

Operating system

Lab 2



Author: 吴杭

Date: 2024/11/17

Autumn-Fall 2024-2025 Semester

Table of Contents

Chapter 1: 实验流程	3
1.1) 初始化工程	3
1.2) 线程调度功能实现	4
1.2.1) 线程初始化	4
1.2.2) dummy 与__dummy 的实现	5
1.2.3) 实现线程的切换	5
1.2.4) 实现调度入口函数	6
1.2.5) 线程调度算法实现	6
Chapter 2: 思考题	7
2.1) 1.	7
2.2) 2.	7
2.3) 3.	7
2.4) 4.	10
Chapter 3: 心得体会	10
3.1) 遇到的问题	10
3.2) 心得体会	10
Declaration	11

Chapter 1: 实验流程

1.1) 初始化工程

从仓库中 clone 相关的代码到本地，由于 lab2 是在 lab1 的基础上开发的，所以我们需要把 lab1 已经完成的代码和 lab2 合并。最后得到结果如下

```
→ lab2 git:(main) X tree .
.
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       ├── include
│       │   ├── defs.h
│       │   ├── mm.h
│       │   ├── proc.h
│       │   └── sbi.h
│       └── kernel
│           ├── Makefile
│           ├── clock.c
│           ├── entry.S
│           ├── head.S
│           ├── mm.c
│           ├── proc.c
│           ├── sbi.c
│           ├── trap.c
│           └── vmlinux.lds
├── fw_jump.bin
├── include
│   ├── printk.h
│   ├── stddef.h
│   ├── stdint.h
│   ├── stdlib.h
│   └── string.h
├── init
│   ├── Makefile
│   ├── main.c
│   └── test.c
└── lib
    ├── Makefile
    ├── printk.c
    ├── rand.c
    └── string.c

8 directories, 28 files
```

Figure 1: 工程结构

然后根据实验文档，为了让 `kalloc` 能正常分配内存，我们需要在 `defs.h` 中添加相应的宏。然后需要在 `_start` 的适当位置调用 `mm_init`，这里我们需要在第一次设置时钟中断之前之前调用 `mm_init`（同样 `task init` 也是在时钟中断之前设置）。

```
# initialize the memory management
call mm_init

# init the tasks before starting the kernel
call task_init
```

1.2) 线程调度功能实现

1.2.1) 线程初始化

线程初始化的时候，需要给每个线程都分配一个 4KiB 的物理页。然后需要初始化一些用于记录线程运行信息的数据结构。而第一个我们需要初始化的特殊线程就是 `idle` 线程（也就是我们运行的操作系统本身）。实现的思路和实验框架给出的注释一致，当然下面展示的代码的注释也已经展示了设计的思路。具体的实现代码如下

```
// allocate a physical page for idle
idle = (struct task_struct*)kalloc();

// set the state as TASK_RUNNING
idle->state = TASK_RUNNING;

// set the counter and priority
idle->counter = idle->priority = 0;

// set the pid as 0
idle->pid = 0;

// set the current and task[0]
current = idle;
task[0] = idle;
```

接下来我们需要初始化其他的线程，而相对于 `idle` 线程不同的是，其他的线程需要参与调度，所以我们需要为他们分配一个随机的优先级。然后由于调度过程中，需要保存 PC 返回的地址，所以我们需要在他们的 `thread_struct` 中设置好 `ra` 寄存器。而对于还没有开始运行过的进程，本实验中我们把他们的 `ra` 设置为一个特殊的地址 `__dummy`，当他们被调度的时候，就会从 `__dummy` 开始运行，接着进入他们各自的程序。

为了能够分离地管理这些线程的内存分配资源，我们需要为他们独立分配栈空间，所以我们也需要在他们的 `thread_struct` 中记录下他们的栈空间的高地址。相应的代码如下：

```
for (int i = 1; i < NR_TASKS; i++) {
    struct task_struct* new_task = (struct task_struct*)kalloc();
    new_task->state = TASK_RUNNING;
    new_task->pid = i;

    // set counter and priority using rand
    new_task->counter = 0;
    new_task->priority = rand() % (PRIORITY_MAX-PRIORITY_MIN+1) + PRIORITY_MIN;

    // set the ra and sp
    new_task->thread.ra = (uint64_t)__dummy;
```

```
new_task->thread.sp = (uint64_t)new_task + PGSIZE; // notice new_task is also an
address of the struct (the bottom of the PAGE)
```

```
task[i] = new_task;
}
```

