

浙江大学

操作系统实验二

课程名称:	操作系统
作业名称:	RV64 内核线程调度
姓 名:	仇国智
学 号:	3220102181
电子邮箱:	3220102181@zju.edu.cn
联系电话:	13714176104
指导教师:	申文博

2024 年 10 月 13 日

目录

1	实验内容及原理	1
1.1	实验内容	1
1.2	实验原理	1
2	实验具体过程与代码实现	3
2.1	准备	3
2.2	线程初始化	4
2.3	__dummy 函数	6
2.4	实现线程切换	6
2.5	实现调度入口函数	7
2.6	线程调度算法实现	8
2.7	添加初始化	9
3	实验结果与分析	9
4	遇到的问题及解决方法	10
4.1	问题一: 没有初始化	10
4.2	问题二: 混淆了 task_struct 和 thread_struct	10
5	思考题	10
5.1	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?	10
5.2	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.	11
5.3	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).	11
5.4	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。	20

1 实验内容及原理

1.1 实验内容

- 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.

1.2 实验原理

进程和线程

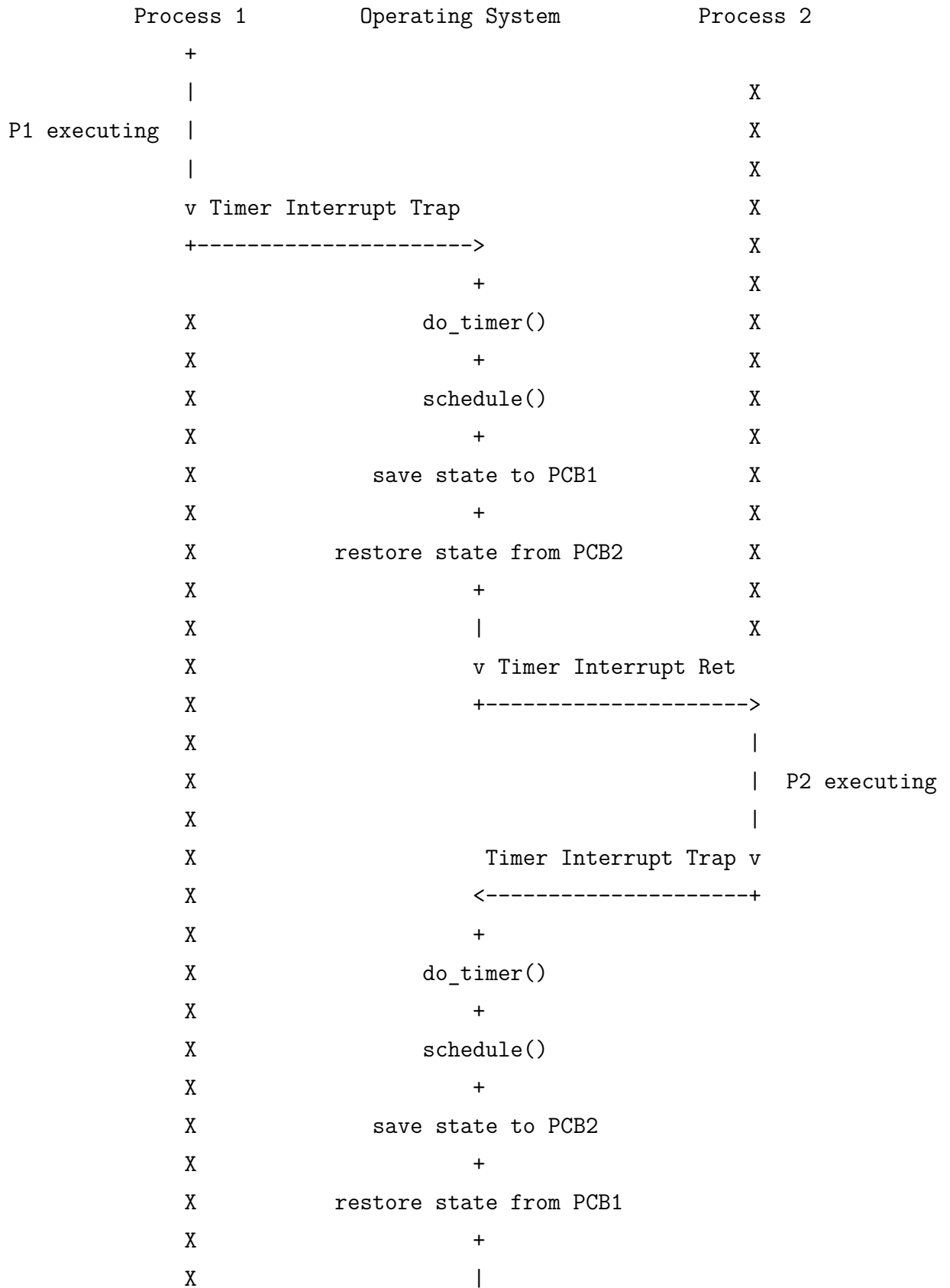
源代码经编译器一系列处理 (编译, 链接, 优化等) 后得到的可执行文件, 我们称之为程序 (Program). 而通俗地说, 进程就是正在运行并使用计算机资源的程序. 进程与程序的不同之处在于, 进程是一个动态的概念, 其不仅需要将其运行的程序的代码 / 数据等加载到内存空间中, 还需要拥有自己的运行栈. 同时一个进程可以对应一个或多个线程, 线程之间往往具有相同的代码, 共享一块内存, 但是却有不同 CPU 执行状态.

