浙江大學

课程名称：	操作系统原理与实践
题　　目：	Lab 2: RV64 内核线程调度
授课教师：	申文博
助　　教：	王鹤翔、陈淦豪、许昊瑞
姓　　名：	潘潇然
学　　号：	3220106049
地　　点：	32舍367

一、实验过程与步骤

1. 准备工程

- 在 `haed.S` 的 `_start` 中的 `jal start_kernel` 前加上 `jal mm_init`
- 在 `defs.h` 中添加对应内容

2. 线程初始化

- 首先补充完成 `arch/riscv/kernel/proc.c` 的 `task_init()` 函数。我们首先每次初始化调用 `kalloc()` 分配一个物理页，同时 `state` 设置为 `TASK_RUNNING` (本次实验只有这一种状态)。
    - 对于 `idle` ,由于其不参与调度，因此我们将其 `counter` , `priority` 和 `pid` 均设为0，最后将 `current` 指针和 `task[0]` 都指向 `idle`
    - 对于其他线程，首先初始化 `counter` 为0， `priority` 为随机数，其值在限定的最大值和最小值之间。之后，将 `ra` 设置为之后会完成的 `__dummy` 的地址， `sp` 设置为物理页高地址。根据内存布局和页大小我们可以知道 `sp` 地址为 `task_page` 加上 `PAGE_SIZE`

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

  void *idle_page = kalloc();
  if (!idle_page) {
    printk("kalloc failed\n");
    return;
  }
  idle = (struct task_struct *)idle_page;
  idle->state = TASK_RUNNING;
  idle->counter = 0;
  idle->priority = 0;
  idle->pid = 0;
  current = idle;
  task[0] = idle;

  for (int i = 1; i < NR_TASKS; ++i) {
    void *task_page = kalloc();
    if (!task_page) {
      printk("kalloc failed\n");
      return;
    }
    task[i] = (struct task_struct *)task_page;
    task[i]->pid = i;
```

```c
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority =
            PRIORITY_MIN + rand() % (PRIORITY_MAX - PRIORITY_MIN + 1);
        task[i]->thread.ra = (uint64_t)__dummy;
        task[i]->thread.sp = (uint64_t)((unsigned long)task_page + PAGE_SIZE);
    }

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

- 之后在 `haed.S` 的 `_start` 中的 `jal start_kernel` 前加上 `jal task_init`

3. `__dummy` 与 `dummy` 的实现

- 在 `arch/riscv/kernel/entry.S` 中添加 `__dummy` 。首先获取 `dummy` 的地址，之后将 `dummy` 值存储到 `sepc` ， 最后 `sret` 从S模式返回

```asm
    .extern dummy
    .globl __dummy
__dummy:
    la a0,dummy
    csrw sepc,a0
    sret
```

4. 实现线程切换

- 函数 `switch_to` 判断当前线程和下一个执行的线程是否为同一个线程，若是则不进行任何处理，否则用 `prev` 储存当前线程，并将当前线程设为下一线程 `next` ，最后对 `prev` 和 `next` 调用 `__switch_to` 进行线程切换

```c
void switch_to(struct task_struct *next) {
  if (current == next) return;
  printk("\nswitch to [PID = %d PRIORITY = %d COUNTER = %d]\n", next->pid,
         next->priority, next->counter);
  struct task_struct *prev = current;
  current = next;
  __switch_to(prev, next);
}
```

- 在 `__switch_to` 中，我们保存当前线程相关数据，并载入下一执行线程。这里要注意的是我们传入的指针是 `task_struct` 类型，而在 `task_struct` 结构体中， `thread` 之前还有四个 `uint64_t` 类型的变量，各占8字节，共占用32字节。因此每次载入或保存前要首先加32。之后依次对 `ra` ， `sp` ， `s0~s11` 进行相关操作，最后 `ret` 返回。

```c
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;
};
```

```asm
__switch_to:
    # save state to prev process
    addi t0,a0,32
    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
    addi t0,a1,32
    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
```

5. 实现调度入口函数

- 实现 `do_timer` 函数。若当前线程为 `idle` 或当前 `counter` 为0，则直接进行 `schedule()`，否则 `counter` 减1并返回

```
void do_timer() {
  if (current == idle || current->counter == 0)
    schedule();
  else if (current->counter > 0) {
    current->counter--;
    return;
  }
}
```