上面的代码中要注意的是，`new_task` 是指向这个 `task` 的指针，同时也是这个 `task` 被分配到的内存空间的低地址，而整个 `task` 被分配到的空间的大小为 `PGSIZE` (也就是 4KiB)，所以每个 `task` 的 `sp` 的值就是 `(uint_64t)new_task + PGSIZE`。

1.2.2) `dummy` 与 `__dummy` 的实现

本实验中，所有的 `task` (`idle` 除外) 都运行同一段代码 `dummy`。

上面我们提到了，线程第一次调度的时候，需要提供一个特殊的地址 `__dummy`，根据实验文档中的设计，我们设计这个函数的代码如下

```
__dummy:
    la t0, dummy # load the address of dummy into the t0 register
    csrw sepc, t0
    sret # the program will return to the address of dummy
```

1.2.3) 实现线程的切换

通过一个 `switch_to` 函数，来实现线程的切换。这个函数接受一个参数，为指向下一个要切换的线程的指针 `next`，然后判断 `next` 和当前的线程的指针 `current` 是否为同一个线程 (通过比较两者的 `pid` 得到)，如果是同一个，那么就不需要调度，如果不是同一个，那么就调用 `__switch_to` 函数进行调度。

```
void switch_to(struct task_struct *next) {
    // check if the current task is the same as the next one using their unique
    // pid
    if (next->pid != current->pid) {
        // if they are not the same, call __switch_to
        printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n", next->pid,
            next->priority, next->counter);
        // save the current task to temp_task and switch to the next task
        temp_task = current;
        current = next;
        __switch_to(temp_task, next);
    }
}
```

`__switch_to` 函数的任务就是储存当前的运行的程序的上下文 `ra`, `sp`, `s0-s11` 到之前用来储存线程信息的数据结构 (`task_struct` 中的 `thread_struct`) 中，然后再把下一个要切换的程序的上下文 load 到 `ra`, `sp`, `s0-s11` 这些寄存器中。这里要注意的是我们传给 `__switch_to` 的是指向 `task_struct` 的指针，而 `thread_struct` 储存在 `task_struct` 中的起始位置在第 32 个 byte，所以我们 store 和 load 的时候需要在 `a0` 的第 32 个 byte 开始。

代码如下：

```

__switch_to:
    # save states to prev process
    # a0 is the base address of task_struct prev, and the content of thread starts
    from 32(a0)
    sd ra, 32(a0)
    ...

    ld ra, 32(a1)
    ...
    # restore state from next process
    ret

```

1.2.4) 实现调度入口函数

设计 `do_timer` 函数，其在时钟中断处理函数中调用，负责执行调度逻辑。

- 当前线程为 `idle` 或当前线程时间片已耗尽，则直接进行调度；
- 否则当前线程仍有时间片，将其剩余时间减 1。
 - 若剩余时间仍然大于 0，则不进行调度直接返回；
 - 否则当前线程的执行时间结束，进行调度。

根据以上逻辑，实现代码如下：

```

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

```

1.2.5) 线程调度算法实现

在 `task_init` 函数中，内核为各线程分配了随机的优先级。调度算法每次选择剩余时间片最大的线程执行，若所有线程时间片均已耗尽，则重新将时间片初始化为优先级并再次进行调度。

根据线程调度算法逻辑，编写工具函数 `get_next_task` 获取下一个要执行的线程：

```

// 选择下一个要运行的线程
static int get_next_task() {
    while (true) {
        int next = 0; // 下一个要运行的线程
        int counter = 0; // next 线程的 counter
        for (int i = 1; i < NR_TASKS; i++) {
            if (task[i] != NULL && task[i]->counter > counter) {
                // 选择 counter 最大的线程
            }
        }
    }
}

```

```

        counter = task[i]->counter;
        next = i;
    }
}
if (counter > 0) { // 找到了 counter > 0 的线程
    return next;
}
// 所有线程的时间片都已耗尽
for (int i = 1; i < NR_TASKS; i++) {
    if (task[i] != NULL) {
        // 重新设置时间片
        task[i]->counter = task[i]->priority;
        printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]->pid,
            task[i]->priority, task[i]->counter);
    }
}
}
}
}
}

```

在 `get_next_task` 函数与 `switch_to` 函数的基础上，实现调度函数 `schedule`：

```
void schedule() { switch_to(task[get_next_task()]); }
```

Chapter 2: 思考题

2.1) 1.

