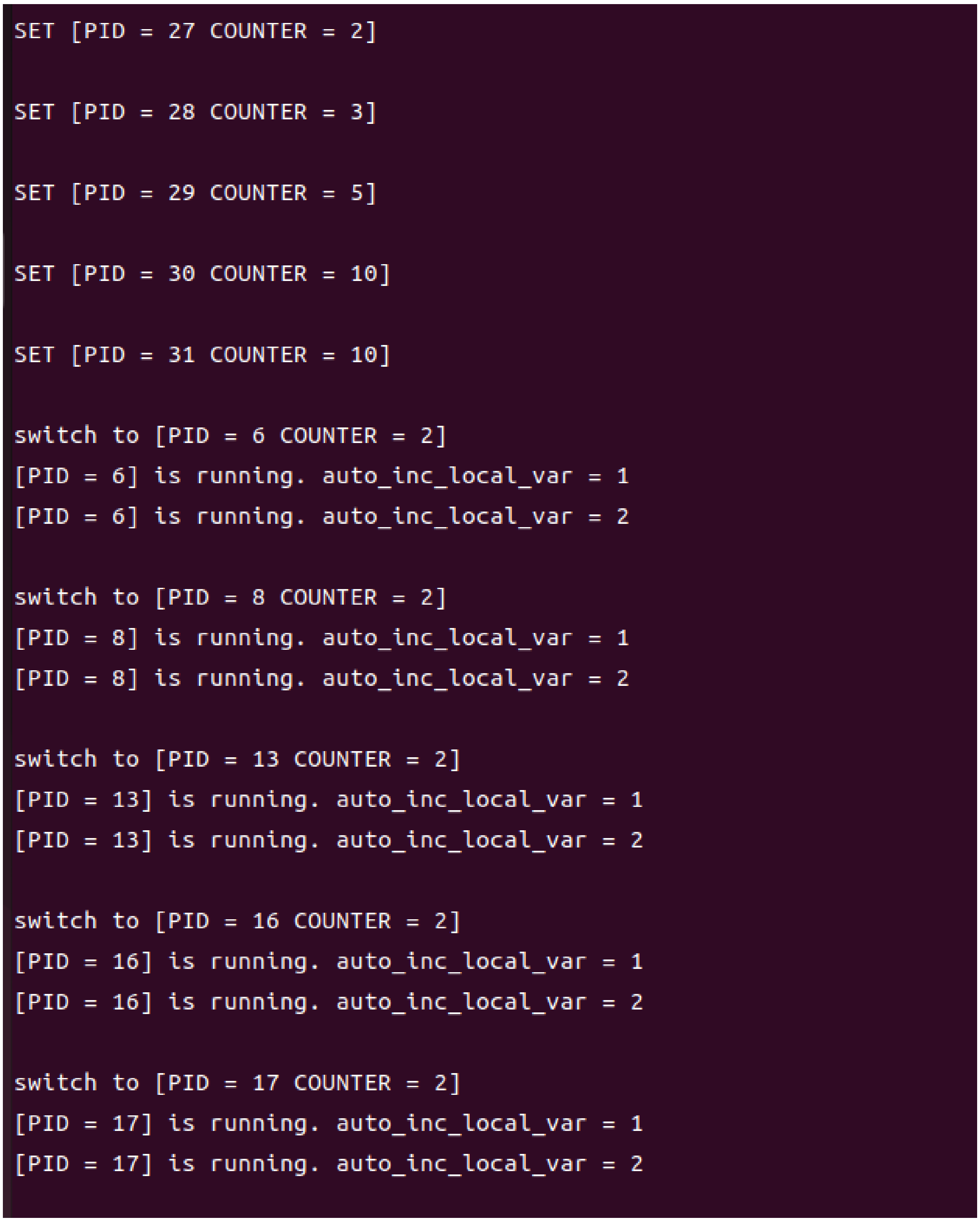
4.3.5.2 优先级调度算法

与短作业优先调度算法的唯一区别在于，将运行状态下的线程优先级最高的线程作为下一个执行的线程

```
void schedule(void)
{
    struct task_struct *next = current;
    uint64 max_priority = PRIORITY_MIN - 1;
    while (1)
    {
        //将优先级最高的线程作为下一个执行的线程
        for (int i = 1; i < NR_TASKS; i++)
        {
            if (task[i]->state == TASK_RUNNING && task[i]->counter > 0)
            {
                if (task[i]->priority > max_priority)
                {
                    max_priority = task[i]->priority;
                    next = task[i];
                }
            }
        }
        //运行剩余时间都为0,重新赋值再进行调度
        if (max_priority == PRIORITY_MIN - 1)
        {
            printk("\n");
            for (int i = 1; i < NR_TASKS; i++)
            {
                task[i]->counter = rand() % 10 + 2;
                printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]-
>pid, task[i]->priority, task[i]->counter);
            }
        }
        else
            break;
    }
    switch_to(next);
}
```

