Lab 3: RV64 内核线程调度

学号: 3190105838

姓名: 范钊瑀

4.1 准备工程

首先我们将Lab2中用于终端处理，系统初始化等部分的代码复用到本次实验中，并在定义头文件 `defs.h` 中加入有关的宏定义。相关的宏定义中包含

```
#define PHY_START 0x0000000080000000
#define PHY_SIZE  128 * 1024 * 1024 // 128MB,  QEMU 默认内存大小
#define PHY_END   (PHY_START + PHY_SIZE)

#define PGSIZE 0x1000 // 4KB
#define PGROUNDUP(addr) ((addr + PGSIZE - 1) & (~(PGSIZE - 1)))
#define PGROUNDDOWN(addr) (addr & (~(PGSIZE - 1)))
```

4.2 proc.h 数据结构定义

添加 `proc.h` 文件用于存储基本数据类型的定义， `thread_struct` 结构体用于存储保存下来的ra、sp、s0、s1等寄存器的值， `task_struct` 用于保存线程的状态、运行时间等数据。

4.3 线程调度功能实现

4.3.1 线程初始化

在这里我们需要修改 `proc.c` 文件的 `task_init()` 函数以初始化各个线程，首先我们需要设置os最初的线程的 `task_struct` ，它本身是一个 `idle` 线程，在初始化的时候需要我们将 `idle` 指针指向它，同时由于它随着os的启动而运行，我们也要将 `current` 指针指向它。
利用 `kalloc()` 函数可以分配一个 `task_struct` 的连续地址，我们将这个地址作为 `idle`

线程的 `task_struct` 的首地址，因此将 `idle` 指针指向它。同理我们也需要对其状态 `state`， `counter`， `priority`， `time_slice`， `pid`，进行初始化，其中 `state` 为 `TASK_RUNNABLE`，代表线程在运行，可被调度， `counter` 为0，因为我们想要立即开始其它线程的调度， `priority` 和 `pid` 也可设为0。

```
idle = (struct task_struct *)kalloc();
idle->state = TASK_RUNNING;
idle->counter = 0;
idle->priority = 0;
idle->pid = 0;
current = idle;
task[0] = idle;
```

类似的，我们也应当对其他线程的 `task_struct` 进行初始化操作，在这里的循环中，每个 `task` 线程数组里的个体都被分配了一块地址空间，并且将其状态 `state`， `counter`， `priority`， `pid` 进行类似的初始化操作。此外我们也需要设置每个线程的 `ra` 和 `sp` 位， `ra` 位被设置成函数 `__dummy` 的首地址， `__dummy` 函数是用于线程第一次调度的一个特殊函数，它设置了 `sepc` 的值，并从中断中返回到 `dummy()` 函数的位置，由于创建完的一个新线程在第一次调度时是没有上下文需要恢复的，因此在第一次调度的时候我们需要将线程的返回地址“引导”到我们的循环 `dummy()` 函数中，做这件事的就是 `__dummy` 函数。而 `sp` 被设置为该线程申请的物理页的高地址。在这里需要注意的是，由于 `task[i]` 的首地址是分配到的低位地址，我们如果想要得到 `sp` 指针的地址需要加上 `PGSIZE` 的偏移量，此外，我们最好将 `task[i]` 和 `PGSIZE` 统一转化成 `uint64` 类型之后再求和，否则可能会产生加上了 `PGSIZE` 倍数的偏移量的后果。

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

}
```

4.3.2 __dummy 与 dummy 介绍

在进程正常运行时，它们会运行到 `dummy()` 函数的代码段，如下所示， `dummy()` 函数主要负责了进程正常运行时的输出，此函数中的 `auto_inc_local_var` 变量在每次counter变化后会自增，而counter在每次timer执行时会减小，减小后会立即在 `dummy()` 中输出，因此做到了每秒输出一次的效果。而它也会在 `context_switch` 的时候被保存下来，同一个线程下次再被调度执行的时候会被恢复。

```
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);
        }
    }
}
```

而 `__dummy` 存在于 `entry.S` 中，正如前文所述，它用于在新创建线程的第一次调度的时候给线程返回到正确的位置，会存于每个线程的 `ra` 寄存器中，如下，它所做的就是将 `dummy()` 函数的地址加载到 `sepc` 中并调用 `sret` 返回。

```
__dummy:
    la x10, dummy
    csrrw x0, sepc, x10 # restore sepc
    sret
```

4.3.3 实现线程切换

为了实现进程之间的切换，我们需要用实现 `switch_to` 和 `__switch_to` 两个函数，下面分别介绍它们。

`switch_to` 函数会在调度 `schedule()` 中被调用，它完成了调用汇编中的 `__switch_to` 并完成进程切换的操作

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

```
void switch_to(struct task_struct* next) {
    printk("switch from [PID = %d COUNTER = %d] to [PID = %d COUNTER =
%d]\n", current->pid, current->counter, next->pid, next->counter);
    if(current->pid != next->pid) {
        struct task_struct * _tmp = current;
        current = next;
        __switch_to(_tmp, next);
    }
    else{
        // do nothing
    }
}
```

下面是线程状态段数据结构 `thread_struct` 和线程数据结构 `task_struct` 的类型定义。

```
/* 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;
};
```

如下所示， `__switch_to` 中完成了前序线程的寄存器保存和后一进程的寄存器载入，由于 `task_struct` 结构体中， `thread_info` 是 `struct thread_info*` 指针类型的， `state` ， `counter` ， `priority` ， `pid` 都是 `uint64` 类型的，这些类型在我们的系统中都是8Byte大小的，因此一共占用40Byte大小，又因为不难发现 `task_struct` 结构体是存放在 `PGSIZE` 中最低地址的部分，也即代表 `thread` 结构体是从 `task[i]` 中存储的首地址偏移40Byte得到的，又由于 `thread_struct` 结构体中是按 `ra` ， `sp` ， `s0` 等寄存器的值依次排放的，在存储地址时我们应该做到对应的顺序关系。

在恢复完上下文后，我们需要通过 `ret` 指令来返回到 `ra` 寄存器中存储的地址中，以完成线程的切换。
需要注意的是， `x10(a0)`，`x11(a1)` 寄存器被用于参数传递，存储了两个线程数据结构的首地址。