- 之后在 `trap.c` 中加入 `do_timer()`

```
if ((scause & ~flag) == exception_code) {  // if timer interrupt
      //   printk("[S] Supervisor Mode Timer Interrupt\n");
      clock_set_next_event();
      do_timer();
    }
```

6. 线程调度算法实现。

- 调度时首先扫描所有线程，执行目前 `counter` 最大的线程，若有多个 `counter` 相同且非零的线程，则优先执行 `pid` 小的线程
- 若所有线程都为0，则令所有线程 `counter` 值为 `priority` 值，即优先执行 `priority` 高的线程
- 之后重新获取 `counter` 最大的线程，并通过 `switch_to` 切换到对应的线程

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

  for (int i = 0; i < NR_TASKS; i++) {
    if (task[i]->counter > max_counter) {
      max_counter = task[i]->counter;
      next = task[i];
    }
  }

  if (!max_counter) {
    for (int i = 1; i < NR_TASKS; i++) {
```

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

  if (next) {
    switch_to(next);
  }
}
```

7. 编译及测试

- `make TEST_SCHED=1 run`输出，与提供的正确输出一致，通过测试

```
...mm_init done!
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 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 [PID = 1 PRIORITY = 7 COUNTER = 7]
[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
[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

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[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

switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
[PID = 4] is running. auto_inc_local_var = 1
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[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
[PID = 2] is running. auto_inc_local_var = 15
[PID = 2] is running. auto_inc_local_var = 16
[PID = 2] is running. auto_inc_local_var = 17
[PID = 2] is running. auto_inc_local_var = 18
[PID = 2] is running. auto_inc_local_var = 19
[PID = 2] is running. auto_inc_local_var = 20

switch to [PID = 1 PRIORITY = 7 COUNTER = 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
[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

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
    Output: 2222222222111111133334222222222211111113
```

- `make run` 的部分输出，可以观察到同样符合代码逻辑

```
...mm_init done!
...task_init done!
```

```
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 1]
SET [PID = 5 PRIORITY = 4 COUNTER = 4]
SET [PID = 6 PRIORITY = 7 COUNTER = 7]
SET [PID = 7 PRIORITY = 5 COUNTER = 5]
SET [PID = 8 PRIORITY = 10 COUNTER = 10]
SET [PID = 9 PRIORITY = 1 COUNTER = 1]
SET [PID = 10 PRIORITY = 9 COUNTER = 9]
SET [PID = 11 PRIORITY = 6 COUNTER = 6]
SET [PID = 12 PRIORITY = 9 COUNTER = 9]
SET [PID = 13 PRIORITY = 6 COUNTER = 6]
SET [PID = 14 PRIORITY = 6 COUNTER = 6]
SET [PID = 15 PRIORITY = 5 COUNTER = 5]
SET [PID = 16 PRIORITY = 8 COUNTER = 8]
SET [PID = 17 PRIORITY = 1 COUNTER = 1]
SET [PID = 18 PRIORITY = 5 COUNTER = 5]
SET [PID = 19 PRIORITY = 3 COUNTER = 3]
SET [PID = 20 PRIORITY = 7 COUNTER = 7]
SET [PID = 21 PRIORITY = 7 COUNTER = 7]
SET [PID = 22 PRIORITY = 3 COUNTER = 3]
SET [PID = 23 PRIORITY = 3 COUNTER = 3]
SET [PID = 24 PRIORITY = 3 COUNTER = 3]
SET [PID = 25 PRIORITY = 4 COUNTER = 4]
SET [PID = 26 PRIORITY = 3 COUNTER = 3]
SET [PID = 27 PRIORITY = 9 COUNTER = 9]
SET [PID = 28 PRIORITY = 1 COUNTER = 1]
SET [PID = 29 PRIORITY = 9 COUNTER = 9]
SET [PID = 30 PRIORITY = 10 COUNTER = 10]
SET [PID = 31 PRIORITY = 3 COUNTER = 3]

switch to [PID = 2 PRIORITY = 10 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 [PID = 8 PRIORITY = 10 COUNTER = 10]
[PID = 8] is running. auto_inc_local_var = 1
```