RISC-V 的寄存器根据调用约定被分成 caller-saved registers 和 callee-saved registers。其中 caller-saved registers 包括 `t0-t6`, `a0-a7`, 而 callee-saved registers 包括 `s0-s11`, `sp`, `ra`。在线程切换的时候，该线程只需要保存 caller-saved registers 即可。

2.2) 2.

`mm_init` 这个函数的目的是开辟内存，它是通过调用 `kfreerange` 来实现的，`kfreerange` 中则是循环地调用 `kfree` 去每次开辟一个 `PGSIZE` 的空间，将其加入到一个 `kmem.freelist` 的链表中，表示空闲的能被调用的 page。这样在循环结束就把整块内存都分成了多个大小为 `PGSIZE` 的 pages，作为管理内存的基本单元。

更加具体地，`kfree` 中先将 `addr` 向上对齐到整数个 `PGSIZE` 的大小的位置，然后将这段内存都赋值为 0，然后把这块内存用一个 `struct run *` 去管理，然后加入到 `kmem.freelist` 的头部。

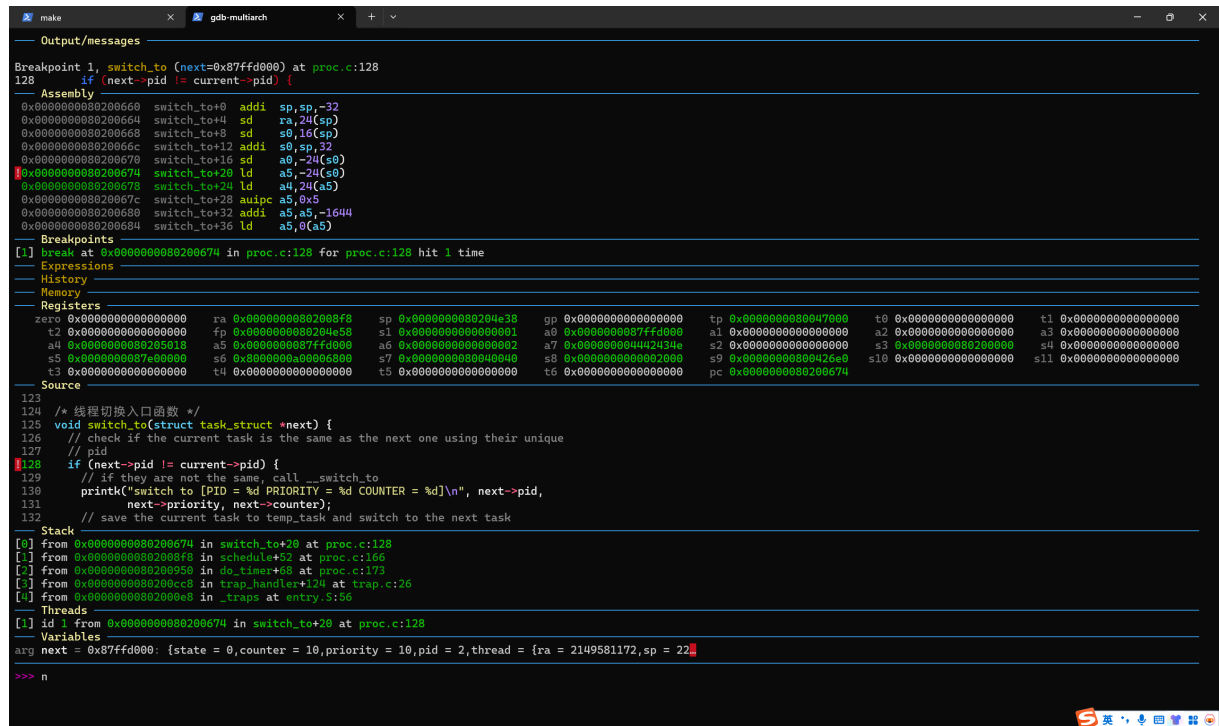
对于 `kalloc`，则是取出 `kmem.freelist` 中的一块内存，重新赋值为 0 之后，返回给进程使用。从这个 `kmem.freelist` 的管理可以看出，当给一个进程分配内存的时候，是从内存地址大的位置开始逐页分配的。

2.3) 3.