```
__switch_to:
    # x10: prev task_struct
    # save state to prev process
    # YOUR CODE HERE
    sd ra, 40(x10)
    sd sp, 48(x10)
    sd s0, 56(x10)
    sd s1, 64(x10)
    sd s2, 72(x10)
    sd s3, 80(x10)
    sd s4, 88(x10)
    sd s5, 96(x10)
    sd s6, 104(x10)
    sd s7, 112(x10)
    sd s8, 120(x10)
    sd s9, 128(x10)
    sd s10, 136(x10)
    sd s11, 144(x10)

    # x11: next task_struct
    # restore state from next process
    # YOUR CODE HERE
    ld ra, 40(x11)
    ld sp, 48(x11)
    ld s0, 56(x11)
    ld s1, 64(x11)
    ld s2, 72(x11)
    ld s3, 80(x11)
    ld s4, 88(x11)
    ld s5, 96(x11)
    ld s6, 104(x11)
    ld s7, 112(x11)
    ld s8, 120(x11)
    ld s9, 128(x11)
    ld s10, 136(x11)
    ld s11, 144(x11)

    ret
```

4.3.4 实现调度入口函数

`do_timer`函数用于进程的计时器，它会在每次时钟中断时被调用，每次调用时将能够运行的程序剩余时间减一，如果当前线程是idle线程（如最初情况）则直接进行调度，否则需要将当前线程的运行剩余时间减1，如果剩余时间为0，则进行调度，否则表明程序还在运行。调度即为调用`schedule()`函数。

```
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 */

    /* YOUR CODE HERE */
    if(current->pid == idle->pid){
        printk("current == idle\n");
        schedule();
    }
    else{
        (current->counter)--;
        if(current->counter > 0){
            return;
        }
        else{
            schedule();
        }

    }
}
```

4.3.5 实现线程调度

4.3.5.1 短作业优先调度算法

在SJF(短作业优先调度算法)中，我们需要找到运行时间最短的线程，并将其设置为current线程。其中该线程需要满足应当处于`TASK_RUNNING`状态的条件，遍历得到最短运行时间的进程后，若发现所有进程的剩余时间均为0，也即一轮程序已经运行完毕，此时我们重新对所有线程的剩余时间赋值并重新调度，以开启下一轮调度。
在找到需要被调度的目标程序后，我们只需通过调用`switch_to()`函数以切换到目标进程。

```
int find_min_time(){
    int _min_time = -1;
    int _min_id = -1;
    for(int i = NR_TASKS-1; i >= 1 ; i--){
        // if the time is not positive, we just pass it by
        if(task[i]->counter <= 0){
            continue;
        }
        if(task[i]->counter < _min_time){
            _min_time = task[i]->counter;
            _min_id = i;
        }
    }
    return _min_id;
}

// Implement SJF
#ifdef SJF
void schedule(void) {
    /* YOUR CODE HERE */
    int all_zeros = 1;
    int min_index = find_min_time();
    int min_time = task[min_index]->counter;
    for(int i = 1; i < NR_TASKS; i++){
        if(task[i]->state == TASK_RUNNING){
            if(all_zeros && task[i]->counter > 0){
                all_zeros = 0;
            }
            if(task[i]->counter < min_time && task[i]->counter > 0){
                min_time = task[i]->counter;
                min_index = i;
            }

        }
    }
    if(all_zeros){
        for(int i = 1; i < NR_TASKS; i++){
            if(task[i]->state == TASK_RUNNING){
                task[i]->counter = rand();
                printk("SET [PID = %d COUNTER = %d]\n", task[i]->pid,
task[i]->counter);

            }
```

```
        }
        // schedule();
        min_index = find_min_time();
        min_time = task[min_index]->counter;
        for(int i = 1; i < NR_TASKS; i++){
            if(task[i]->state == TASK_RUNNING){
                if(task[i]->counter < min_time){
                    min_time = task[i]->counter;
                    min_index = i;
                }

            }
        }
    }
    // schedule ith process
    switch_to(task[min_index]);
}
#endif
```

4.3.5.2 优先级调度算法

在这里需要注意的是，优先级赋值的时候需要参考Linux V0.11中的代码实现，否则会产生错误的结果。

```
#ifdef PRIORITY
void schedule(void){
    /* YOUR CODE HERE */
    int all_zeros = 1;
    int max_index = -1;
    int max_priority = 0;
    for(int i = NR_TASKS - 1; i >= 1; i--){
        if(task[i]->state == TASK_RUNNING){
            if(all_zeros && task[i]->counter > 0){
                all_zeros = 0;
            }
            if(task[i]->priority > max_priority && task[i]->counter > 0){
                max_priority = task[i]->priority;
                max_index = i;
            }

        }
    }
    if(all_zeros){
```

```
        for(int i = 1; i < NR_TASKS; i++){
            if(task[i]->state == TASK_RUNNING){
                task[i]->counter = (task[i]->counter >> 1) + task[i]-
>priority;
                printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n",
task[i]->pid, task[i]->priority, task[i]->counter);

            }

        }
        // schedule();
        max_index = -1;
        max_priority = 0;
        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;
                    max_index = i;
                }