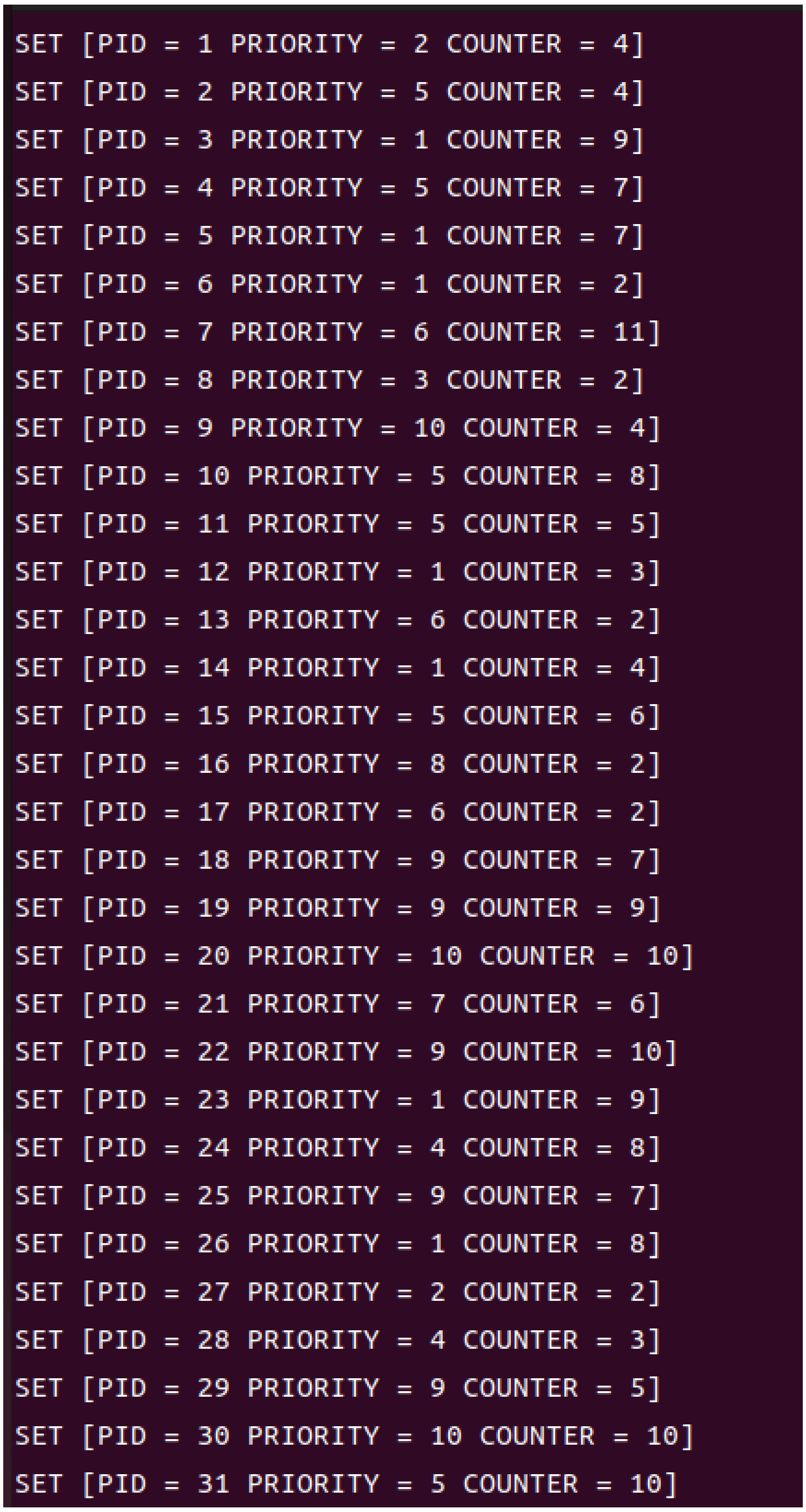
4.4 编译及测试

短作业优先调度

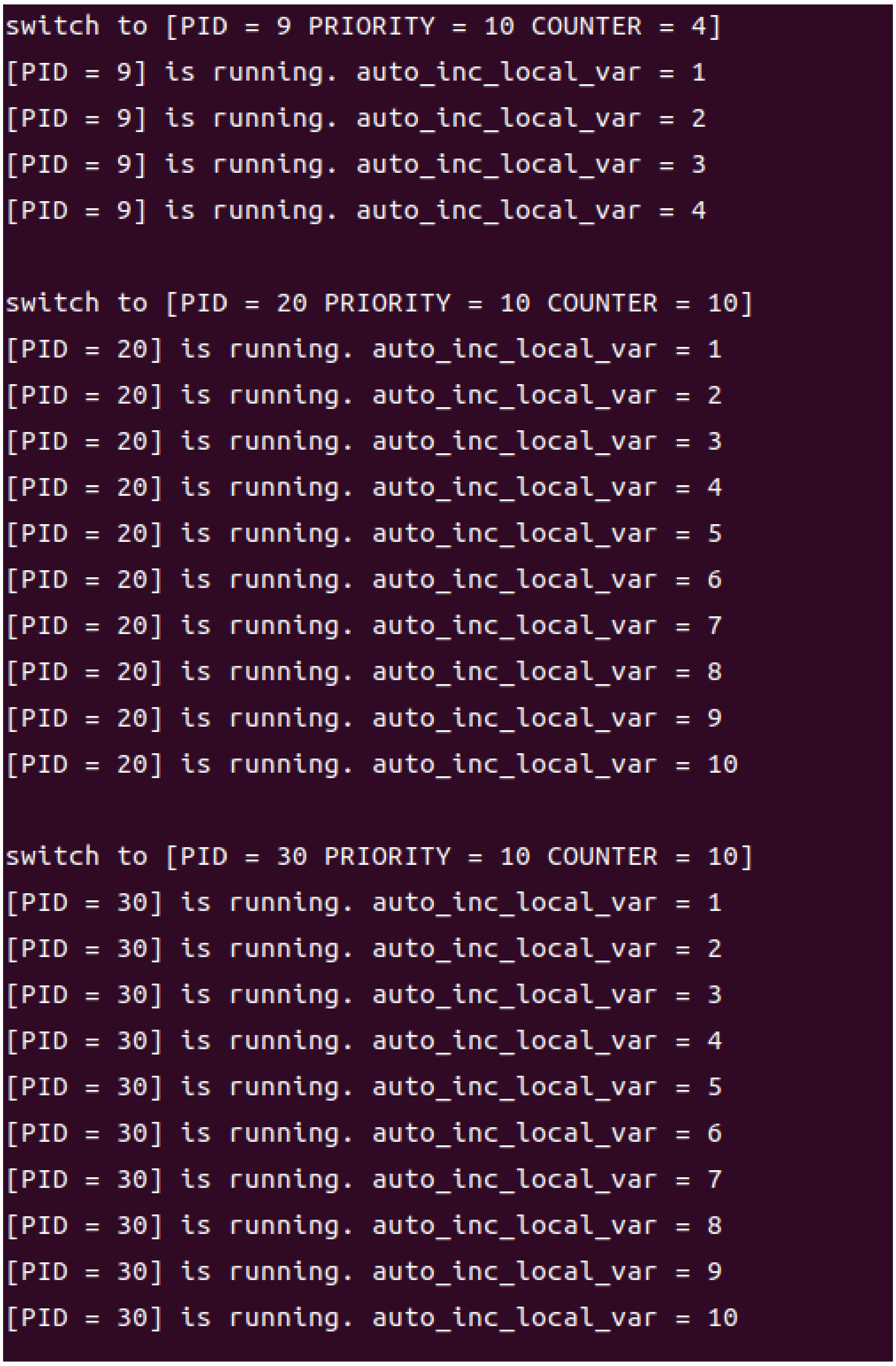


```
SET [PID = 27 COUNTER = 2]

SET [PID = 28 COUNTER = 3]

SET [PID = 29 COUNTER = 5]

SET [PID = 30 COUNTER = 10]

SET [PID = 31 COUNTER = 10]

switch to [PID = 6 COUNTER = 2]
[PID = 6] is running. auto_inc_local_var = 1
[PID = 6] is running. auto_inc_local_var = 2

switch to [PID = 8 COUNTER = 2]
[PID = 8] is running. auto_inc_local_var = 1
[PID = 8] is running. auto_inc_local_var = 2

switch to [PID = 13 COUNTER = 2]
[PID = 13] is running. auto_inc_local_var = 1
[PID = 13] is running. auto_inc_local_var = 2

switch to [PID = 16 COUNTER = 2]
[PID = 16] is running. auto_inc_local_var = 1
[PID = 16] is running. auto_inc_local_var = 2

switch to [PID = 17 COUNTER = 2]
[PID = 17] is running. auto_inc_local_var = 1
[PID = 17] is running. auto_inc_local_var = 2
```