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

switch to [PID = 12 PRIORITY = 9 COUNTER = 9]
[PID = 12] is running. auto_inc_local_var = 1
[PID = 12] is running. auto_inc_local_var = 2
[PID = 12] is running. auto_inc_local_var = 3
[PID = 12] is running. auto_inc_local_var = 4
[PID = 12] is running. auto_inc_local_var = 5
[PID = 12] is running. auto_inc_local_var = 6
[PID = 12] is running. auto_inc_local_var = 7
[PID = 12] is running. auto_inc_local_var = 8
[PID = 12] is running. auto_inc_local_var = 9

switch to [PID = 27 PRIORITY = 9 COUNTER = 9]
[PID = 27] is running. auto_inc_local_var = 1
[PID = 27] is running. auto_inc_local_var = 2
```

```
[PID = 27] is running. auto_inc_local_var = 3
[PID = 27] is running. auto_inc_local_var = 4
```

二、实验心得

这次实验主要卡在了两个地方，一个一开始 `__switch_to` 没有考虑到 `task_struct` 和 `thread_struct` 结构体在内存布局上的区别，因此一开始没有加上32，后面才注意到。另一个是在 `trap.c` 中没有弄清楚 `clock_set_next_event()` 和 `do_timer()` 这两个函数的顺序，导致逻辑出现了一些混乱。具体来说就是这两个函数如果先调用 `do_timer()` 再调用 `clock_set_next_event()` ，会导致在进程没有上下文时输出少一次。

三、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?

- 这是因为 `__switch_to` 是在C语言的 `switch_to` 函数中被调用的，而在上一个实验中我们已经知道了有Caller Saved Register和Callee Saved Register的区别。而C语言在函数调用过程中会自动保存Caller Saved Register的部分，因此我们只需要保存Callee Saved Register(`sp` 及 `s0~s11`)以及保存了 `__switch_to` 函数调用点 `ra` 即可

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` 调用 `kfreerange` 来释放 `_ekernel` 到 `PHY_END` 之间的内存。其中 `_ekernel` 代表kernel代码的结束地址， `PHY_END` 代表物理内存的结束位置，在 `defs.h` 中 `PHY_END` 被定义为 `PHY_START+PHY_SIZE` ，其中前者是起始地址，即0，后者是QEMU的默认内存大小128MiB。而 `kfreerange` 函数首先将 `start` 地址上对齐到页面边界，再以 `PGSIZE` 为步长，释放每个页面
- 内存管理是通过 `freelist` 完成的，所有可用的内存都储存到其中。当调用 `kalloc` 的时候，从 `freelist` 取出一个可用的内存块并将其中的内容清零以避免内存泄漏。而调用 `kfree` 的时候首先对齐地址，再清零内存块内容，最后放入自由链表。

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）。

- 首先在终端1输入 `make TEST_SCHED=1 debug` 启动qemu以减少运行的线程，终端2输入 `gdb-multiarch vmlinux` ，注意这两个终端都在本次实验工程根目录下运行，以运行正确的内核
- 启动gdb后 `target remote :1234`
- 分别在 `__dummy` 和 `__switch_to` 设置两个断点