            }
        }
    }
    // schedule ith process
    switch_to(task[max_index]);
}
#endif
```

4.4 编译及测试

4.4.1 SJF算法结果

- 4个task时

  第一次循环与运行：

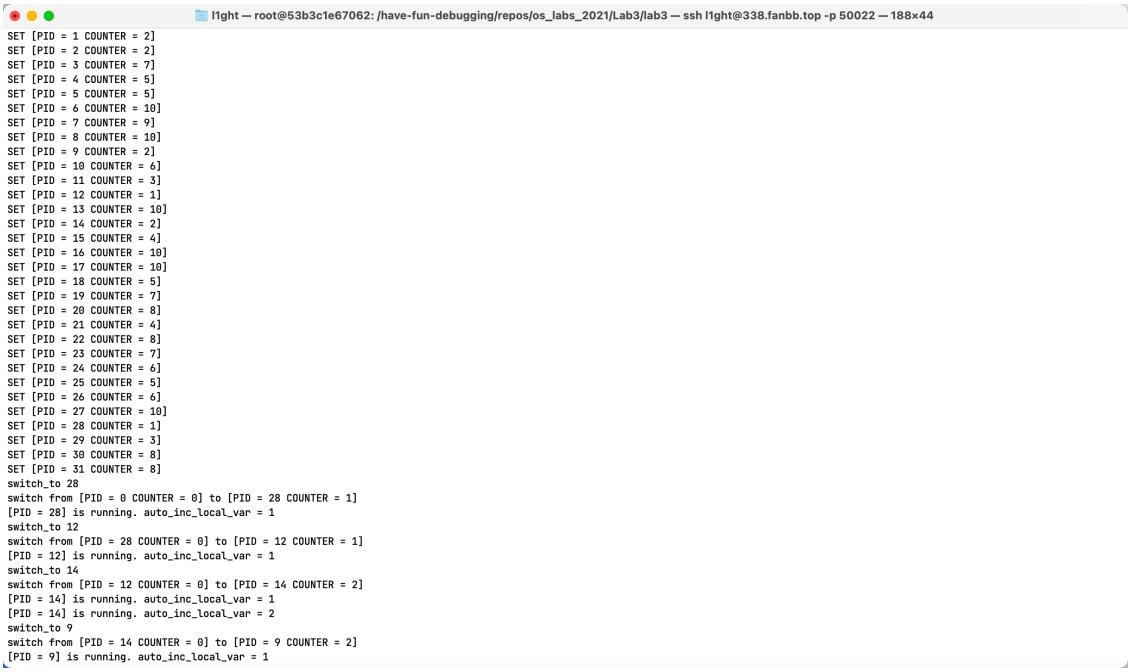
```
l1ght — root@53b3c1e67062: /have-fun-debugging/repos/os_labs_2021/Lab3/lab3 — ssh l1ght@338.fanbb.top -p 50022 — 143×32
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 5]
SET [PID = 4 COUNTER = 2]
switch_to 4
switch from [PID = 0 COUNTER = 0] to [PID = 4 COUNTER = 2]
[PID = 4] is running. auto_inc_local_var = 1
[PID = 4] is running. auto_inc_local_var = 2
switch_to 3
switch from [PID = 4 COUNTER = 0] to [PID = 3 COUNTER = 5]
[PID = 3] is running. auto_inc_local_var = 1
[PID = 3] is running. auto_inc_local_var = 2
[PID = 3] is running. auto_inc_local_var = 3
[PID = 3] is running. auto_inc_local_var = 4
[PID = 3] is running. auto_inc_local_var = 5
switch_to 2
switch from [PID = 3 COUNTER = 0] to [PID = 2 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 1
[PID = 2] is running. auto_inc_local_var = 2
[PID = 2] is running. auto_inc_local_var = 3
[PID = 2] is running. auto_inc_local_var = 4
[PID = 2] is running. auto_inc_local_var = 5
[PID = 2] is running. auto_inc_local_var = 6
[PID = 2] is running. auto_inc_local_var = 7
[PID = 2] is running. auto_inc_local_var = 8
[PID = 2] is running. auto_inc_local_var = 9
[PID = 2] is running. auto_inc_local_var = 10
switch_to 1
switch from [PID = 2 COUNTER = 0] to [PID = 1 COUNTER = 10]
[PID = 1] is running. auto_inc_local_var = 1
[PID = 1] is running. auto_inc_local_var = 2
[PID = 1] is running. auto_inc_local_var = 3
```

- 第一轮结束后，重新对counter赋值并运行，进入第二轮：

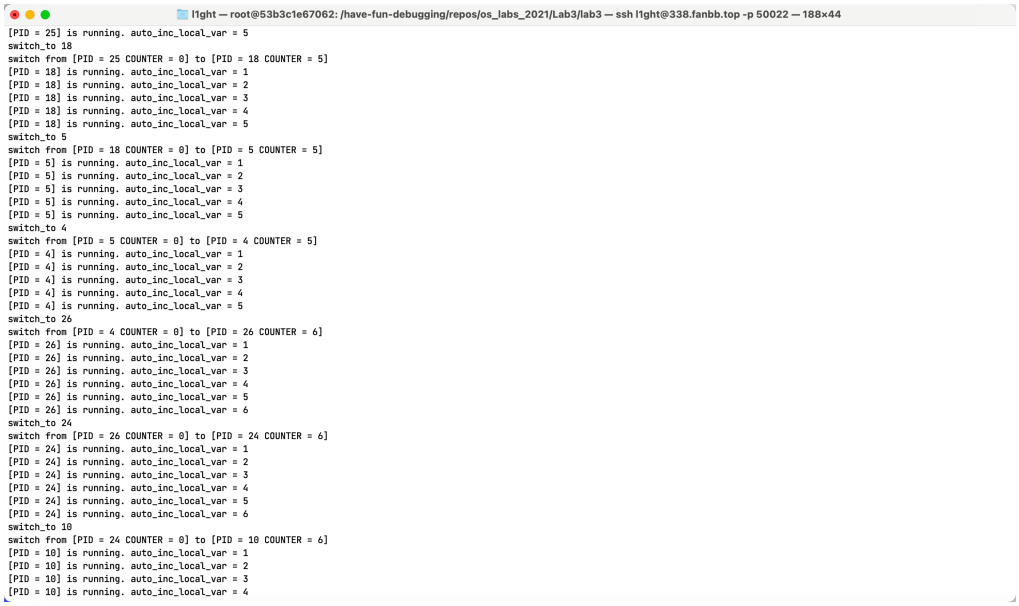
```
l1ght — root@53b3c1e67062: /have-fun-debugging/repos/os_labs_2021/Lab3/lab3 — ssh l1ght@338.fanbb.top -p 50022 — 143×32
[PID = 1] is running. auto_inc_local_var = 3
[PID = 1] is running. auto_inc_local_var = 4
[PID = 1] is running. auto_inc_local_var = 5
[PID = 1] is running. auto_inc_local_var = 6
[PID = 1] is running. auto_inc_local_var = 7
[PID = 1] is running. auto_inc_local_var = 8
[PID = 1] is running. auto_inc_local_var = 9
[PID = 1] is running. auto_inc_local_var = 10
SET [PID = 1 COUNTER = 9]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 4]
SET [PID = 4 COUNTER = 10]
switch_to 3
switch from [PID = 1 COUNTER = 9] to [PID = 3 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 6