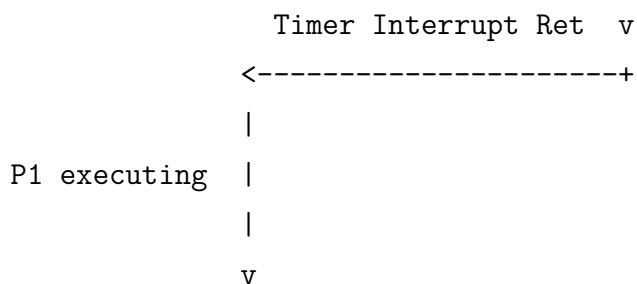
线程相关属性

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.

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.

线程调度算法

本次实验我们需要参考 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

2 实验具体过程与代码实现

2.1 准备

从实验仓库同步以下代码:

```

arch
    riscv
        include
            mm.h
            proc.h
        kernel
            mm.c          # 一个简单的物理内存管理接口
            proc.c        # 本次实验的重点部分,进行线程的管理
    include
        stdlib.h          # rand 及 srand 在这里(与 C 语言 stdlib.h 一致)
        string.h          # memset 在这里(与 C 语言 string.h 一致)
    lib
        rand.c            # rand 和 srand 的实现(参考 musl libc)
        string.c          # memset 的实现

```

2.2 线程初始化

在 proc.c 中, 编写 task_init 函数

```

1  srand(2024);
2  for(int i = 0; i < NR_TASKS; i++) {
3      task[i] = NULL;
4  }
5  // 1. 调用 kalloc() 为 idle 分配一个物理页
6  // 2. 设置 state 为 TASK_RUNNING;
7  // 3. 由于 idle 不参与调度,可以将其 counter / priority 设置为 0
8  // 4. 设置 idle 的 pid 为 0
9  // 5. 将 current 和 task[0] 指向 idle
10 uint64_t p = (uint64_t)kalloc();
11 idle = (struct task_struct *)p;
12 idle->state = TASK_RUNNING;
13 idle->counter = 0;
14 idle->priority = 0;
15 idle->pid = 0;
16 current = idle;
17 task[0] = idle;

```

```

18
19  /* YOUR CODE HERE */
20
21  // 1. 参考 idle 的设置,为 task[1] ~ task[NR_TASKS - 1] 进行初始化
22  // 2. 其中每个线程的 state 为 TASK_RUNNING, 此外,counter 和
    priority 进行如下赋值:
23  // - counter = 0;
24  // - priority = rand() 产生的随机数(控制范围在 [PRIORITY_MIN,
    PRIORITY_MAX] 之间)
25  // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中的 ra
    和 sp
26  // - ra 设置为 __dummy(见 4.2.2)的地址
27  // - sp 设置为该线程申请的物理页的高地址
28
29  /* YOUR CODE HERE */
30  for(int i=0;i<NR_TASKS;i++){
31      if(task[i]==NULL)
32      {
33          uint64_t p = (uint64_t)kalloc();
34          task[i] = (struct task_struct *)p;
35          task[i]->state = TASK_RUNNING;
36          task[i]->counter = 0;
37          task[i]->priority = PRIORITY_MIN + rand() % (PRIORITY_MAX -
    PRIORITY_MIN + 1);
38          task[i]->pid = i;
39          task[i]->thread.ra = (uint64_t)__dummy;
40          task[i]->thread.sp = p + PGSIZE;
41          task[i]->thread.s[0] = dummy; // 用于指示返回的开始函数
42      }
43  }

```

我们首先为 idle 线程分配一个物理页,并设置其状态为 TASK_RUNNING,counter 和 priority 为 0,然后将其 pid 设置为 0,并将 current 和 task[0] 指向 idle 线程.接着我们为 task[1] 到 task[NR_TASKS-1] 进行初始化,设置其状态为 TASK_RUNNING,counter 为 0,priority 为 PRIORITY_MIN 到 PRIORITY_MAX 之间的随机数,并设置其

pid 为 i,ra 为 `__dummy` 的地址,sp 为该线程申请的物理页的高地址. 并且设置其 `thread.s[0]` 为 `dummy`, 用于指示返回的开始函数 (便于之后添加 `exec` 生成进程).

2.3 `__dummy` 函数

在 `entry.S` 中, 编写 `__dummy` 函数

```
1 __dummy:
2     # YOUR CODE HERE
3     csrw sepc, s0
4     sret
```

我们在 `__dummy` 函数中, 将 `sepc` 设置为 `s0`, 然后执行 `sret` 指令, 返回到 `s0` 指向的地址. 之所以需要这样做, 而不是直接在 `__switch_to` 设置对应的 `ra`, 是因为在切换进程时, 系统处于内核中, 需要通过 `sret` 指令返回到用户态 (虽然这里还没有用户态).

