Lab 3: RV64 内核线程调度

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 实验步骤和原理分析

3.1 代码结构

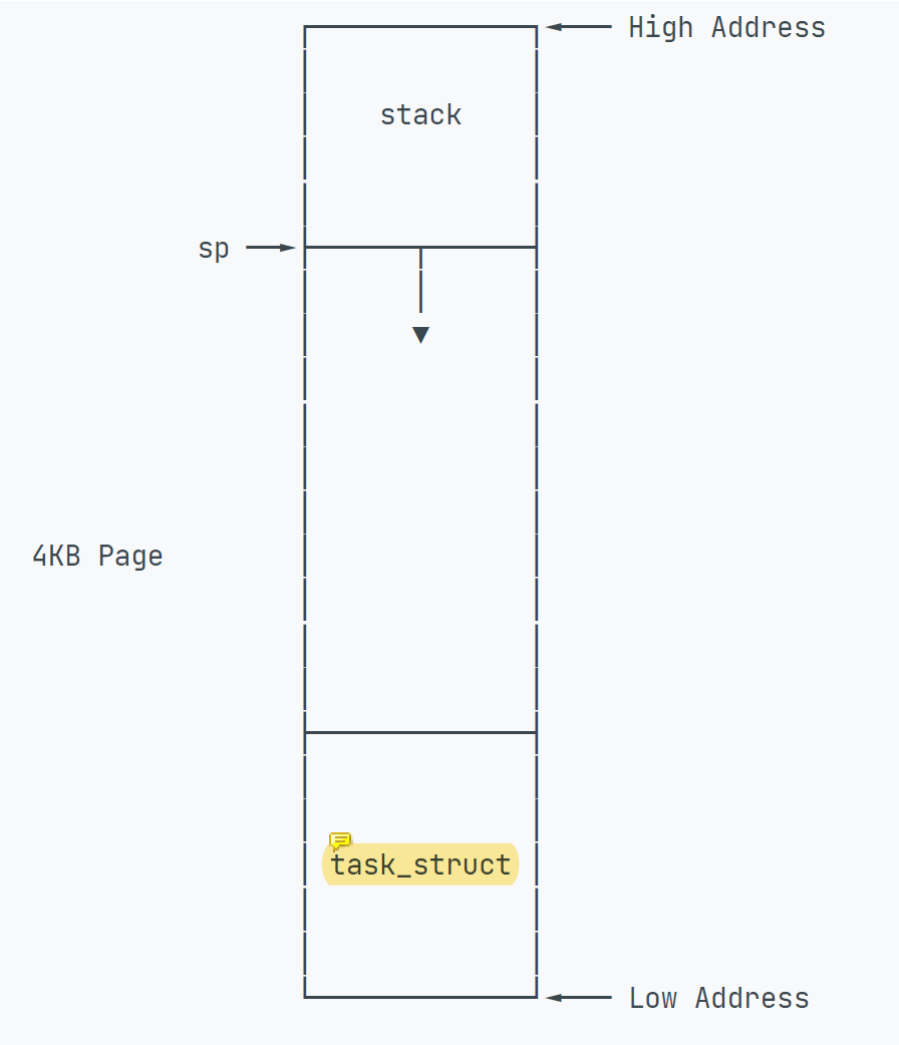
从 repo 同步以下代码: rand.h/rand.c , string.h/string.c , mm.h/mm.c 。并按照以下步骤将这些文件正确放置。其中 mm.h/mm.c 提供了简单的物理内存管理接口,rand.h/rand.c 提供了 rand() 接口用以提供伪随机数序列, string.h/string.c 提供了memset 接口用以初始化一段内存空间。

在 lab3 中需要同学需要添加并修改 arch/riscv/include/proc.h和arch/riscv/kernel/proc.c 两个文件。

```
├── arch
│ └── riscv
│ ├── include
│ │ └── mm.h
│ └── kernel
│ └── mm.c
├── include
│ ├── rand.h
│ └── string.h
└── lib
├── rand.c
└── string.c
```

3.2 线程初始化

在初始化部分，需要为每个线程分配一个 4KB 的物理页, 我们将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。具体内存布局如下图所示：