[PID = 3] is running. auto_inc_local_var = 7
[PID = 3] is running. auto_inc_local_var = 8
[PID = 3] is running. auto_inc_local_var = 9
switch_to 2
switch from [PID = 3 COUNTER = 0] to [PID = 2 COUNTER = 4]
[PID = 2] is running. auto_inc_local_var = 11
[PID = 2] is running. auto_inc_local_var = 12
[PID = 2] is running. auto_inc_local_var = 13
[PID = 2] is running. auto_inc_local_var = 14
switch_to 1
switch from [PID = 2 COUNTER = 0] to [PID = 1 COUNTER = 9]
[PID = 1] is running. auto_inc_local_var = 11
[PID = 1] is running. auto_inc_local_var = 12
[PID = 1] is running. auto_inc_local_var = 13
[PID = 1] is running. auto_inc_local_var = 14
[PID = 1] is running. auto_inc_local_var = 15
[PID = 1] is running. auto_inc_local_var = 16
```

- 32个task时

  初始化counter：

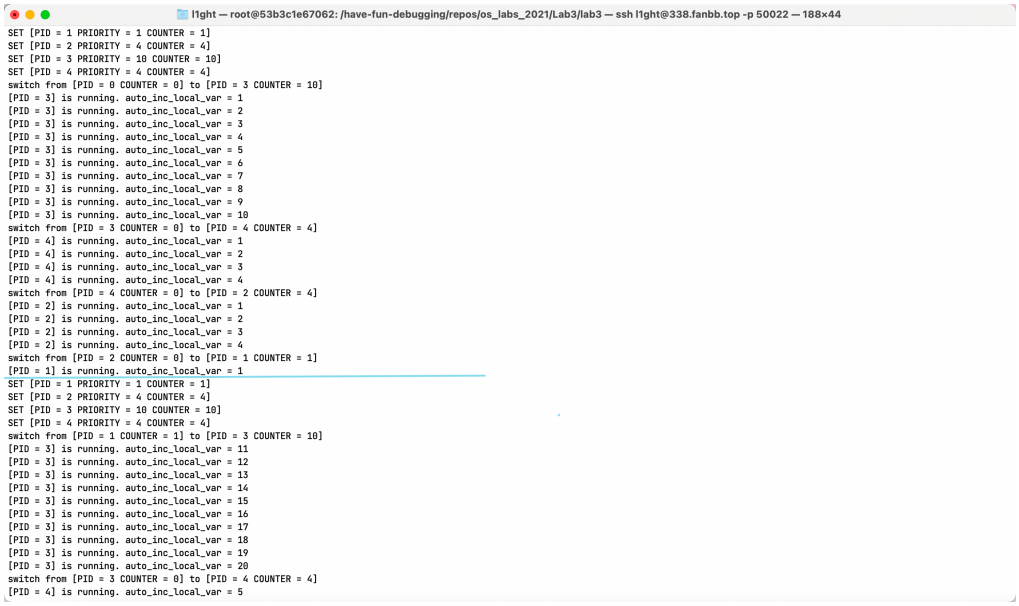

l1ght — root@53b3c1e67062: /have-fun-debugging/repos/os_labs_2021/Lab3/lab3 — ssh l1ght@338.fanbb.top -p 50022 — 188×44
SET [PID = 1 COUNTER = 2]
SET [PID = 2 COUNTER = 2]
SET [PID = 3 COUNTER = 7]
SET [PID = 4 COUNTER = 5]
SET [PID = 5 COUNTER = 5]
SET [PID = 6 COUNTER = 10]
SET [PID = 7 COUNTER = 9]
SET [PID = 8 COUNTER = 10]
SET [PID = 9 COUNTER = 2]
SET [PID = 10 COUNTER = 6]
SET [PID = 11 COUNTER = 3]
SET [PID = 12 COUNTER = 1]
SET [PID = 13 COUNTER = 10]
SET [PID = 14 COUNTER = 2]
SET [PID = 15 COUNTER = 4]
SET [PID = 16 COUNTER = 10]
SET [PID = 17 COUNTER = 10]
SET [PID = 18 COUNTER = 5]
SET [PID = 19 COUNTER = 7]
SET [PID = 20 COUNTER = 8]
SET [PID = 21 COUNTER = 4]
SET [PID = 22 COUNTER = 8]
SET [PID = 23 COUNTER = 7]
SET [PID = 24 COUNTER = 6]
SET [PID = 25 COUNTER = 5]
SET [PID = 26 COUNTER = 6]
SET [PID = 27 COUNTER = 10]
SET [PID = 28 COUNTER = 1]
SET [PID = 29 COUNTER = 3]
SET [PID = 30 COUNTER = 8]
SET [PID = 31 COUNTER = 8]
switch_to 28
switch from [PID = 0 COUNTER = 0] to [PID = 28 COUNTER = 1]
[PID = 28] is running. auto_inc_local_var = 1
switch_to 12
switch from [PID = 28 COUNTER = 0] to [PID = 12 COUNTER = 1]
[PID = 12] is running. auto_inc_local_var = 1
switch_to 14
switch from [PID = 12 COUNTER = 0] to [PID = 14 COUNTER = 2]
[PID = 14] is running. auto_inc_local_var = 1
[PID = 14] is running. auto_inc_local_var = 2
switch_to 9
switch from [PID = 14 COUNTER = 0] to [PID = 9 COUNTER = 2]
[PID = 9] is running. auto_inc_local_var = 1

正常线程切换时截图：

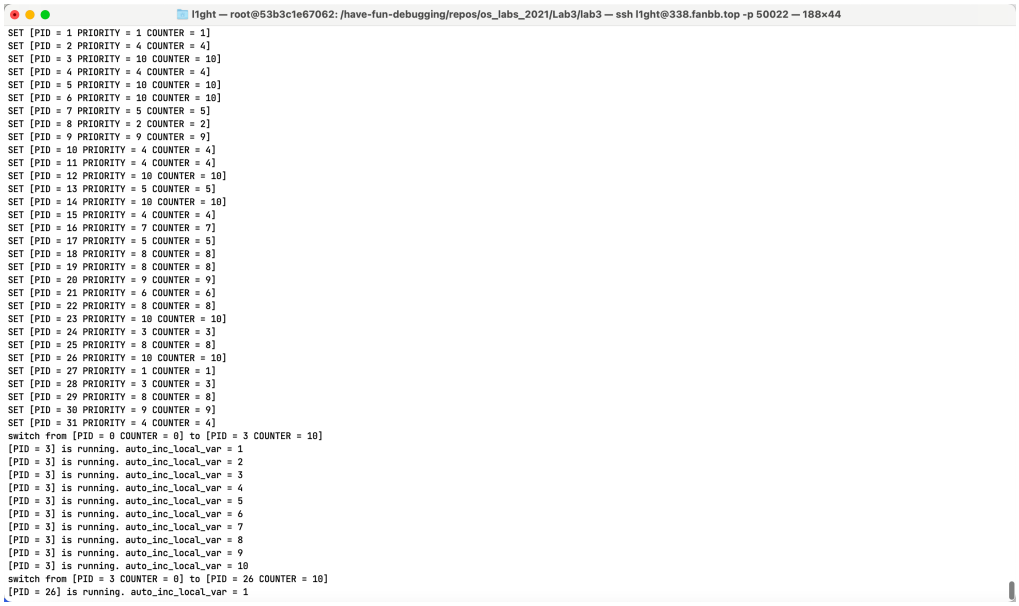

l1ght — root@53b3c1e67062: /have-fun-debugging/repos/os_labs_2021/Lab3/lab3 — ssh l1ght@338.fanbb.top -p 50022 — 188×44