```
>>> b __dummy
Breakpoint 1 at 0x80200178: file entry.S, line 88.
>>> b __switch_to
Breakpoint 2 at 0x80200188: file entry.S, line 95.
```

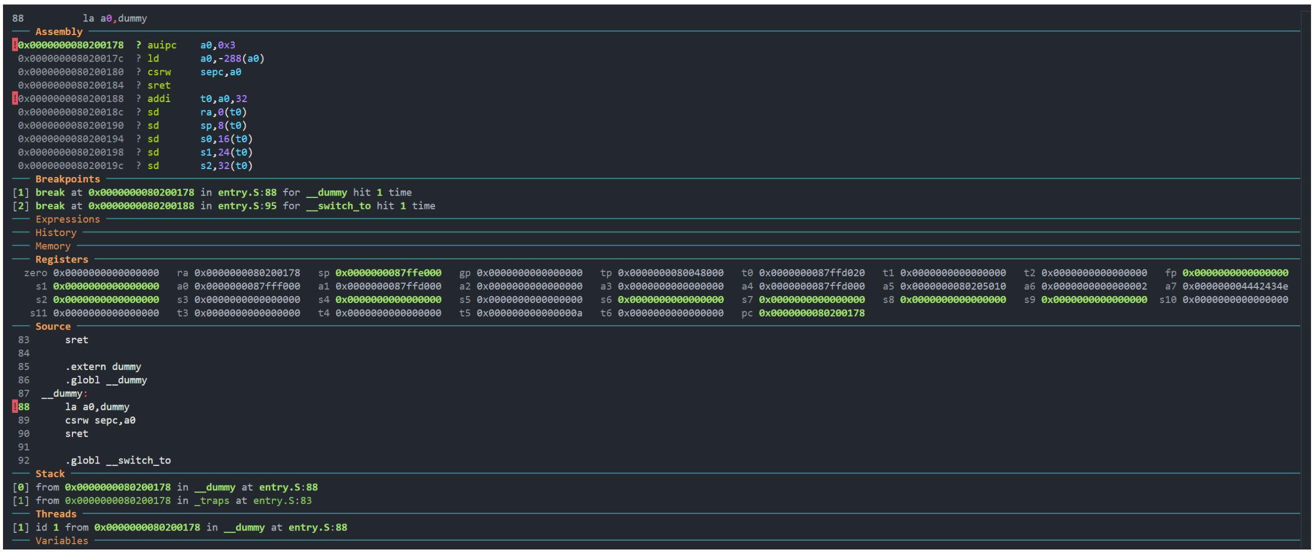
- 运行 `c` ，切换到 `pid=2` 线程，触发 `__switch_to` 断点，此时将要存进 `ra` 的值是 `0x0000000080200b7c` ，对应 `switch_to+128 at proc.c:161` ，即C语言中 `switch_to` 函数被调用的下一行。