在之后的线程调度 `__switch_to` 过程中，`ra` 保存的函数返回点即为程序运行到 `__switch_to` 入口的 PC 值。

完整的线程切换过程如下:

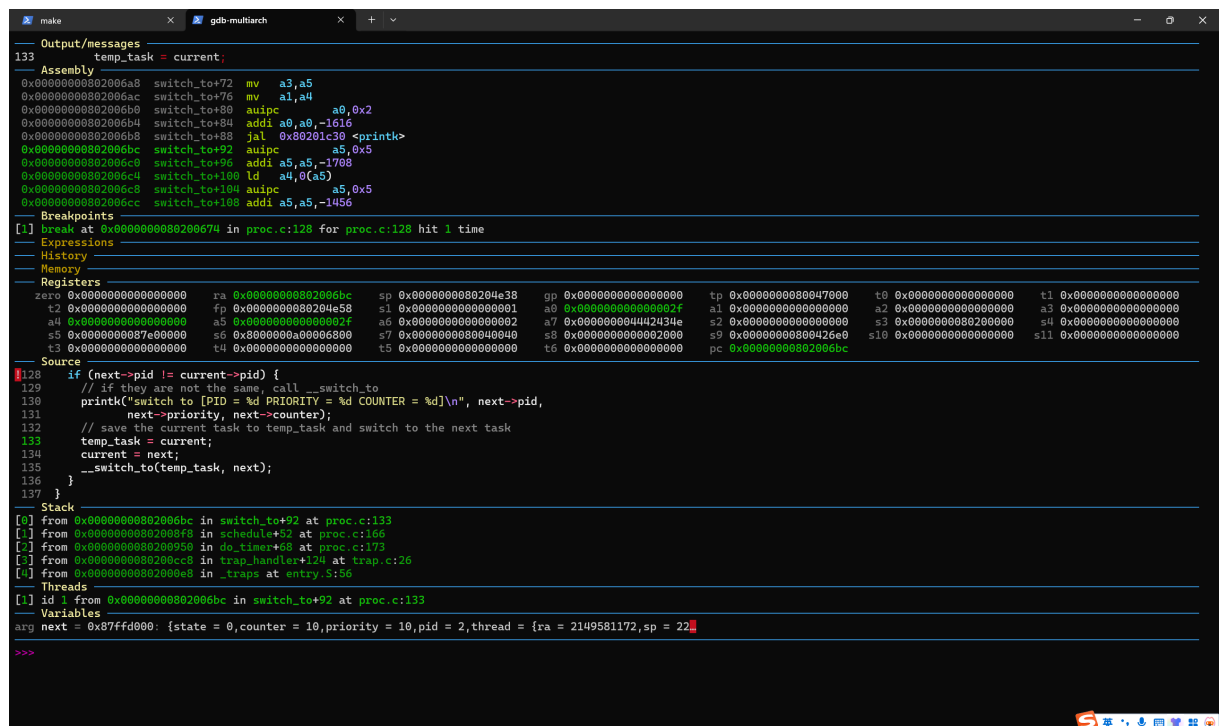
程序刚进入 `switch_to` 函数时, `ra` 的值在下图中可以看到



```
make x gdb-multiarch
Output/messages
Breakpoint 1, switch_to (next=0x87ffd000) at proc.c:128
128 if (next->pid != current->pid) {
Assembly
0x00000000200660 switch_to+0 addi sp,sp,-32
0x00000000200664 switch_to+4 sd ra,24(sp)
0x00000000200668 switch_to+8 sd s0,16(sp)
0x0000000020066c switch_to+12 addi s0,sp,32
0x00000000200670 switch_to+16 sd a0,-24(s0)
0x00000000200674 switch_to+20 ld a5,-24(s0)
0x00000000200678 switch_to+24 ld a4,24(a5)
0x0000000020067c switch_to+28 auipc a5,0x5
0x00000000200680 switch_to+32 addi a5,a5,-1644
0x00000000200684 switch_to+36 ld a5,0(a5)
Breakpoints
[1] break at 0x00000000200674 in proc.c:128 for proc.c:128 hit 1 time
Expressions
History
Memory
Registers
zero 0x0000000000000000 ra 0x000000002006f8 sp 0x00000000204e38 gp 0x0000000000000000 tp 0x000000008047600 t0 0x0000000000000000 t1 0x0000000000000000
t2 0x0000000000000000 fp 0x00000000204e58 s1 0x0000000000000001 a0 0x0000000087ffd000 a1 0x0000000000000000 a2 0x0000000000000000 a3 0x0000000000000000
a4 0x00000000205918 a5 0x0000000087ffd000 a6 0x0000000000000002 a7 0x000000004442434e a8 0x0000000000000000 a9 0x0000000000000000 a10 0x0000000000000000
s0 0x000000007e000000 s1 0x00000000a0000000 s2 0x0000000000000000 s3 0x0000000000000000 s4 0x0000000000000000 s5 0x0000000000000000 s6 0x00000000a0000000 s7 0x0000000000000000 s8 0x0000000000000000 s9 0x0000000000000000 s10 0x0000000000000000 s11 0x0000000000000000
t3 0x0000000000000000 t4 0x0000000000000000 t5 0x0000000000000000 t6 0x0000000000000000 pc 0x00000000200674
Source
123
124 /* 线程切换入口函数 */
125 void switch_to(struct task_struct *next) {
126 // check if the current task is the same as the next one using their unique
127 // pid
128 if (next->pid != current->pid) {
129 // if they are not the same, call __switch_to
130 printk("switch to [PID = %d PRIORITY = %d] COUNTER = %d\n", next->pid,
131 next->priority, next->counter);
132 // save the current task to temp_task and switch to the next task
133 Stack
[0] from 0x00000000200674 in switch_to+20 at proc.c:128
[1] from 0x000000002006f8 in schedule+52 at proc.c:166
[2] from 0x00000000200950 in do_timer+68 at proc.c:173
[3] from 0x00000000200cc8 in trap_handler+124 at trap.c:26
[4] from 0x000000002000e8 in _traps at entry.S:56
Threads
[1] id 1 from 0x00000000200674 in switch_to+20 at proc.c:128
Variables
arg next = 0x87ffd000: {state = 0, counter = 10, priority = 10, pid = 2, thread = {ra = 2149581172, sp = 22}}
>>> n
```