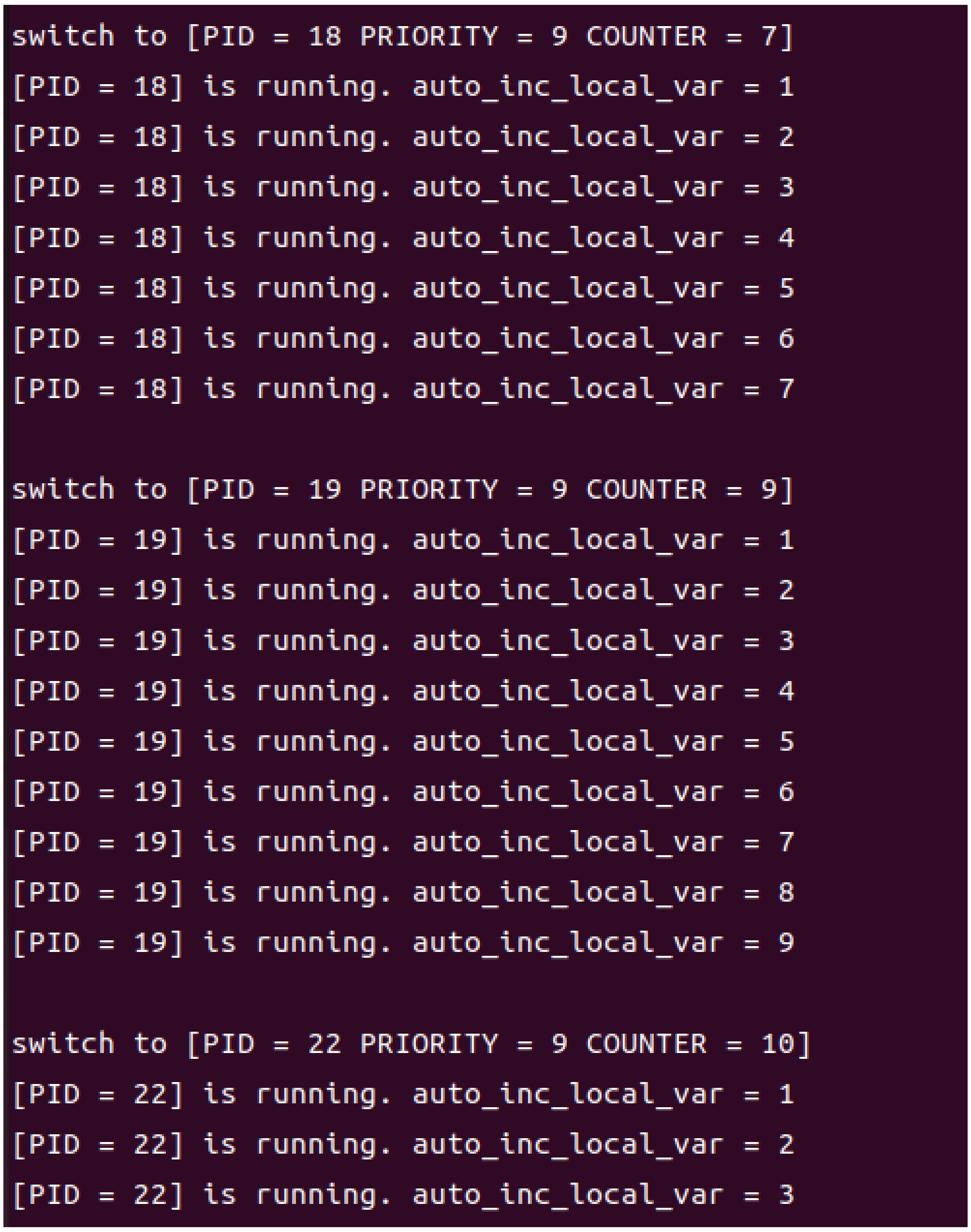
优先级调度



```
SET [PID = 1 PRIORITY = 2 COUNTER = 4]
SET [PID = 2 PRIORITY = 5 COUNTER = 4]
SET [PID = 3 PRIORITY = 1 COUNTER = 9]
SET [PID = 4 PRIORITY = 5 COUNTER = 7]
SET [PID = 5 PRIORITY = 1 COUNTER = 7]
SET [PID = 6 PRIORITY = 1 COUNTER = 2]
SET [PID = 7 PRIORITY = 6 COUNTER = 11]
SET [PID = 8 PRIORITY = 3 COUNTER = 2]
SET [PID = 9 PRIORITY = 10 COUNTER = 4]
SET [PID = 10 PRIORITY = 5 COUNTER = 8]
SET [PID = 11 PRIORITY = 5 COUNTER = 5]
SET [PID = 12 PRIORITY = 1 COUNTER = 3]
SET [PID = 13 PRIORITY = 6 COUNTER = 2]
SET [PID = 14 PRIORITY = 1 COUNTER = 4]
SET [PID = 15 PRIORITY = 5 COUNTER = 6]
SET [PID = 16 PRIORITY = 8 COUNTER = 2]
SET [PID = 17 PRIORITY = 6 COUNTER = 2]
SET [PID = 18 PRIORITY = 9 COUNTER = 7]
SET [PID = 19 PRIORITY = 9 COUNTER = 9]
SET [PID = 20 PRIORITY = 10 COUNTER = 10]
SET [PID = 21 PRIORITY = 7 COUNTER = 6]
SET [PID = 22 PRIORITY = 9 COUNTER = 10]
SET [PID = 23 PRIORITY = 1 COUNTER = 9]
SET [PID = 24 PRIORITY = 4 COUNTER = 8]
SET [PID = 25 PRIORITY = 9 COUNTER = 7]
SET [PID = 26 PRIORITY = 1 COUNTER = 8]
SET [PID = 27 PRIORITY = 2 COUNTER = 2]
SET [PID = 28 PRIORITY = 4 COUNTER = 3]
SET [PID = 29 PRIORITY = 9 COUNTER = 5]
SET [PID = 30 PRIORITY = 10 COUNTER = 10]
SET [PID = 31 PRIORITY = 5 COUNTER = 10]
```



```
switch to [PID = 9 PRIORITY = 10 COUNTER = 4]
[PID = 9] is running. auto_inc_local_var = 1
[PID = 9] is running. auto_inc_local_var = 2
[PID = 9] is running. auto_inc_local_var = 3
[PID = 9] is running. auto_inc_local_var = 4

switch to [PID = 20 PRIORITY = 10 COUNTER = 10]
[PID = 20] is running. auto_inc_local_var = 1
[PID = 20] is running. auto_inc_local_var = 2
[PID = 20] is running. auto_inc_local_var = 3
[PID = 20] is running. auto_inc_local_var = 4
[PID = 20] is running. auto_inc_local_var = 5