[PID = 25] is running. auto_inc_local_var = 5
switch_to 18
switch from [PID = 25 COUNTER = 0] to [PID = 18 COUNTER = 5]
[PID = 18] is running. auto_inc_local_var = 1
[PID = 18] is running. 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
switch_to 5
switch from [PID = 18 COUNTER = 0] to [PID = 5 COUNTER = 5]
[PID = 5] is running. auto_inc_local_var = 1
[PID = 5] is running. auto_inc_local_var = 2
[PID = 5] is running. auto_inc_local_var = 3
[PID = 5] is running. auto_inc_local_var = 4
[PID = 5] is running. auto_inc_local_var = 5
switch_to 4
switch from [PID = 5 COUNTER = 0] to [PID = 4 COUNTER = 5]
[PID = 4] is running. auto_inc_local_var = 1
[PID = 4] is running. auto_inc_local_var = 2
[PID = 4] is running. auto_inc_local_var = 3
[PID = 4] is running. auto_inc_local_var = 4
[PID = 4] is running. auto_inc_local_var = 5
switch_to 26
switch from [PID = 4 COUNTER = 0] to [PID = 26 COUNTER = 6]
[PID = 26] is running. auto_inc_local_var = 1
[PID = 26] is running. auto_inc_local_var = 2
[PID = 26] is running. auto_inc_local_var = 3
[PID = 26] is running. auto_inc_local_var = 4
[PID = 26] is running. auto_inc_local_var = 5
[PID = 26] is running. auto_inc_local_var = 6
switch_to 24
switch from [PID = 26 COUNTER = 0] to [PID = 24 COUNTER = 6]
[PID = 24] is running. auto_inc_local_var = 1
[PID = 24] is running. auto_inc_local_var = 2
[PID = 24] is running. auto_inc_local_var = 3
[PID = 24] is running. auto_inc_local_var = 4
[PID = 24] is running. auto_inc_local_var = 5
[PID = 24] is running. auto_inc_local_var = 6
switch_to 10
switch from [PID = 24 COUNTER = 0] to [PID = 10 COUNTER = 6]
[PID = 10] is running. auto_inc_local_var = 1
[PID = 10] is running. auto_inc_local_var = 2
[PID = 10] is running. auto_inc_local_var = 3
[PID = 10] is running. auto_inc_local_var = 4

4.4.2 PRIORITY算法结果

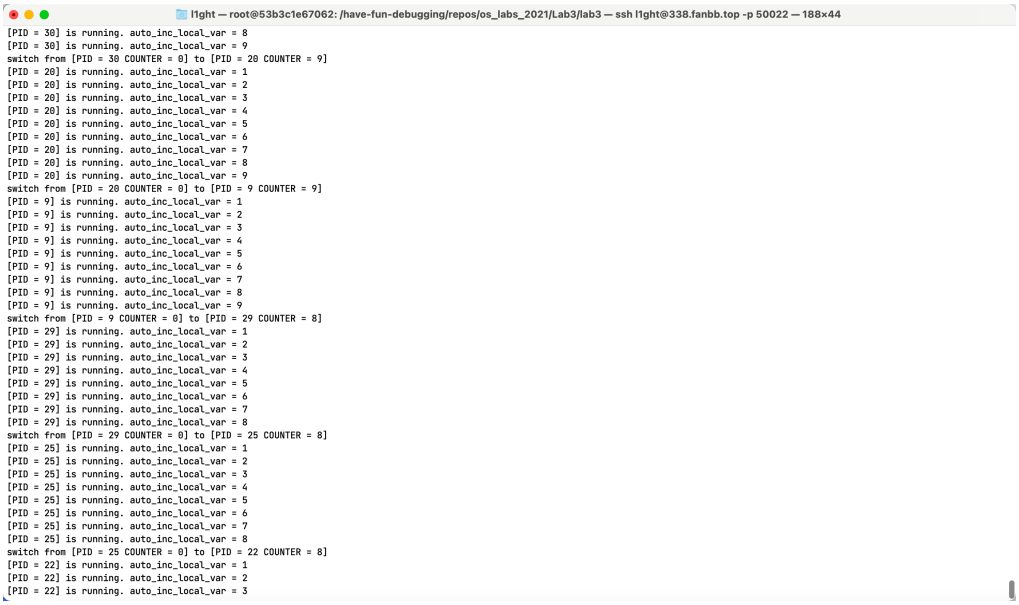
- 4个task时
  根据优先级，下图中蓝线上半部分为第一轮运行结果，下半部分为第二轮的结果，可以发现由于参考了Linux v0.11的调度实现，我们的输出是正常的



- 32个task时
  初始化counter：



正常的线程切换：



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？
ra, sp寄存器存储了程序的返回地址和栈帧地址，而s0到s11寄存器是Saved Registers, 其他的寄存器都是用来存储程序的参数和局部变量的。这些寄存器除了ra以外，在RISC-V都被约定为是Callee-saved，这些寄存器在线程中通常会被编译器优先调用，例如存储线程内部所用到的变量的值，也就是说在context_switch过后需要优先将该线程独占有的变量存储情况恢复。这一点我们可以通过 `objdump` 来反汇编得到，如下为 `dummy()` 函数的反汇编结果，不难发现编译器在函数内部先将Saved Registers保存下来，此后的局部变量都是利用Saved Registers来保存：