为了实现线程初始化，需要在task_init内实现以下任务：

（1）为 idle 设置 task_struct 。并将 current ,task[0] 都指向 idle

（2）将 task[1] ~ task[NR_TASKS - 1] , 全部初始化, 这里和 idle设置的区别在于要为这些线程设置 thread_struct 中的 ra 和 sp.

task_init

```
#include "proc.h"
#include "mm.h"
#include "rand.h"
#include "printk.h"
#include "defs.h"

extern void __dummy();

struct task_struct* idle;           // idle process
struct task_struct* current;        // 指向当前运行线程的 `task_struct`
struct task_struct* task[NR_TASKS]; // 线程数组, 所有的线程都保存在此

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

```
    /* YOUR CODE HERE */
    uint64 addr_idle = kalloc();
    //在mm.c里，返回一个uint64类型，一个分配好空间的内存页
    idle = (struct task_struct*)addr_idle;
    //类型转换，到底部去更改
    idle->state = TASK_RUNNING;
    idle->counter = idle->priority = 0;
    idle->pid = 0;

    current = task[0] = idle;

    // 1. 参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进行初始化
    // 2. 其中每个线程的 state 为 TASK_RUNNING, counter 为 0, priority 使用 rand() 来设置, pid 为该线程在线程数组中的下标。
    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct` 中的 `ra` 和 `sp`,
    // 4. 其中 `ra` 设置为 __dummy （见 4.3.2）的地址， `sp` 设置为 该线程申请的物理页的高地址

    /* YOUR CODE HERE */
    for(int i = 1; i < NR_TASKS; i++){
        uint64 task_addr = kalloc();
        task[i] = (struct task_struct*)task_addr;
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand() % 10 + 1;//随机分配，最大不超过10
        task[i]->pid = i;//进程号就是这是第几个进程
        task[i]->thread.ra = (uint64)__dummy;//设置为dummy的地址
        task[i]->thread.sp = task_addr + 4096;//最低的地方+4096到4KB页的最上面
    }

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

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;
            //这里指的是自动增，用来表示计数器的线程数
            printk("[PID = %d] is running. auto_inc_local_var = %d\n", current->pid, auto_inc_local_var);
        }
    }
}
```

3.3 线程首次调度

当线程在运行时, 由于时钟中断的触发, 会将当前运行线程的上下文环境保存在栈上。当线程再次被调度时, 会将上下文从栈上恢复, 但是当我们创建一个新的线程, 此时线程的栈为空，当这个线程被调度时, 是没有上下文需要被恢复的, 所以我们需要为线程第一次调度提供一个特殊的返回函数dummy。

因此，在这个部分需要实现：在__dummy 中将 sepc 设置为 dummy() 的地址, 并使用 sret 从中断中返回。

_dummy

```
    la t0, dummy
    #借助寄存器去完成sepc为dummy的赋值
    csrw sepc, t0
    sret
```

3.4 线程切换

如果下一个执行的线程 next 与当前的线程 current不为同一个线程，则需要进行线程切换，也就是调用_swtich to。

switch_to

```
void switch_to(struct task_struct* next) {
    if (next == current) return; // switching to current is needless
    else {
        struct task_struct* current_saved = current;
        current = next;
        __switch_to(current_saved, next);//传入两个参数，进行线程切换
    }
}
```

此处，__switch_to 接受两个 task_struct 指针作为参数。实现的任务是，保存当前线程的ra , sp , s0~s11 到当前线程的thread_struct 中将下一个线程的 thread_struct 中的相关数据载入到ra , sp , s0~s11 中

_swtich_to

```
__switch_to:
    #这里传进来的a0是保存的，a1是next的地址
    # save state to prev process
    # including ra, sp, s0-s11
    add t0, a0, 40  # offset of thread_struct in current 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
    add t0, a1, 40  # offset of thread_struct in 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
```

3.5 调度函数入口设计

在proc.c中，要实现的是时钟中断处理函数。

do_timer()

```
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 == task[0]) schedule();
    else {
        current->counter -= 1;
        if (current->counter == 0) schedule();
    }
}
```

3.6 线程调度函数

首先要实现的是短作业优先调度。

这里的实现思路为：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。

DSJF