[PID = 20] is running. auto_inc_local_var = 6
[PID = 20] is running. auto_inc_local_var = 7
[PID = 20] is running. auto_inc_local_var = 8
[PID = 20] is running. auto_inc_local_var = 9
[PID = 20] is running. auto_inc_local_var = 10

switch to [PID = 30 PRIORITY = 10 COUNTER = 10]
[PID = 30] is running. auto_inc_local_var = 1
[PID = 30] is running. auto_inc_local_var = 2
[PID = 30] is running. auto_inc_local_var = 3
[PID = 30] is running. auto_inc_local_var = 4
[PID = 30] is running. auto_inc_local_var = 5
[PID = 30] is running. auto_inc_local_var = 6
[PID = 30] is running. auto_inc_local_var = 7
[PID = 30] is running. auto_inc_local_var = 8
[PID = 30] is running. auto_inc_local_var = 9
[PID = 30] is running. auto_inc_local_var = 10
```



```
switch to [PID = 18 PRIORITY = 9 COUNTER = 7]
[PID = 18] is running. auto_inc_local_var = 1
[PID = 18] is running. auto_inc_local_var = 2
[PID = 18] is running. auto_inc_local_var = 3
[PID = 18] is running. auto_inc_local_var = 4
[PID = 18] is running. auto_inc_local_var = 5
[PID = 18] is running. auto_inc_local_var = 6
[PID = 18] is running. auto_inc_local_var = 7

switch to [PID = 19 PRIORITY = 9 COUNTER = 9]
[PID = 19] is running. auto_inc_local_var = 1
[PID = 19] is running. auto_inc_local_var = 2
[PID = 19] is running. auto_inc_local_var = 3
[PID = 19] is running. auto_inc_local_var = 4
[PID = 19] is running. auto_inc_local_var = 5
[PID = 19] is running. auto_inc_local_var = 6
[PID = 19] is running. auto_inc_local_var = 7
[PID = 19] is running. auto_inc_local_var = 8
[PID = 19] is running. auto_inc_local_var = 9

switch to [PID = 22 PRIORITY = 9 COUNTER = 10]
[PID = 22] is running. auto_inc_local_var = 1
[PID = 22] is running. auto_inc_local_var = 2
[PID = 22] is running. auto_inc_local_var = 3
```