```
0000000080200050c <dummy>:
    8020050c: fc010113            addi  sp,sp,-64
    80200510: 01513423            sd  s5,8(sp)
    80200514: 3b9adab7            lui s5,0x3b9ad
    80200518: 02813823            sd  s0,48(sp)
    8020051c: 02913423            sd  s1,40(sp)
    80200520: 03213023            sd  s2,32(sp)
    80200524: 01313c23            sd  s3,24(sp)
    80200528: 01413823            sd  s4,16(sp)
    8020052c: 02113c23            sd  ra,56(sp)
    80200530: fff00413            li  s0,-1
    80200534: 00000493            li  s1,0
    80200538: 00006917            auipc s2,0x6
    8020053c: a9890913            addi  s2,s2,-1384 # 80205fd0 <current>
    80200540: fff00a13            li  s4,-1
```

```
    80200544: a07a8a93            addi  s5,s5,-1529 # 3b9aca07 <_skernel-
0x448535f9>
    80200548: 00002997            auipc s3,0x2
    8020054c: b2098993            addi  s3,s3,-1248 # 80202068
<_srodata+0x68>
    80200550: 00148793            addi  a5,s1,1
    80200554: 00098513            mv  a0,s3
    80200558: 01440863            beq s0,s4,80200568 <dummy+0x5c>
    8020055c: 00093703            ld  a4,0(s2)
    80200560: 01073703            ld  a4,16(a4) # ffffffffffff010
<t+0xffffffff7fdf902c>
    80200564: fe8706e3            beq a4,s0,80200550 <dummy+0x44>
    80200568: 0357f4b3            remu  s1,a5,s5
    8020056c: 00093783            ld  a5,0(s2)
    80200570: 0207b583            ld  a1,32(a5)
    80200574: 0107a403            lw  s0,16(a5)
    80200578: 00048613            mv  a2,s1
    8020057c: 6f0000ef            jal ra,80200c6c <printk>
    80200580: 00148793            addi  a5,s1,1
    80200584: 00098513            mv  a0,s3
    80200588: fd441ae3            bne s0,s4,8020055c <dummy+0x50>
    8020058c: fddff06f            j 80200568 <dummy+0x5c>
```

其余的Caller-saved寄存器都已经被上层的父进程保存过。而由于我们需要知道从context_switch中返回另一个进程的地址，因此我们同样也需要将ra寄存器的值保存下来。所以一共需要保存14个寄存器的值。

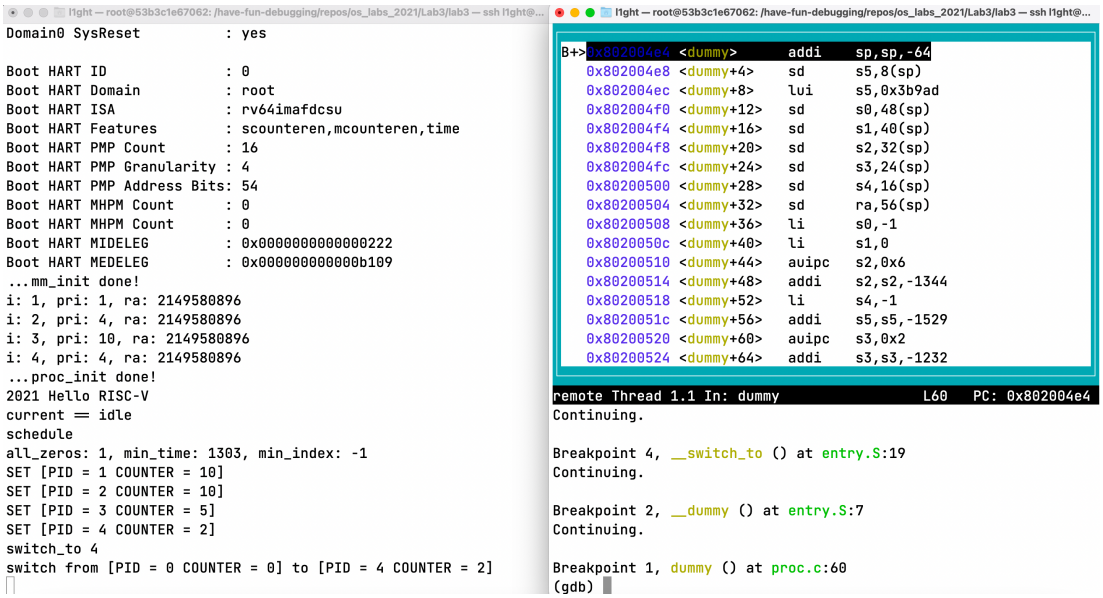
2.当线程第一次调用时， 其 ra 所代表的返回点是 __dummy。 那么在之后的线程调用中 context_switch 中，ra 保存/恢复的函数返回点是什么呢？ 请同学用gdb尝试追踪一次完整的线程切换流程， 并关注每一次 ra 的变换。

在之后的函数调用中，`ra`寄存器理应保存了真实的上下文环境，也即函数的返回地址，由于之前进程在`dummy()`函数的死循环中无限循环，返回之后应当重新回到`dummy()`函数中的`while(1)`处。

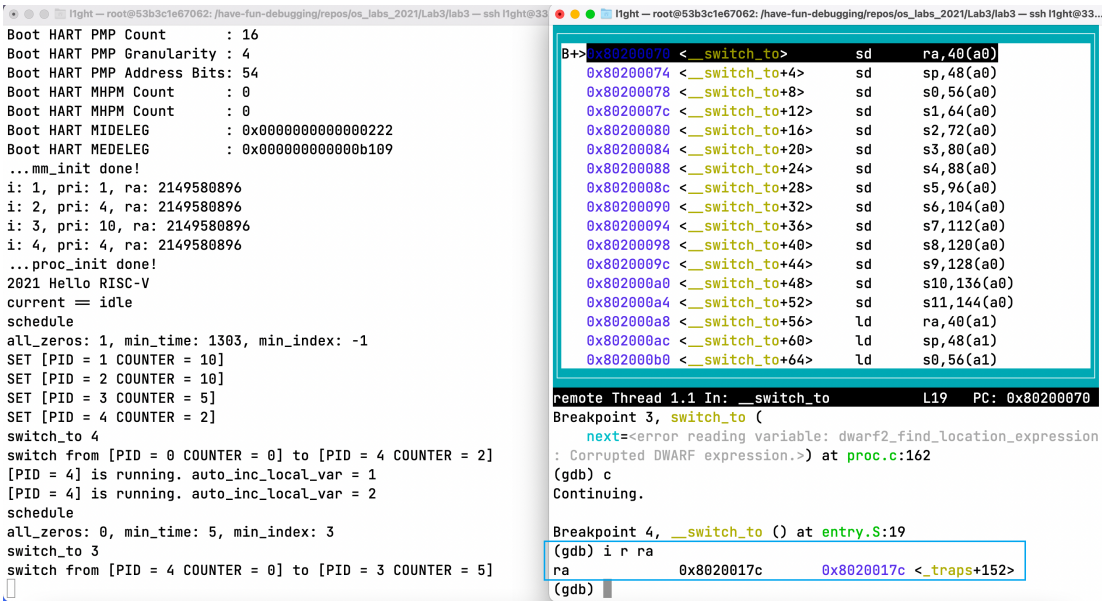
线程切换流程：
我们以`SJF`算法中，4个线程时，第一次由PID=4的线程切换到PID=3的线程的过程为例，计时程序在中断时触发，进入do_timer程序，继而根据情况判断是否需要执行调度，调度后则需要重新选择下一个运行的程序，并执行context_switch。由于是PID=3的线程的第一次调用，因此会返回到`__dummy`的地址
下面是gdb中用到的指令（已忽略单步执行和继续运行）