```
#ifdef DSJF
void schedule(void){
    uint64 min_count = INF;
    struct task_struct* next = NULL;
    char all_zeros = 1;
    for(int i = 1; i < NR_TASKS; i++){
        if (task[i]->state == TASK_RUNNING && task[i]->counter > 0) {
            if (task[i]->counter < min_count) {
                min_count = task[i]->counter;
                next = task[i];
            }
            all_zeros = 0;
        }
    }
```

```
    if (all_zeros) {//如果都没有进行分配
        printk("\n");
        for(int i = 1; i < NR_TASKS; i++){
            task[i]->counter = rand() % 10 + 1;
            printk("SET [PID = %d COUNTER = %d]\n", task[i]->pid, task[i]->counter);
        }
        schedule();
    }
    else {//具有下一个，继续做
        if (next) {
            printk("\nswitch to [PID = %d COUNTER = %d]\n", next->pid, next->counter);
            switch_to(next);
        }
    }
}
#endif
```

其次要实现的是优先级调度。参考的网页为'sched.c - kernel/sched.c - Linux source code (0.11) - Bootlin',从优先级中选取并进行操作。

DPRIORITY

```
#ifdef DPRIORITY
void schedule(void){
    uint64 c, i, next;
    struct task_struct** p;
    while(1)
    {
        c = 0;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while(--i) {
            if (!*--p) continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }

        if (c) break;
        // else all counters are 0s
        printk("\n");
        for(p = &task[NR_TASKS-1]; p > &task[0]; --p) {
            if (*p) {
                (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
                printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", (*p)->pid, (*p)->priority, (*p)->counter);
            }
        }
    }

    printk("\nswitch to [PID = %d PRIORITY = %d COUNTER = %d]\n", task[next]->pid, task[next]->priority, task[next]->counter);
    switch_to(task[next]);
}
#endif
```

4 实验结果验证

输入以下命令：

```
docker start oslab22
docker exec -it oslab22 /bin/bash
cd /home/oslab/os22fall-stu/src/lab3
export RISCV=/opt/riscv
export PATH=$PATH:$RISCV/bin
riscv64-unknown-linux-gnu-gdb vmlinux#调试使用语句
```

首先进行的是优先级调度：如下是使用随机数函数进行相应的赋值：

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

其次是进行调度，截取其中一部分：

```
switch to [PID = 18 PRIORITY = 9 COUNTER = 9]
[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
[PID = 18] is running. auto_inc_local_var = 8
[PID = 18] is running. auto_inc_local_var = 9

switch to [PID = 16 PRIORITY = 8 COUNTER = 8]
[PID = 16] is running. auto_inc_local_var = 1
[PID = 16] is running. auto_inc_local_var = 2
[PID = 16] is running. auto_inc_local_var = 3
[PID = 16] is running. auto_inc_local_var = 4
[PID = 16] is running. auto_inc_local_var = 5
[PID = 16] is running. auto_inc_local_var = 6
[PID = 16] is running. auto_inc_local_var = 7
[PID = 16] is running. auto_inc_local_var = 8

switch to [PID = 21 PRIORITY = 7 COUNTER = 7]
[PID = 21] is running. auto_inc_local_var = 1
[PID = 21] is running. auto_inc_local_var = 2
[PID = 21] is running. auto_inc_local_var = 3
[PID = 21] is running. auto_inc_local_var = 4
[PID = 21] is running. auto_inc_local_var = 5
[PID = 21] is running. auto_inc_local_var = 6
[PID = 21] is running. auto_inc_local_var = 7
```

修改CFLAG，进行下一种调度

```
CFLAG = ${CF} ${INCLUDE} -D DSJF
```

接着进行的是短作业优先调度

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

这里按照运行时间，短作业优先，顺序执行