2.4 实现线程切换

编写 `switch_to` 函数:

```
1 void switch_to(struct task_struct *next) {
2     if(current == next) return;
3     struct task_struct *prev = current;
4     current = next;
5     // printk("switch to task %d\n", next->pid);
6     __switch_to(prev, next);
7 }
```

我们首先判断当前进程是否为下一个进程, 如果是则直接返回. 然后将当前进程设置为下一个进程, 并调用 `__switch_to` 函数进行切换. 接着我们编写 `__switch_to` 函数:

```
1 .globl __switch_to
2 __switch_to:
3     addi a0, a0, 0x20
4     addi a1, a1, 0x20
5     sd ra, 0(a0)
6     sd sp, 8(a0)
7     sd s0, 16(a0)
8     sd s1, 24(a0)
```



```

9      sd s2, 32(a0)
10     sd s3, 40(a0)
11     sd s4, 48(a0)
12     sd s5, 56(a0)
13     sd s6, 64(a0)
14     sd s7, 72(a0)
15     sd s8, 80(a0)
16     sd s9, 88(a0)
17     sd s10, 96(a0)
18     sd s11, 104(a0)
19     ld ra, 0(a1)
20     ld sp, 8(a1)
21     ld s0, 16(a1)
22     ld s1, 24(a1)
23     ld s2, 32(a1)
24     ld s3, 40(a1)
25     ld s4, 48(a1)
26     ld s5, 56(a1)
27     ld s6, 64(a1)
28     ld s7, 72(a1)
29     ld s8, 80(a1)
30     ld s9, 88(a1)
31     ld s10, 96(a1)
32     ld s11, 104(a1)
33     ret

```

我们首先将 a0 和 a1 加上 0x20, 将 a0 和 a1 指向的地址设置为 prev 和 next 的 task_struct 结构体的地址. 然后我们保存 prev 的 ra,sp,s0 到 s11, 并将 next 的 ra,sp,s0 到 s11 载入到对应的寄存器中. 最后执行 ret 指令, 返回到 next 的 ra 指向的地址.

2.5 实现调度入口函数

完成 do_timer 函数:

```

1      void do_timer() {
2          // 1. 如果当前线程是 idle
           线程或当前线程时间片耗尽则直接进行调度

```

```

3      // 2. 否则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
4      if(current == idle)
5          schedule();
6      current->counter--;
7      if((int)current->counter < 0)
8          current->counter = 0;
9      // printk("kernel current->counter = %d\n", current->counter);
10     if(current->counter == 0)
11         schedule();
12 }

```

我们首先判断当前线程是否为 idle 线程, 或者当前线程的时间片是否耗尽, 如果是则直接进行调度. 然后我们将当前线程的 counter 减 1, 如果当前线程的 counter 小于 0, 则将其设置为 0. 接着我们判断当前线程的 counter 是否为 0, 如果是则进行调度.

2.6 线程调度算法实现

完成 schedule 函数:

```

1  void schedule() {
2      struct task_struct *next = NULL;
3      // printk("schedule\n");
4      for(int i = 1; i < NR_TASKS; i++) {
5          if(task[i] == NULL || task[i]->state !=
              TASK_RUNNING || task[i]->counter == 0) continue;
6          if(next == NULL || task[i]->counter > next->counter) {
7              next = task[i];
8          }
9      }
10     // printk("new circle\n");
11     if(next == NULL) {
12         for(int i = 1; i < NR_TASKS; i++) {
13             if(task[i] == NULL || task[i]->state != TASK_RUNNING)
14                 continue;
15             task[i]->counter = task[i]->priority;
16             if(next == NULL || task[i]->counter > next->counter) {

```

```

16         next = task[i];
17     }
18 }
19 }
20     switch_to(next);
21     return;
22 }

```

我们首先定义一个 `next` 指针, 用于指向下一个要执行的线程. 然后我们遍历 `task[1]` 到 `task[NR_TASKS-1]`, 如果 `task[i]` 为 `NULL`, 或者 `task[i]` 的状态不为 `TASK_RUNNING`, 或者 `task[i]` 的 `counter` 为 0, 则直接跳过. 然后我们判断 `next` 是否为 `NULL`, 或者 `task[i]` 的 `counter` 是否大于 `next` 的 `counter`, 如果是则将 `next` 指向 `task[i]`. 这样我们就找到了 `counter` 最大的线程, 同时如果 `next` 为 `NULL`, 则说明所有线程的 `counter` 都为 0, 我们需要重新设置所有线程的 `counter` 为其 `priority`, 然后再次遍历 `task[1]` 到 `task[NR_TASKS-1]`, 找到 `counter` 最大的线程. 最后我们调用 `switch_to` 函数进行线程切换.

2.7 添加初始化

在 `start_kernel` 中添加初始化代码:

```

1  int start_kernel() {
2      mm_init();
3      task_init();
4      test();
5      return 0;
6  }

```

3 实验结果与分析

输入如下命令编译并运行:

```

1  make TEST_SCHED=1 run

```

结果截图如下:

```
Supervisor timer interrupt
sepc: 0x80200940
[PID = 2] is running. auto_inc_local_var = 18
Supervisor timer interrupt
sepc: 0x80200940
[PID = 2] is running. auto_inc_local_var = 19
Supervisor timer interrupt
sepc: 0x80200940
[PID = 2] is running. auto_inc_local_var = 20
Switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
Supervisor timer interrupt
sepc: 0x80200940
[PID = 1] is running. auto_inc_local_var = 8
Supervisor timer interrupt
sepc: 0x80200940
[PID = 1] is running. auto_inc_local_var = 9
Supervisor timer interrupt
sepc: 0x80200950
[PID = 1] is running. auto_inc_local_var = 10
Supervisor timer interrupt
sepc: 0x80200940
[PID = 1] is running. auto_inc_local_var = 11
Supervisor timer interrupt
sepc: 0x80200950
[PID = 1] is running. auto_inc_local_var = 12
Supervisor timer interrupt
sepc: 0x80200940
[PID = 1] is running. auto_inc_local_var = 13
Supervisor timer interrupt
sepc: 0x80200940
[PID = 1] is running. auto_inc_local_var = 14
Switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
Supervisor timer interrupt
sepc: 0x80200940
[PID = 3] is running. auto_inc_local_var = 5
Test passed!
Output: 222222222211111113333422222222221111113
~/course/OS2U/os24fall-stu/src/lab2 lab2 12 48s Py base 18:50:01
```

图 1: 线程调度算法测试结果

通过测试

4 遇到的问题及解决方法

4.1 问题一: 没有初始化

在 `start_kernel` 中调用 `mm_init` 和 `task_init` 函数即可.