```
target remote :1234
layout asm
b dummy
b __dummy
b switch_to
b __switch_to
```

此时程序还在执行PID=4的线程中的 `dummy` 循环，等待着下一次时钟中断的触发



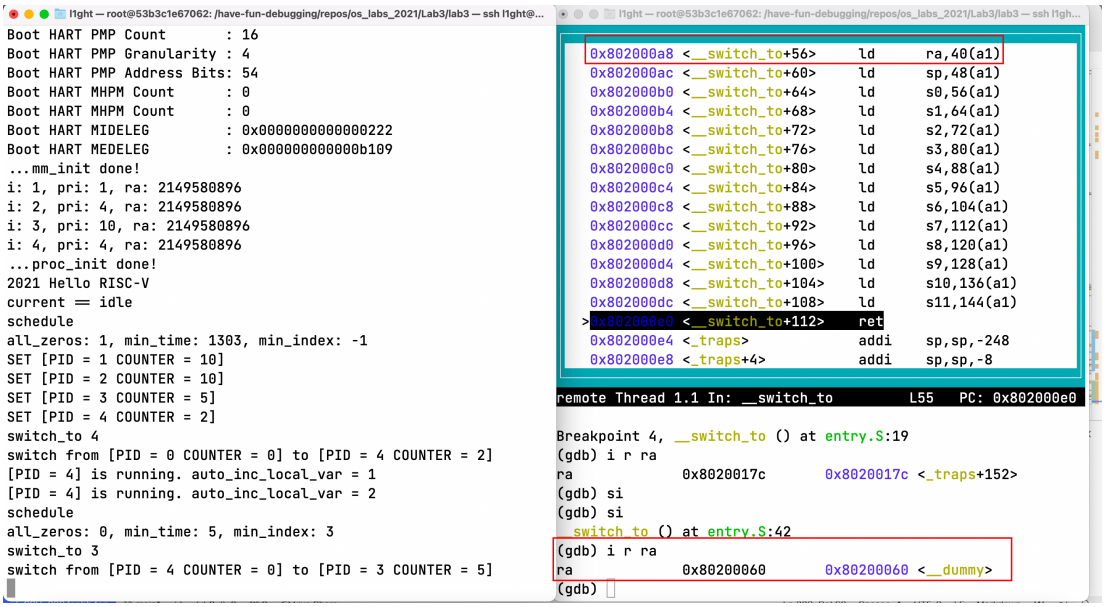
此时刚刚经历了 `schedule` 线程调度，发现需要切换到PID=3的线程，通过调用 `switch_to` 来进入 `__switch_to` ，以保存上文并重新载入新线程的下文，此时我们能够发现ra寄存器的值对应的并不是需要进入的 `__dummy` 地址，而是在 `entry.S` 的 `_traps` 中



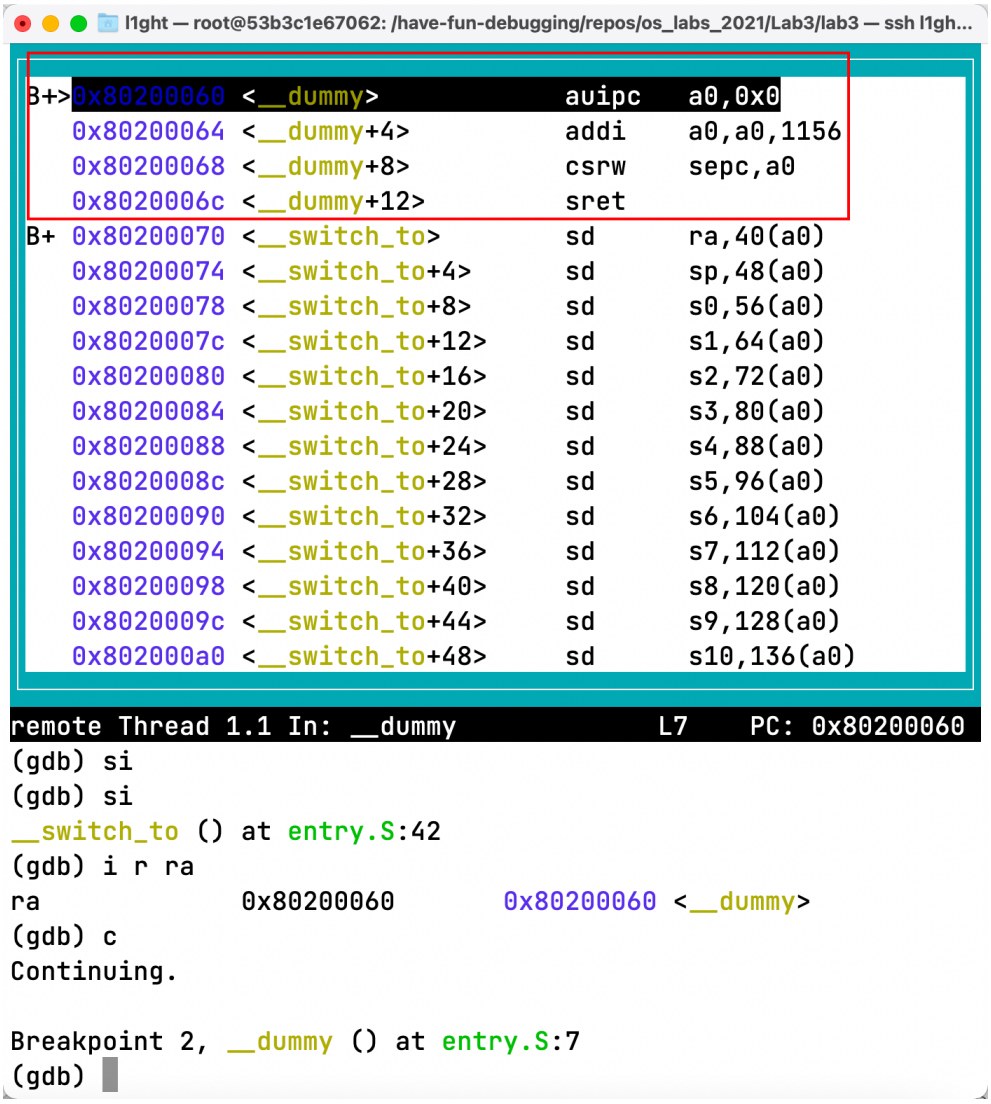
我们回头看，发现是由于在 `_trap` 中执行了 `call trap_handler` ，以将ra寄存器赋值成了 `entry.S` 中 `call trap_handler` 之后的地址，不过这并不会影响我们的线程切换，因为我们将马上在 `__switch_to` 中将正确的返回地址load入ra寄存器