```
终端
 0x0000000080200190  ? sd       sp,8(t0)
 0x0000000080200194  ? sd       s0,16(t0)
 0x0000000080200198  ? sd       s1,24(t0)
 0x000000008020019c  ? sd       s2,32(t0)
 0x00000000802001a0  ? sd       s3,40(t0)
 0x00000000802001a4  ? sd       s4,48(t0)
 0x00000000802001a8  ? sd       s5,56(t0)
 0x00000000802001ac  ? sd       s6,64(t0)
── Breakpoints ──────────────────────────────────────────────
[1] break at 0x0000000080200178 in entry.S:88 for __dummy
[2] break at 0x0000000080200188 in entry.S:95 for __switch_to hit 1 time
── Expressions ──────────────────────────────────────────────
── History ──────────────────────────────────────────────────
── Memory ───────────────────────────────────────────────────
── Registers ────────────────────────────────────────────────
  zero 0x0000000000000000   ra 0x0000000080200b7c   sp 0x0000000080204e30   gp 0x0000000000000000   tp 0x0000000080048000   t0 0x0000000000000000   t1 0x0000000000000000   t2 0x0000000000000000   fp 0x0000000080204e60
    s1 0x0000000000000001   a0 0x0000000087fff000   a1 0x0000000087ffd000   a2 0x0000000000000000   a3 0x0000000000000000   a4 0x0000000087ffd000   a5 0x0000000080205010   a6 0x0000000000000002   a7 0x000000004442434e
    s2 0x0000000080200000   s3 0x0000000000000000   s4 0x0000000087e00000   s5 0x0000000000000000   s6 0x8000000a00006800   s7 0x0000000000000001   s8 0x0000000000002000   s9 0x0000000080043710  s10 0x0000000000000000
   s11 0x0000000000000000   t3 0x0000000000000000   t4 0x0000000000000000   t5 0x000000000000000a   t6 0x0000000000000000   pc 0x0000000080200188
── Source ───────────────────────────────────────────────────
 90      sret
 91
 92      .globl __switch_to
 93  __switch_to:
 94      # save state to prev process
!95      addi t0,a0,32
 96      sd ra, 0(t0)
 97      sd sp, 8(t0)
 98      sd s0, 16(t0)
 99      sd s1, 24(t0)
── Stack ────────────────────────────────────────────────────
[0] from 0x0000000080200188 in __switch_to at entry.S:95
[1] from 0x0000000080200b7c in switch_to+128 at proc.c:161
[2] from 0x0000000080200a60 in schedule+540 at proc.c:138
[3] from 0x0000000080200ab8 in do_timer+68 at proc.c:148
[4] from 0x0000000080200ec4 in trap_handler+92 at trap.c:19
[5] from 0x00000000802000ec in _traps at entry.S:44
── Threads ──────────────────────────────────────────────────
[1] id 1 from 0x0000000080200188 in __switch_to at entry.S:95
── Variables ────────────────────────────────────────────────
```

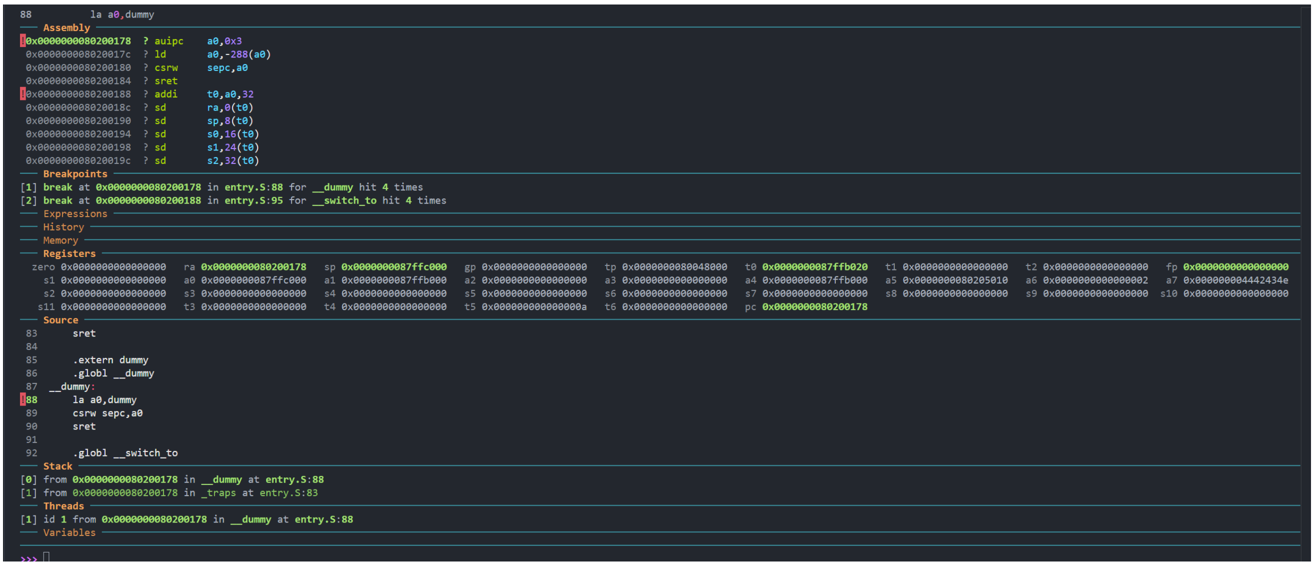
- 之后 `si` 到 `ld ra, 0(t0)` 发现 `ra` 读取的值为 `__dummy` 的值