4.2 问题二: 混淆了 `task_struct` 和 `thread_struct`

在 `__switch_to` 函数中, 我们需要将存有 `prev` 和 `next` 的 `a0` 和 `a1` 加上 `0x20`, 将指向的地址设置为 `thread_struct` 的地址.

5 思考题

5.1 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?

回答: 在目前的系统中存在两种切换, 一是中断引发的切换, 二是线程切换. 前者需要保存全部的寄存器, 因为前者的产生不存在接口 (中断可以在任何时刻发生, 无法设定接口), 所以需要保存全部的寄存器. 而后者是通过函数的接口实现的, 即它的切换就像进行了函数调用一样, 所以只需要保存调用者保存的寄存器即可.

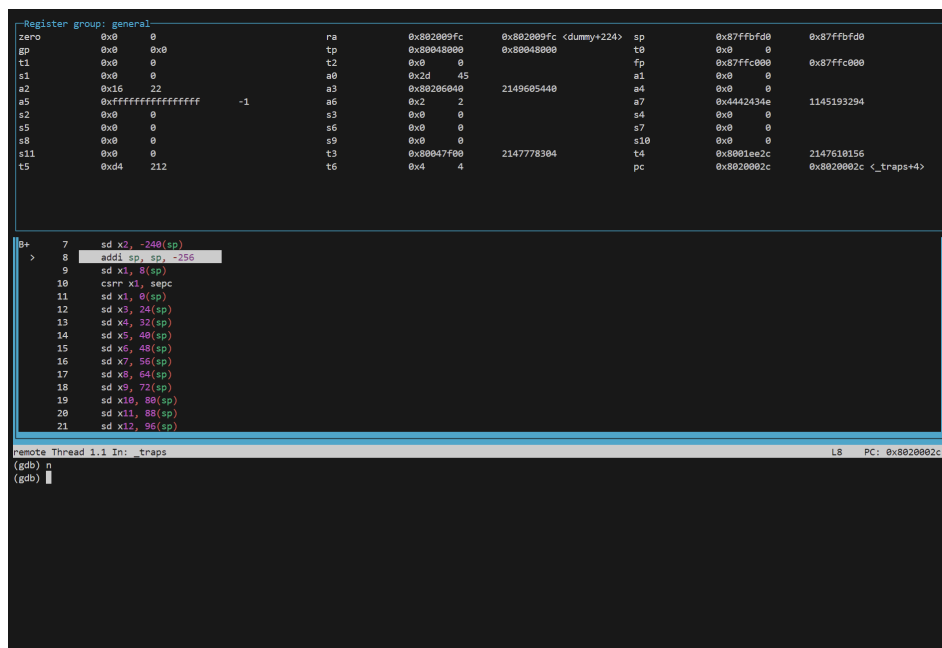
5.2 阅读并理解 arch/riscv/kernel/mm.c 代码, 尝试说明 mm_init 函数都做了什么, 以及在 kalloc 和 kfree 的时候内存是如何被管理的.

mm_init 释放了从内核末页到内存末页的内存 (内核已经加载到内存中, 所以不能释放内核所在的内存), 空闲的内存是通过链表的管理的, 每个页面在自己的头部有一个指向下一个页面的指针, 这样就可以通过遍历链表来找到空闲的内存.kalloc 就是将链表中的第一个页面分配出去, 然后将链表的头指针指向下一个页面,kfree 就是将当前页面储存链表的头指针, 然后将当前页面的头指针指向链表的头指针.

5.3 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).

若待转移线程为第一次接受转移, 则其 ra 所代表的返回点是 __dummy, 若不是第一次接受转移, 则 ra 所代表的返回点是其进程中 switch_to 函数中调用 __switch_to 函数的下一条指令的地址.

通过 gdb, 观察 ra 的变化如下:



```
Register group: general
zero 0x0 0 ra 0x802009fc 0x802009fc <dummy+224> sp 0x87ffbfd0 0x87ffbfd0
gp 0x0 0x0 tp 0x80048000 0x80048000
t1 0x0 0 t2 0x0 0 fp 0x87ffb000 0x87ffb000
a1 0x0 0 a0 0x2d 45 a1 0x0 0
a2 0x16 22 a3 0x802006040 2149605440 a4 0x0 0
a5 0xffffffffffffff -1 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 0x80047f00 2147778304 t4 0x8001ee2c 2147610156
t5 0xd4 212 t6 0x4 4 pc 0x8020002c 0x8020002c <_traps+4>

B+ 7 sd x2, -240(sp)
> 8 addi sp, sp, -256
9 sd x1, 8(sp)
10 csrw x1, sepc
11 sd x1, 0(sp)
12 sd x3, 24(sp)
13 sd x4, 32(sp)
14 sd x5, 40(sp)
15 sd x6, 48(sp)
16 sd x7, 56(sp)
17 sd x8, 64(sp)
18 sd x9, 72(sp)
19 sd x10, 80(sp)
20 sd x11, 88(sp)
21 sd x12, 96(sp)

remote Thread 1.1 In: _traps
(gdb) n
(gdb)
```