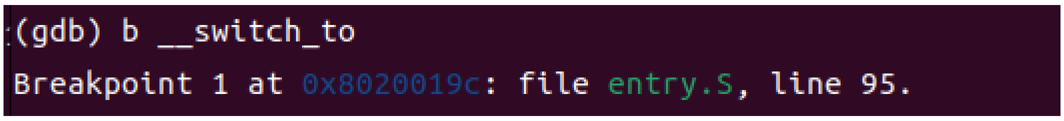
思考题

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

   s0-s11,ra,sp是callee saved register，需要自行保存。其他寄存器属于caller saved register，在c语言调用中会被保存在栈中。

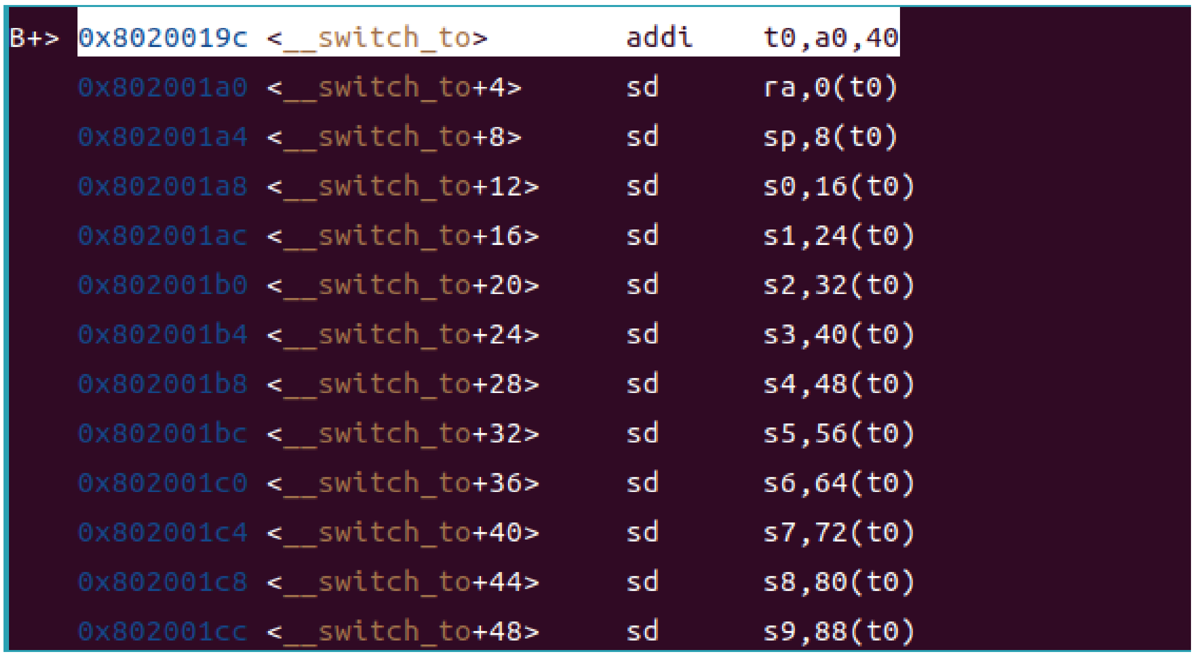
2. 当线程第一次调用时, 其 `ra` 所代表的返回点是 `__dummy`。那么在之后的线程调用中 `context_switch` 中, `ra` 保存/恢复的函数返回点是什么呢? 请同学用 gdb 尝试追踪一次完整的线程切换流程, 并关注每一次 `ra` 的变换 (需要截图)。

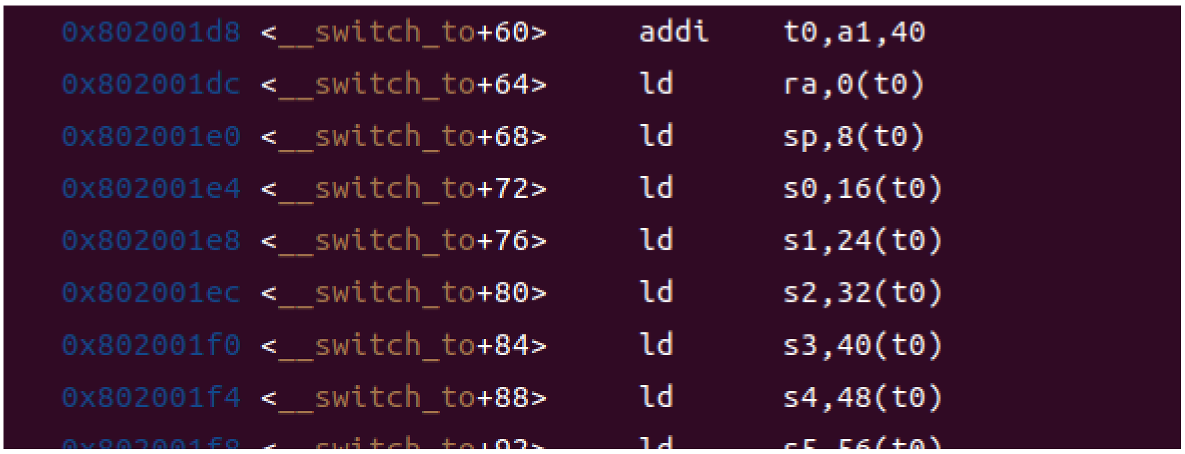
   首先在__switch_to处打断点,



```
(gdb) b __switch_to
Breakpoint 1 at 0x8020019c: file entry.S, line 95.
```