```
0x80200178 <_traps+148> jal      ra,0x80200bb8 <trap_handler>
0x8020017c <_traps+152> ld       a0,0(sp)
0x80200180 <_traps+156> csrw     sepc,a0
0x80200184 <_traps+160> ld       zero,248(sp)
0x80200188 <_traps+164> ld       ra,240(sp)
0x8020018c <_traps+168> ld       gp,232(sp)
0x80200190 <_traps+172> ld       tp,224(sp)
0x80200194 <_traps+176> ld       t0,216(sp)
0x80200198 <_traps+180> ld       t1,208(sp)
0x8020019c <_traps+184> ld       t2,200(sp)
0x802001a0 <_traps+188> ld       s0,192(sp)
```

下图已从PID=4的线程PCB中load出正确的上下文环境，此时再查看ra寄存器的值，已经是我们想要的 `__dummy` 了



再继续执行后会到达 `__dummy` ，继而通过 `sret` 回到 `dummy` 中继续循环并等待下一次时钟中断，回到 `dummy` 之后线程切换已经完成，后续将会再次返回


l1ght — root@53b3c1e67062: /have-fun-debugging/repos/os_labs_2021/Lab3/lab3 — ssh l1gh...
B+>0x80200060 <__dummy> auipc a0,0x0
0x80200064 <__dummy+4> addi a0,a0,1156
0x80200068 <__dummy+8> csrw sepc,a0
0x8020006c <__dummy+12> sret
B+ 0x80200070 <__switch_to> sd ra,40(a0)
0x80200074 <__switch_to+4> sd sp,48(a0)
0x80200078 <__switch_to+8> sd s0,56(a0)
0x8020007c <__switch_to+12> sd s1,64(a0)
0x80200080 <__switch_to+16> sd s2,72(a0)
0x80200084 <__switch_to+20> sd s3,80(a0)
0x80200088 <__switch_to+24> sd s4,88(a0)
0x8020008c <__switch_to+28> sd s5,96(a0)
0x80200090 <__switch_to+32> sd s6,104(a0)
0x80200094 <__switch_to+36> sd s7,112(a0)
0x80200098 <__switch_to+40> sd s8,120(a0)
0x8020009c <__switch_to+44> sd s9,128(a0)
0x802000a0 <__switch_to+48> sd s10,136(a0)
remote Thread 1.1 In: __dummy L7 PC: 0x80200060
(gdb) si
(gdb) si
__switch_to () at entry.S:42
(gdb) i r ra
ra 0x80200060 0x80200060 <__dummy>
(gdb) c
Continuing.
Breakpoint 2, __dummy () at entry.S:7
(gdb)