图 2: 进入 _traps

```

Register group: general
zero 0x0 0 ra 0x802000bc 0x802000bc < traps+148> sp 0x87ffbed0 0x87ffbed0
gp 0x0 0x0 tp 0x80048000 0x80048000 t0 0x0 0
t1 0x0 0 t2 0x0 0 t1 0x87fffc00 0x87fffc00
s1 0x0 0 a0 0x8000000000000005 -92233720368547 a1 0x80200950 2149583184
a2 0x16 22 a3 0x80206040 2149605440 a4 0x0 0
a5 0xffffffffff -1 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 0x80047f00 2147778304 t4 0x8001ee2c 2147610156
t5 0xd4 212 t6 0x4 4 pc 0x80200fac 0x80200fac < trap_handler

trap.c
59 if((int)current->counter < 0)
60 current->counter = 0;
61 // printk("kernel current->counter = %d\n", current->counter);
62 if(current->counter == 0)
63 schedule();
64 }
> void trap_handler(uint64_t scause, uint64_t sepc) {
65
66
67 if ((scause >> 63) == 0) {
68 // Handle exceptions
69 switch (scause) {
70 case 0x0:
71 printk("Instruction address misaligned\n");
72 break;
73 case 0x1:

```

Remote Thread 1.1 In: trap_handler L65 PC: 0x80200fac

```

(gdb) n
(gdb) n 29
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) si
(gdb) si
trap_handler (scause=2281684904, sepc=0) at trap.c:65
(gdb)

```

图 3: 进入 trap_handler

```

Register group: general
zero 0x0 0 ra 0x8020117c 0x8020117c < trap_handler sp 0x87ffbea0 0x87ffbea0
gp 0x0 0x0 tp 0x80048000 0x80048000 t0 0x0 0
t1 0x0 0 t2 0x0 0 t1 0x87fffc00 0x87fffc00
s1 0x0 0 a0 0x8000000000000005 -92233720368547 a1 0x80200950 2149583184
a2 0x16 22 a3 0x80206040 2149605440 a4 0x5 5
a5 0x5 5 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 0x80047f00 2147778304 t4 0x8001ee2c 2147610156
t5 0xd4 212 t6 0x4 4 pc 0x802001f0 0x802001f0 < clock_set_n

clock.c
8 // 编写内核汇编, 使用 rdtsc 获取 time 寄存器中 (也就是 mtime 寄存器) 的值并返回
9 uint64_t ret;
10 __asm__ volatile("rdtsc %0" : "=r"(ret));
11 return ret;
12 }
13
> void clock_set_next_event() {
14 // 下一次时钟中断的时间点
15 uint64_t next = get_cycles() + TIMECLOCK;
16
17 // 使用 sbi_set_timer 来完成对下一次时钟中断的设置
18 sbi_set_timer(next);
19 }
20
21 void clock_init()
22 {

```

Remote Thread 1.1 In: clock_set_next_event L14 PC: 0x802001f0

```

(gdb) n
(gdb) n
(gdb) si
(gdb) si
trap_handler (scause=2281684904, sepc=0) at trap.c:65
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) ni
(gdb) si
clock_set_next_event () at clock.c:14
(gdb)

```

图 4: 进入 clock_set_next_event

```

--Register group: general
zero    0x0 0
sp      0x87ffbe80 0x87ffbe80
tp      0x80048000 0x80048000
t1      0x0 0
fp      0x87ffbea0 0x87ffbea0
a0      0x1dc789 30459785
a2      0x1c18540 23459785
a6      0x2 2
s2      0x0 0
s4      0x0 0
s6      0x0 0
ra      0x80200224 0x80200224 <clock_set_next_event+52>
gp      0x0 0
t0      0x0 0
t2      0x0 0
s1      0x0 0
a1      0x80200950 2149583184
a3      0x80200940 2149585440
a5      0x1dc789 30459785
a7      0x4442434e 1145193294
s3      0x0 0
s5      0x0 0
s7      0x0 0

sbi.c
23      : "memory");
24      return ret;
25  }
26
27  struct sbiret sbi_set_timer(uint64_t stime_value)
> 28  {
29      struct sbiret ret;
30      ret = sbi_ecall(0x5449d4d5, 0x0, stime_value, 0, 0, 0, 0, 0);
31      return ret;
32  }
33  struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t reset_reason)
34  {

remote Thread 1.1 In: sbi_set_timer
(gdb) si
(gdb) si
sbi_set_timer (stime_value=720575940379279360) at sbi.c:28
(gdb)

```

图 5: 进入 sbi_set_timer

```

--Register group: general
zero    0x0 0
sp      0x87ffbe80 0x87ffbe80
tp      0x80048000 0x80048000
t1      0x0 0
fp      0x87ffbea0 0x87ffbea0
a0      0x0 0
a2      0x0 0
a4      0x0 0
a6      0x0 0
a8      0x0 0
s2      0x0 0
s4      0x0 0
s6      0x0 0
ra      0x80200224 0x80200224 <clock_set_next_event+52>
gp      0x0 0
t0      0x0 0
t2      0x0 0
s1      0x0 0
a1      0x0 0
a3      0x0 0
a5      0x0 0
a7      0x5449d4d5 1414090053
s3      0x0 0
s5      0x0 0
s7      0x0 0

clock.c
15      // 下一次时钟中断的时间点
16      uint64_t next = get_cycles() + TIMECLOCK;
17
18      // 使用 sbi_set_timer 来完成对下一次时钟中断的设置
19      sbi_set_timer(next);
> 20  {
21  void clock_init()
22  {
23      // 设置下一次时钟中断
24      clock_set_next_event();
25  }
26

remote Thread 1.1 In: clock_set_next_event
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
clock_set_next_event () at clock.c:20
(gdb)