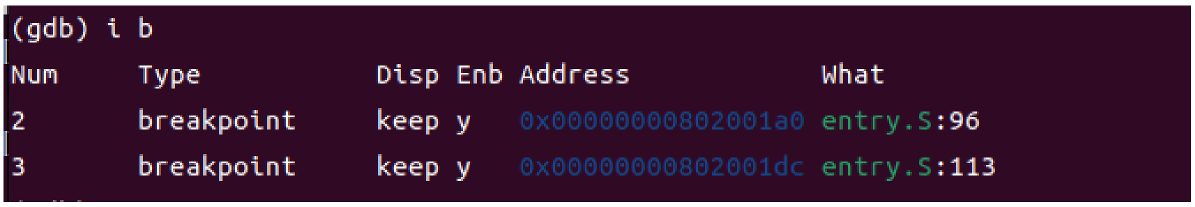
   运行到断点后查看汇编代码


```
B+> 0x8020019c <__switch_to>         addi    t0,a0,40
    0x802001a0 <__switch_to+4>       sd      ra,0(t0)
    0x802001a4 <__switch_to+8>       sd      sp,8(t0)
    0x802001a8 <__switch_to+12>      sd      s0,16(t0)
    0x802001ac <__switch_to+16>      sd      s1,24(t0)
    0x802001b0 <__switch_to+20>      sd      s2,32(t0)
    0x802001b4 <__switch_to+24>      sd      s3,40(t0)
    0x802001b8 <__switch_to+28>      sd      s4,48(t0)
    0x802001bc <__switch_to+32>      sd      s5,56(t0)
    0x802001c0 <__switch_to+36>      sd      s6,64(t0)
    0x802001c4 <__switch_to+40>      sd      s7,72(t0)
    0x802001c8 <__switch_to+44>      sd      s8,80(t0)
    0x802001cc <__switch_to+48>      sd      s9,88(t0)
```

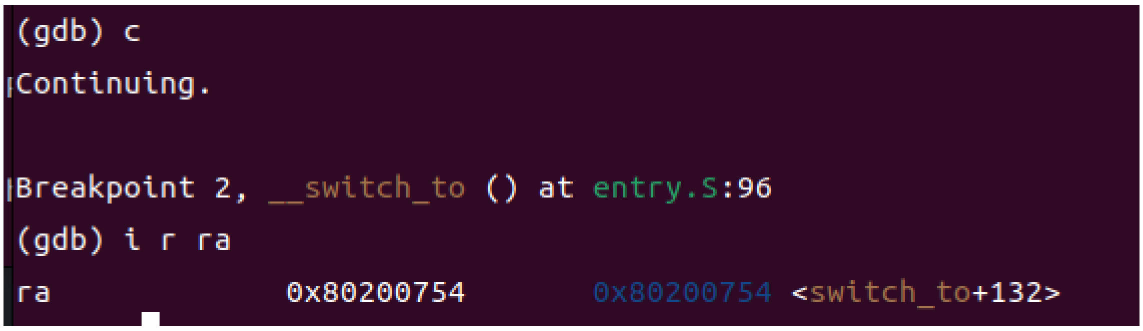

```
    0x802001d8 <__switch_to+60>      addi    t0,a1,40
    0x802001dc <__switch_to+64>      ld      ra,0(t0)
    0x802001e0 <__switch_to+68>      ld      sp,8(t0)
    0x802001e4 <__switch_to+72>      ld      s0,16(t0)
    0x802001e8 <__switch_to+76>      ld      s1,24(t0)
    0x802001ec <__switch_to+80>      ld      s2,32(t0)
    0x802001f0 <__switch_to+84>      ld      s3,40(t0)
    0x802001f4 <__switch_to+88>      ld      s4,48(t0)
    0x802001f8 <__switch_to+92>      ld      s5,56(t0)
```