Figure 2: 进入 `switch_to`

程序继续运行, 可以看到当运行到 `proc.c` 的 133 行的时候 `ra` 再次发生变化



```
make x gdb-multiarch
Output/messages
133 temp_task = current;
Assembly
0x000000002006a8 switch_to+72 mv a3,a5
0x000000002006ac switch_to+76 mv a1,a4
0x000000002006b0 switch_to+80 auipc a0,0x2
0x000000002006b4 switch_to+84 addi a0,a0,-1616
0x000000002006b8 switch_to+88 jal 0x00201c30 <printk>
0x000000002006bc switch_to+92 auipc a5,0x5
0x000000002006c0 switch_to+96 addi a5,a5,-1708
0x000000002006c4 switch_to+100 ld a4,0(a5)
0x000000002006c8 switch_to+104 auipc a5,0x5
0x000000002006cc switch_to+108 addi a5,a5,-1456
Breakpoints
[1] break at 0x00000000200674 in proc.c:128 for proc.c:128 hit 1 time
Expressions
History
Memory
Registers
zero 0x0000000000000000 ra 0x000000002006bc sp 0x00000000204e38 gp 0x0000000000000000 tp 0x000000008047600 t0 0x0000000000000000 t1 0x0000000000000000
t2 0x0000000000000000 fp 0x00000000204e58 s1 0x0000000000000001 a0 0x000000000000002f a1 0x0000000000000000 a2 0x0000000000000000 a3 0x0000000000000000
a4 0x0000000000000000 a5 0x000000000000002f a6 0x0000000000000002 a7 0x000000004442434e a8 0x0000000000000000 a9 0x0000000000000000 a10 0x0000000000000000
s0 0x000000007e000000 s1 0x00000000a0000000 s2 0x0000000000000000 s3 0x0000000000000000 s4 0x0000000000000000 s5 0x0000000000000000 s6 0x00000000a0000000 s7 0x0000000000000000 s8 0x0000000000000000 s9 0x0000000000000000 s10 0x0000000000000000 s11 0x0000000000000000
t3 0x0000000000000000 t4 0x0000000000000000 t5 0x0000000000000000 t6 0x0000000000000000 pc 0x000000002006bc
Source
128 if (next->pid != current->pid) {
129 // if they are not the same, call __switch_to
130 printk("switch to [PID = %d PRIORITY = %d] COUNTER = %d\n", next->pid,
131 next->priority, next->counter);
132 // save the current task to temp_task and switch to the next task
133 temp_task = current;
134 current = next;
135 __switch_to(temp_task, next);
136 }
137 }
Stack
[0] from 0x000000002006bc in switch_to+92 at proc.c:133
[1] from 0x000000002006f8 in schedule+52 at proc.c:166
[2] from 0x00000000200950 in do_timer+68 at proc.c:173
[3] from 0x00000000200cc8 in trap_handler+124 at trap.c:26
[4] from 0x000000002000e8 in _traps at entry.S:56
Threads
[1] id 1 from 0x000000002006bc in switch_to+92 at proc.c:133
Variables
arg next = 0x87ffd000: {state = 0, counter = 10, priority = 10, pid = 2, thread = {ra = 2149581172, sp = 22}}
>>>
```