```

图 6: 退出 sbi_set_timer 至 clock_set_next_event

```

Register group: general
zero 0x0 0 ra 0x8020117c 0x8020117c (trap_handler+464)
sp 0x87ffbba0 0x87ffbba0 bp 0x0 0x0
tp 0x80048000 0x80048000 t0 0x0 0
t1 0x0 0 t2 0x0 0
fp 0x87ffbed0 0x87ffbed0 s1 0x0 0
a0 0x0 0 a1 0x0 0
a2 0x0 0 a3 0x0 0
a4 0x0 0 a5 0x0 0
a6 0x0 0 a7 0x54494d45 1414090053
s2 0x0 0 s3 0x0 0
s4 0x0 0 s5 0x0 0
s6 0x0 0 s7 0x0 0

trap.c
122 case 0x1:
123 printk("Supervisor software interrupt\n");
124 break;
125 case 0x5:
126 clock_set_next_event();
> 127 do_timer();
128 printk("Supervisor timer interrupt\n");
129 break;
130 case 0x9:
131 printk("Supervisor external interrupt\n");
132 break;
133 case 0xD:

Remote Thread i.1 In: trap_handler L127 PC: 0x8020117c
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
clock_set_next_event () at clock.c:20
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
trap_handler (scause=9223372036854775813, sepc=2149583184) at trap.c:127
(gdb)

```

图 7: 退出 clock_set_next_event 至 trap_handler

```

Register group: general
zero 0x0 0 ra 0x80201180 0x80201180 (trap_handler+468)
sp 0x87ffbba0 0x87ffbba0 bp 0x0 0x0
tp 0x80048000 0x80048000 t0 0x0 0
t1 0x0 0 t2 0x0 0
fp 0x87ffbed0 0x87ffbed0 s1 0x0 0
a0 0x0 0 a1 0x0 0
a2 0x0 0 a3 0x0 0
a4 0x0 0 a5 0x0 0
a6 0x0 0 a7 0x54494d45 1414090053
s2 0x0 0 s3 0x0 0
s4 0x0 0 s5 0x0 0
s6 0x0 0 s7 0x0 0

trap.c
51 extern struct task_struct *current; // 指向当前运行线程的 task_struct
52
53 void do_timer()
> 54 // 1. 如果当前线程是 idle 线程或当前线程时间片耗尽则直接进行调度
55 // 2. 否则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
56 if(current == idle)
57 schedule();
58 current->counter--;
59 if((int)current->counter < 0)
60 current->counter = 0;
61 // printk("kernel current->counter = %d\n", current->counter);
62 if(current->counter == 0)

Remote Thread i.1 In: do_timer L53 PC: 0x80200f10
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
clock_set_next_event () at clock.c:20
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
trap_handler (scause=9223372036854775813, sepc=2149583184) at trap.c:127
(gdb) si
do_timer () at trap.c:53
(gdb)

```

图 8: 进入 do_timer


```

--Register group: general
zero    0x0    0
tp      0x87ffb640    0x80048800
t1      0x0    0
fp      0x87ffb670    0x87ffb670
a0      0x87ffb600    2281688896
a2      0x0    0
a4      0x0    0
a6      0x2    2
s2      0x0    0
s4      0x0    0
s6      0x0    0
ra      0x80200704    0x80200704 <switch_to+152>
gp      0x0    0x0
t0      0x0    0
t2      0x0    0
s1      0x0    0
a1      0x87ffd000    2281689888
a3      0x0    0
a5      0x2f    47
a7      0x4442434e    1145193294
s3      0x0    0
s5      0x0    0
s7      0x0    0

entry.S
106 # uint64_t sp;
107 # uint64_t s[12];
108 # };
109 .globl __switch_to
110 __switch_to:
> 111     addi a0, a0, 0x20
112     addi a1, a1, 0x20
113     sd ra, 0(a0)
114     sd sp, 8(a0)
115     sd s0, 16(a0)
116     sd s1, 24(a0)
117     sd s2, 32(a0)

Remote Thread 1.1 In: __switch_to
(gdb) n
(gdb) n
(gdb) n
(gdb) si
switch_to (next=0x0) at proc.c:65
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) si
__switch_to () at entry.S:111
(gdb)

```

图 11: 进入 __switch_to

```

--Register group: general
zero    0x0    0
tp      0x87ffb640    0x80048800
t1      0x0    0
fp      0x87ffb670    0x87ffb670
a0      0x87ffb620    2281688928
a2      0x0    0
a4      0x0    0
a6      0x2    2
s2      0x0    0
s4      0x0    0
s6      0x0    0
ra      0x80200704    0x80200704 <switch_to+152>
gp      0x0    0x0
t0      0x0    0
t2      0x0    0
s1      0x0    0
a1      0x87ffd020    2281689120
a3      0x0    0
a5      0x2f    47
a7      0x4442434e    1145193294
s3      0x0    0
s5      0x0    0
s7      0x0    0

entry.S
119     sd s4, 48(a0)
120     sd s5, 56(a0)
121     sd s6, 64(a0)
122     sd s7, 72(a0)
123     sd s8, 80(a0)
124     sd s9, 88(a0)
125     sd s10, 96(a0)
126     sd s11, 104(a0)
127     ld ra, 0(a1)
> 128     ld a0, 16(a1)
129     ld s1, 24(a1)
130

Remote Thread 1.1 In: __switch_to
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) si
(gdb) si
(gdb)