```
114        ld sp, 8(t0)
── Assembly ─────────────────────────────────────────────────
 0x00000000802001cc  ? ld       sp,8(t0)
 0x00000000802001d0  ? ld       s0,16(t0)
 0x00000000802001d4  ? ld       s1,24(t0)
 0x00000000802001d8  ? ld       s2,32(t0)
 0x00000000802001dc  ? ld       s3,40(t0)
 0x00000000802001e0  ? ld       s4,48(t0)
 0x00000000802001e4  ? ld       s5,56(t0)
 0x00000000802001e8  ? ld       s6,64(t0)
 0x00000000802001ec  ? ld       s7,72(t0)
 0x00000000802001f0  ? ld       s8,80(t0)
── Breakpoints ──────────────────────────────────────────────
[1] break at 0x0000000080200178 in entry.S:88 for __dummy
[2] break at 0x0000000080200188 in entry.S:95 for __switch_to hit 1 time
── Expressions ──────────────────────────────────────────────
── History ──────────────────────────────────────────────────
── Memory ───────────────────────────────────────────────────
── Registers ────────────────────────────────────────────────
  zero 0x0000000000000000   ra 0x0000000080200178   sp 0x0000000080204e30   gp 0x0000000000000000   tp 0x0000000080048000   t0 0x0000000087ffd020   t1 0x0000000000000000   t2 0x0000000000000000   fp 0x0000000080204e60
    s1 0x0000000000000001   a0 0x0000000087fff000   a1 0x0000000087ffd000   a2 0x0000000000000000   a3 0x0000000000000000   a4 0x0000000087ffd000   a5 0x0000000080205010   a6 0x0000000000000002   a7 0x000000004442434e
    s2 0x0000000080200000   s3 0x0000000000000000   s4 0x0000000087e00000   s5 0x0000000000000000   s6 0x8000000a00006800   s7 0x0000000000000001   s8 0x0000000000002000   s9 0x0000000080043710  s10 0x0000000000000000
   s11 0x0000000000000000   t3 0x0000000000000000   t4 0x0000000000000000   t5 0x000000000000000a   t6 0x0000000000000000   pc 0x00000000802001cc
── Source ───────────────────────────────────────────────────
 109      sd s11, 104(t0)
 110
 111      # restore state from next process
 112      addi t0,a1,32
 113      ld ra, 0(t0)
 114      ld sp, 8(t0)
 115      ld s0, 16(t0)
 116      ld s1, 24(t0)
 117      ld s2, 32(t0)
 118      ld s3, 40(t0)
── Stack ────────────────────────────────────────────────────
[0] from 0x00000000802001cc in __switch_to at entry.S:114
[1] from 0x0000000080200178 in _traps at entry.S:83
── Threads ──────────────────────────────────────────────────
[1] id 1 from 0x00000000802001cc in __switch_to at entry.S:114
── Variables ────────────────────────────────────────────────
```

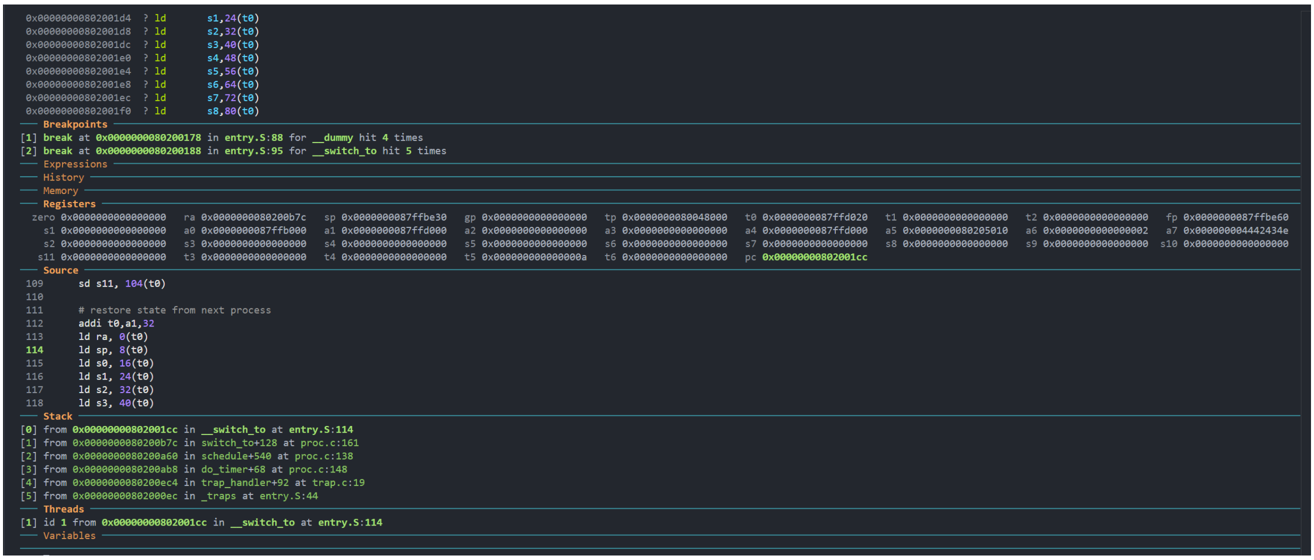
- 同时 `c` 会触发 `__dummy` 断点，说明进入了 `__dummy`



- 如此过程重复四次，可以得到相同的结果



- 之后重复第五次，再次触发 `__switch_to` 断点，发现 `ra` 中将要储存的值依旧为C语言中 `switch_to` 函数被调用的下一行。但再次单步到 `ld ra, 0(t0)` 后发现 `ra` 的值不发生变化，没有被重置为 `__dummy` 的值



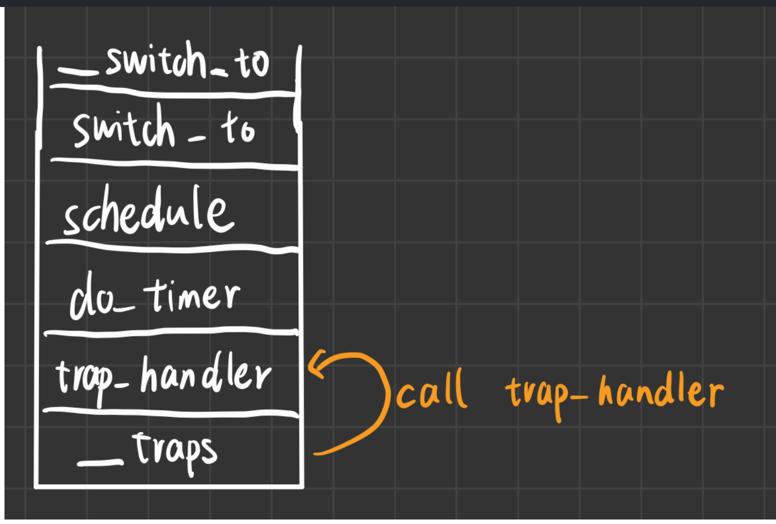
- 以上 `TEST_SCHED=1` 模式下，一共有四个线程，由于初始没有上下文，且我们在初始化函数中将其设置为了 `__dummy` 的值，因此每个线程第一次进入 `__switch_to` 的时候会

被恢复为 `__dummy` 的值，而在之后就不会恢复为 `__dummy` 的值，都保持为C语言中 `switch_to` 函数的下一行。

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。)

- 我们启动gdb，在 `__switch_to` 设置断点，运行 `c` 两次，再运行 `bt` ，从而得到在输出第二次进程切换信息时的函数帧栈布局，如下示意图所示。