Figure 3: 运行至 133 行

程序继续运行，可以看到当运行到__switch_to 的入口的时候，ra 再次发生变化，这时就会把 ra 存入到当前备切换线程的栈中，也就是当前线程的函数返回点。

```

-- Output/messages
__switch_to () at entry.S:113
113      sd ra, 32(a0)
-- Assembly
0x00000000200184 ? sd      ra,32(a0)
0x00000000200188 ? sd      sp,40(a0)
0x0000000020018c ? sd      s0,48(a0)
0x00000000200190 ? sd      s1,56(a0)
0x00000000200194 ? sd      s2,64(a0)
0x00000000200198 ? sd      s3,72(a0)
0x0000000020019c ? sd      s4,80(a0)
0x000000002001a0 ? sd      s5,88(a0)
0x000000002001a4 ? sd      s6,96(a0)
0x000000002001a8 ? sd      s7,104(a0)
-- Breakpoints
[1] break at 0x00000000200674 in proc.c:128 for proc.c:128 hit 1 time
-- Expressions
-- History
-- Memory
-- Registers
zero 0x0000000000000000 ra 0x000000002006fc sp 0x00000000204e38 gp 0x0000000000000000 tp 0x0000000000007000 t0 0x0000000000000000 t1 0x0000000000000000
t2 0x0000000000000000 fp 0x00000000204e58 s1 0x0000000000000001 a0 0x0000000087fff000 a1 0x0000000087ffd000 a2 0x0000000000000000 a3 0x0000000000000000
a4 0x0000000087fff000 a5 0x0000000087fff000 a6 0x0000000000000002 a7 0x000000004442134e s2 0x0000000000000000 s3 0x0000000020000000 s4 0x0000000000000000
s5 0x0000000087f00000 s6 0x0000000a00006800 s7 0x0000000000004040 s8 0x0000000000002000 s9 0x0000000000004260 s10 0x0000000000000000 s11 0x0000000000000000
t3 0x0000000000000000 t4 0x0000000000000000 t5 0x0000000000000000 t6 0x0000000000000000 pc 0x00000000200184
-- Source
108 __switch_to:
109
110 # TODO: might need to be fixed
111 # save states to prev process
112 # a0 is the base address of task_struct prev, and the content of thread starts from 32(a0)
113 sd ra, 32(a0)
114 sd sp, 40(a0)
115 sd s0, 48(a0)
116 sd s1, 56(a0)
117 sd s2, 64(a0)
-- Stack
[0] from 0x00000000200184 in __switch_to at entry.S:113
[1] from 0x000000002006fc in __switch_to+156 at proc.c:135
[2] from 0x000000002006f8 in schedule+52 at proc.c:166
[3] from 0x00000000200950 in do_timer+68 at proc.c:173
[4] from 0x00000000200cc8 in trap_handler+124 at trap.c:26
[5] from 0x000000002000e8 in _traps at entry.S:56
-- Threads
[1] id 1 from 0x00000000200184 in __switch_to at entry.S:113
-- Variables
>>> n