```

图 12: 恢复 next 进程的 ra, 但是由于由同一个函数接口调用, 所以 ra 的值不变

```

Register group: general
zero    0x0      0
sp       0x87ffde40  0x87ffde40
tp       0x80048000  0x80048000
t1       0x0      0
fp       0x87ffde70  0x87ffde70
a0       0x87ffb020  2281680928
a2       0x0      0
a4       0x0      0
a6       0x2      2
s2       0x0      0
s4       0x0      0
s6       0x0      0
ra       0x80200704  0x80200704 <switch_to+152>
gp       0x0      0x0
t0       0x0      0
t2       0x0      0
s1       0x0      0
a1       0x87ffd020  2281689120
a3       0x0      0
a5       0x2f     47
a7       0x4442434e 1145193294
s3       0x0      0
s5       0x0      0
s7       0x0      0

proc.c
65 void switch_to(struct task_struct *next) {
66     if(current == next) return;
67     struct task_struct *prev = current;
68     current = next;
69     printf("switch to [%d] = %d PRIORITY = %d COUNTER = %d\n", current->pid, current->priority, current->counter);
> 70     __switch_to(prev, next);
71 }
72
73 // task_init 的时候随机为各个线程赋予了优先级
74 // 调度时选择 counter 最大的线程运行
75 // 如果所有线程 counter 都为 0, 则令所有线程 counter = priority
76 // 即优先级越高, 运行的时间越长, 且越先运行

Remote Thread 1.1 In: switch_to
(gdb) si
__switch_to () at entry.S:129
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) si
0x80000000080200704 in __switch_to (next=0x87ffe000) at proc.c:70
(gdb)

```

图 13: 退出 __switch_to 至 switch_to

```

Register group: general
zero    0x0      0
sp       0x87ffde70  0x87ffde70
tp       0x80048000  0x80048000
t1       0x0      0
fp       0x87ffde90  0x87ffde90
a0       0x87ffb020  2281680928
a2       0x0      0
a4       0x0      0
a6       0x2      2
s2       0x0      0
s4       0x0      0
s6       0x0      0
ra       0x80200908  0x80200908 <schedule+492>
gp       0x0      0x0
t0       0x0      0
t2       0x0      0
s1       0x0      0
a1       0x87ffd020  2281689120
a3       0x0      0
a5       0x2f     47
a7       0x4442434e 1145193294
s3       0x0      0
s5       0x0      0
s7       0x0      0

proc.c
95     next = task[i];
96     }
97     }
98     }
99     switch_to(next);
> 100    return;
101 }
102
103 #if TEST_SCHED
104 #define MAX_OUTPUT ((NR_TASKS - 1) * 10)
105 char tasks_output[MAX_OUTPUT];
106 int tasks_output_index = 0;

Remote Thread 1.1 In: schedule
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) si
0x80000000080200704 in __switch_to (next=0x87ffe000) at proc.c:70
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
schedule () at proc.c:100
(gdb)

```

图 14: 退出 switch_to 至 schedule

```

Register group: general
zero    0x0 0
sp      0x87ffdea0 0x87ffdea0
tp      0x80048000 0x80048000
t1      0x0 0
fp      0x87ffdea0 0x87ffdea0
a0      0x87ffb020 228168928
a2      0x0 0
a4      0x0 0
a6      0x2 2
s2      0x0 0
s4      0x0 0
s6      0x0 0
ra      0x80200f98 0x80200f98 <do_timer+136>
gp      0x0 0
t0      0x0 0
t2      0x0 0
s1      0x0 0
a1      0x87ffdb20 2281689120
a3      0x0 0
a5      0x2f 47
a7      0x4442434e 1145193294
s3      0x0 0
s5      0x0 0
s7      0x0 0

trap.c
59  if((int)current->counter < 0)
60      current->counter = 0;
61  // printk("kernel current->counter = %d\n", current->counter);
62  if(current->counter == 0)
63      schedule();
> 64
65  void trap_handler(uint64_t scause, uint64_t sepc) {
66
67      if ((scause >> 63) == 0) {
68          // Handle exceptions
69          switch (scause) {
70              case 0x0:

```

Remote Thread 1.1 In: do_timer L64 PC: 0x80200f98

```

(gdb) si
(gdb) si
0x00000000200704 in switch_to (next=0x87ffe000) at proc.c:70
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
schedule () at proc.c:100
(gdb) si
(gdb) si
(gdb) si
(gdb) si
do_timer () at trap.c:64
(gdb)