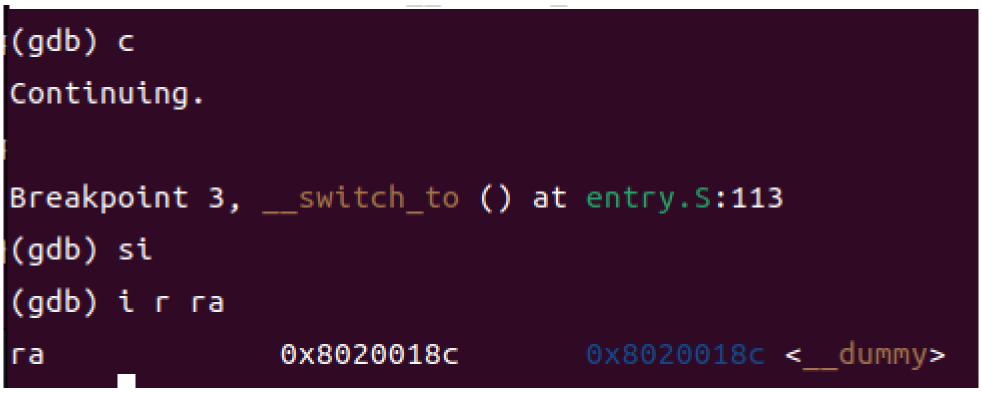
在ra的恢复和保存处打断点，即0x802001a0和0x802001dc

清除__switch_to处的断点


```
(gdb) i b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000802001a0 entry.S:96
3       breakpoint     keep y   0x00000000802001dc entry.S:113
```

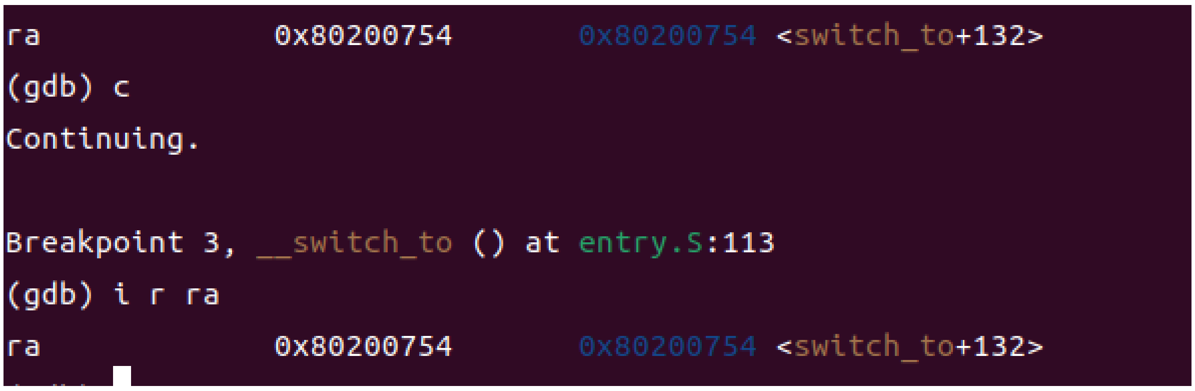
第一次保存的返回点是switch_to+132,第一次恢复的返回点是__dummy


```
(gdb) c
Continuing.

Breakpoint 2, __switch_to () at entry.S:96
(gdb) i r ra
ra             0x80200754          0x80200754 <switch_to+132>
```



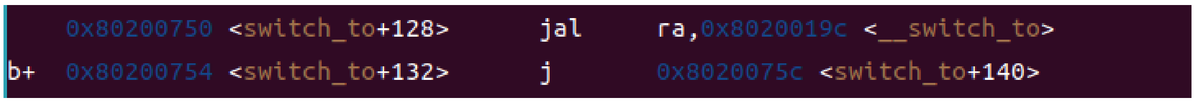
```
(gdb) c
Continuing.

Breakpoint 3, __switch_to () at entry.S:113
(gdb) si
(gdb) i r ra
ra             0x8020018c       0x8020018c <__dummy>
```

之后每次保存和恢复的返回点都是switch_to+132



```
ra             0x80200754       0x80200754 <switch_to+132>
(gdb) c
Continuing.

Breakpoint 3, __switch_to () at entry.S:113
(gdb) i r ra
ra             0x80200754       0x80200754 <switch_to+132>
```

可以看到switch_to+132为__switch_to的下一行



```
    0x80200750 <switch_to+128>      jal     ra,0x8020019c <__switch_to>
b+  0x80200754 <switch_to+132>      j       0x8020075c <switch_to+140>
```