```

Figure 4: 运行至__switch_to 入口

程序继续运行，可以看到当运行到 ld ra, 32(a1)的时候，ra 被赋值为了将要切换到的 next 线程的函数返回点。

```

-- Output/messages
129      ld sp, 40(a1)
-- Assembly
0x000000002001c0 ? ld      sp,40(a1)
0x000000002001c4 ? ld      s0,48(a1)
0x000000002001c8 ? ld      s1,56(a1)
0x000000002001cc ? ld      s2,64(a1)
0x000000002001d0 ? ld      s3,72(a1)
0x000000002001d4 ? ld      s4,80(a1)
0x000000002001d8 ? ld      s5,88(a1)
0x000000002001dc ? ld      s6,96(a1)
0x000000002001e0 ? ld      s7,104(a1)
0x000000002001e4 ? ld      s8,112(a1)
-- Breakpoints
[1] break at 0x00000000200674 in proc.c:128 for proc.c:128 hit 1 time
-- Expressions
-- History
-- Memory
-- Registers
zero 0x0000000000000000 ra 0x00000000200174 sp 0x00000000204e38 gp 0x0000000000000000 tp 0x0000000000007000 t0 0x0000000000000000 t1 0x0000000000000000
t2 0x0000000000000000 fp 0x00000000204e58 s1 0x0000000000000001 a0 0x0000000087fff000 a1 0x0000000087ffd000 a2 0x0000000000000000 a3 0x0000000000000000
a4 0x0000000087fff000 a5 0x0000000087fff000 a6 0x0000000000000002 a7 0x000000004442134e s2 0x0000000000000000 s3 0x0000000020000000 s4 0x0000000000000000
s5 0x0000000087f00000 s6 0x0000000a00006800 s7 0x0000000000004040 s8 0x0000000000002000 s9 0x0000000000004260 s10 0x0000000000000000 s11 0x0000000000000000
t3 0x0000000000000000 t4 0x0000000000000000 t5 0x0000000000000000 t6 0x0000000000000000 pc 0x000000002001c0
-- Source
124 sd s9, 120(a0)
125 sd s10, 128(a0)
126 sd s11, 136(a0)
127
128 ld ra, 32(a1)
129 ld sp, 40(a1)
130 ld s0, 48(a1)
131 ld s1, 56(a1)
132 ld s2, 64(a1)
133 ld s3, 72(a1)
-- Stack
[0] from 0x000000002001c0 in __switch_to at entry.S:129
[1] from 0x00000000200174 in _traps at entry.S:101
-- Threads
[1] id 1 from 0x000000002001c0 in __switch_to at entry.S:129
-- Variables
>>> |

```

Figure 5: 运行至 ld ra, 32(a1)入口

2.4) 4.

```
+-----+
| .....|
| (No function calls or data on these threads' stack)|
+-----+
| Process 2: Empty Stack|
| (No function calls or data on this thread's stack)|
| +-----+ <-sp|
+-----+
| Process 1:|
| +-----+|
| | _traps|
| +-----+|
| | trap_handler|
| +-----+|
| | do_timer|
| +-----+|
| | schedule|
| +-----+|
| | switch_to|
| +-----+ <-sp|
+-----+
| boot_stack:|
| +-----+|
| | _traps|
| +-----+|
| | trap_handler|
| +-----+|
| | do_timer|
| +-----+|
| | schedule|
| +-----+|
| | switch_to|
| +-----+ <-sp|
+-----+
```

Chpater 3: 心得体会

3.1) 遇到的问题

遇到的主要问题是 `mm_init` 的调用位置，一开始我把 `mm_init` 放在设置下一次时钟中断之后调用，但是这样出现的问题是，如果时钟中断的时间间隔太小，那么 `mm_init` 就会在发生时钟中断的时候仍然没有被运行完，这会出现问题，因为处理时钟中断的逻辑中，会调用到 `do_timer`，如果这时候各个线程并没有被分配好的话，就会出现问题。所以要把 `mm_init` 和 `task_init` 放在设置时钟中断之前调用。

3.2) 心得体会

这个实验带领我们实现了简易的线程调度逻辑，加深了我对课上提到的时间片轮转算法的理解，我认为是一个框架非常完善，内容非常充实的实验。尤其是在考虑 `mm_init` 的过

程中，让我意识到处理中断的过程和内核的运行过程是分割的，这让我对计算机的底层逻辑有了更深刻的理解。

Declaration

We hereby declare that all the work done in this lab 2 is of our independent effort.