```
>>> bt
#0  __switch_to () at entry.S:95
#1  0x000000008020b7c in switch_to (next=0x87ffe000) at proc.c:161
#2  0x0000000080200a60 in schedule () at proc.c:138
#3  0x0000000080200ab8 in do_timer () at proc.c:148
#4  0x0000000080200ed0 in trap_handler (scause=9223372036854775813, sepc=2149582460) at trap.c:19
#5  0x00000000802000ec in _traps () at entry.S:44
Backtrace stopped: frame did not save the PC
```



- 首先是启动程序后，运行 `main.c` 中的 `test()` 函数
- 每隔一段时间，触发一次时钟中断(只有时钟中断才会进入后续的 `do_timer`)，这时进入 `entry.S` 中的 `_traps` 。在 `_traps` 中，先保存32个寄存器的值到栈上，之后调用 `trap_handler` 处理
- 在 `trap_handler` 中，由于检测到是时钟中断，因此在 `clock_set_next_event()` 之后会进入 `do_timer()` 进行进程调度相关处理
- 在 `do_timer` 中，若当前进程的 `counter` 大于0，则减少1即可，不会触发进程切换。若当前进程为 `idle` (`idle` 的 `counter` 我们设为了0)或当前进程的 `counter` 为0，则调用 `schedule()` 进行进程调度
- 在 `schedule` 中，通过一系列处理获取下一个要切换到的进程 `next` ，并调用 `switch_to(next)` 进行切换
- 在 `switch_to` 中，我们处理好 `current` 后，调用 `__switch_to` 进行进程切换，到此结束