```

图 15: 退出 schedule 至 do_timer

```

Register group: general
zero    0x0 0
sp      0x87ffdea0 0x87ffdea0
tp      0x80048000 0x80048000
t1      0x0 0
fp      0x87ffdea0 0x87ffdea0
a0      0x87ffb020 228168928
a2      0x0 0
a4      0x0 0
a6      0x2 2
s2      0x0 0
s4      0x0 0
s6      0x0 0
ra      0x80201180 0x80201180 <trap_handler+468>
gp      0x0 0
t0      0x0 0
t2      0x0 0
s1      0x87ffdb20 2281689120
a3      0x0 0
a5      0x2f 47
a7      0x4442434e 1145193294
s3      0x0 0
s5      0x0 0
s7      0x0 0

trap.c
123      printk("Supervisor software interrupt\n");
124      break;
125      case 0x5:
126          clock_set_next_event();
127          do_timer();
> 128      printk("Supervisor timer interrupt\n");
129      break;
130      case 0x9:
131          printk("Supervisor external interrupt\n");
132          break;
133      case 0xD:
134          printk("Counter-overflow interrupt\n");

```

Remote Thread 1.1 In: trap_handler L128 PC: 0x80201180

```

(gdb) si
(gdb) si
schedule () at proc.c:100
(gdb) si
(gdb) si
(gdb) si
do_timer () at trap.c:64
(gdb) si
(gdb) si
(gdb) si
(gdb) si
trap_handler (scause=9223372036854775813, sepc=2149583204) at trap.c:128
(gdb)

```

图 16: 退出 do_timer 至 trap_handler

5.4 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。

我修改了 `_traps` 的代码, 添加了 CFI 的调试信息, 使得 backtrace 可以正确的显示函数调用栈 (由于 fp 指针可以优化, riscv 的 backtrace 不依据 fp 指针, 而是依据 CFI 信息). 代码如下:

```
1  _traps:
2
3  .cfi_startproc
4  .cfi_def_cfa sp, 0
5  sd x2, -240(sp)
6  addi sp, sp, -256
7  .cfi_def_cfa sp, 256
8  sd x1, 0(sp)
9  csrr x1, sepc
10 sd x1, 248(sp)
11 sd x3, 24(sp)
12 sd x4, 32(sp)
13 sd x5, 40(sp)
14 sd x6, 48(sp)
15 sd x7, 56(sp)
16 sd x8, 240(sp)
17 addi x8, sp, 256
18 .cfi_offset x1, -8
19 .cfi_offset x8, -16
20 sd x9, 72(sp)
21 sd x10, 80(sp)
22 ...
23 ld x30, 64(sp)
24 ld x31, 8(sp)
25 ld x2, 16(sp)
26
```

```
27
28     sret
29     .cfi_endproc
```

然后使用 backtrace 跟踪得到栈帧布局如下:

idle线程

```
#0  __switch_to () at entry.S:118
#1  0x0000000080200708 in switch_to (next=0x87ffd000) at proc.c:70
#2  0x000000008020090c in schedule () at proc.c:99
#3  0x0000000080200f44 in do_timer () at trap.c:57
#4  0x0000000080201180 in trap_handler (scause=9223372036854775813,
sepc=2149585456) at trap.c:127
#5  0x00000000802000c0 in _traps () at entry.S:50
#6  0x0000000080201230 in test () at test.c:4
#7  0x0000000080201204 in start_kernel () at main.c:9
#8  0x0000000080200028 in _stext () at head.S:20
```

第一个执行的线程

```
#0  __switch_to () at entry.S:118
#1  0x0000000080200708 in switch_to (next=0x87ffe000) at proc.c:70
#2  0x000000008020090c in schedule () at proc.c:99
#3  0x0000000080200f98 in do_timer () at trap.c:63
#4  0x0000000080201180 in trap_handler (scause=9223372036854775813,
sepc=2149583192) at trap.c:127
#5  0x00000000802000c0 in _traps () at entry.S:50
#6  0x0000000080200958 in dummy () at proc.c:116
```

待转移线程和其余线程栈为空

```

(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x000000000001000 in ?? ()
(gdb) b __switch_to
Breakpoint 1 at 0x80200150: file entry.S, line 118.
(gdb) c
Continuing.

Breakpoint 1, __switch_to () at entry.S:118
118      addi a0, a0, 0x20
(gdb) backtrace
#0  __switch_to () at entry.S:118
#1  0x0000000000200708 in switch_to (next=0x87ffd000) at proc.c:70
#2  0x000000000020090c in schedule () at proc.c:99
#3  0x0000000000200f44 in do_timer () at trap.c:57
#4  0x0000000000201180 in trap_handler (scause=9223372036854775813, sepc=2149585456) at trap.c:127
#5  0x00000000002000c0 in _traps () at entry.S:50
#6  0x0000000000201230 in test () at test.c:4
#7  0x0000000000201204 in start_kernel () at main.c:9
#8  0x0000000000200028 in _stext () at head.S:20
Backtrace stopped: frame did not save the PC
(gdb) █

```

图 19: idle 线程的 backtrace

```

Backtrace stopped: frame did not save the PC
(gdb) c
Continuing.

Breakpoint 1, __switch_to () at entry.S:118
118      addi a0, a0, 0x20
(gdb) backtrace
#0  __switch_to () at entry.S:118
#1  0x0000000000200708 in switch_to (next=0x87ffe000) at proc.c:70
#2  0x000000000020090c in schedule () at proc.c:99
#3  0x0000000000200f98 in do_timer () at trap.c:63
#4  0x0000000000201180 in trap_handler (scause=9223372036854775813, sepc=2149583192) at trap.c:127
#5  0x00000000002000c0 in _traps () at entry.S:50
#6  0x0000000000200958 in dummy () at proc.c:116

```

图 20: 第一个执行的线程的 backtrace