```
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
switch to [PID = 17 COUNTER = 1]
[PID = 17] is running. auto_inc_local_var = 1

switch to [PID = 27 COUNTER = 1]
[PID = 27] is running. auto_inc_local_var = 1

switch to [PID = 12 COUNTER = 2]
[PID = 12] is running. auto_inc_local_var = 1
[PID = 12] is running. auto_inc_local_var = 2

switch to [PID = 28 COUNTER = 2]
[PID = 28] is running. auto_inc_local_var = 1
[PID = 28] is running. auto_inc_local_var = 2

switch to [PID = 1 COUNTER = 3]
[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

switch to [PID = 2 COUNTER = 3]
[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

switch to [PID = 9 COUNTER = 3]
[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

switch to [PID = 14 COUNTER = 3]
```

5 思考题与心得

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？

首先每个进程需要保存ra 和 sp 的值，之外需要 12 个寄存器来保存 s0~s11, 这样就是总共的 14 个寄存 器，而剩下的部分寄存器，如 gp tp 在本实验中没有涉及到，所以没有保存的必要，还有部分寄存器 是 caller saved registers , 在 context_switch 部分不需要保存。

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)

在之后的线程调用context_switch 中，ra 保存/恢复的函数返回点是 traps 中trap_handler 返回后的下一条指令 。实际上我们实现的进程没有为每一个进程保存各自的 pc 值，而是所有进程共同执行一段代码（如果所有进程都已经执行过了的情况下才是这样，如果开始执行一个之前没有执行过的程序，那么会进入dummy_dummy而重新开始执而重新开始执行行dummy）。因此当发生进程切换后，下一个进程最终将返回dummy中上一个进程发中上一个进程发生时钟中断的地址处继续执行。

riscv64-unknown-linux-gnu-gdb vmlinux

- 当发生时钟中断并需要进行调度时，首先进入到_traps，ra保存dummy当前执行的位置，然后ra被压栈保存，进入中断处理程序

```
[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 = 4, priority = 5, counter = 7]
[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
[PID = 4] is running. auto_inc_local_var = 6
[PID = 4] is running. auto_inc_local_var = 7
[PID = 4] is running. auto_inc_local_var = 8
[PID = 4] is running. auto_inc_local_var = 9
set [pid = 1, priority = 2, counter = 5]
set [pid = 2, priority = 5, counter = 8]
set [pid = 3, priority = 1, counter = 8]
set [pid = 4, priority = 5, counter = 9]
switch to [pid = 1, priority = 2, counter = 5]
[PID = 1] is running. auto_inc_local_var = 15
[PID = 1] is running. auto_inc_local_var = 16
[PID = 1] is running. auto_inc_local_var = 17
[PID = 1] is running. auto_inc_local_var = 18
[PID = 1] is running. auto_inc_local_var = 19
```

```
   0x80200140 <_traps+76>  sd      s3,-160(sp)
   0x80200144 <_traps+80>  sd      s4,-168(sp)
   0x80200148 <_traps+84>  sd      s5,-176(sp)
   0x8020014c <_traps+88>  sd      s6,-184(sp)
   0x80200150 <_traps+92>  sd      s7,-192(sp)
   0x80200154 <_traps+96>  sd      s8,-200(sp)
B+>0x80200158 <_traps+100> sd      s9,-208(sp)
   0x8020015c <_traps+104> sd      s10,-216(sp)
   0x80200160 <_traps+108> sd      s11,-224(sp)
   0x80200164 <_traps+112> sd      t3,-232(sp)
   0x80200168 <_traps+116> sd      t4,-240(sp)
   0x8020016c <_traps+120> sd      t5,-248(sp)
   0x80200170 <_traps+124> sd      t6,-256(sp)
remote Thread 1.1 In: _traps                                 L??   PC: 0x8020015

Breakpoint 1, 0x000000008020074c in do_timer ()
Continuing.

Breakpoint 5, 0x0000000080200158 in _traps ()
(gdb) i r ra
ra             0x80200548       0x80200548 <dummy+116>
```

- 进入中断处理程序后，ra保存的是traps中调用trap_handler的位置，之后ra保持不变至switch_to，ra被保存到该进程的栈中，并将下一个进程的栈中的ra拿到寄存器ra中，如果该进程是第一次执行，那么取出来的值是dummy，如果不是，那么取出来的值是上一次放入的traps中trap_handler返回的位置。

```
[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 = 4, priority = 5, counter = 7]
[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
[PID = 4] is running. auto_inc_local_var = 6
[PID = 4] is running. auto_inc_local_var = 7
[PID = 4] is running. auto_inc_local_var = 8
[PID = 4] is running. auto_inc_local_var = 9
set [pid = 1, priority = 2, counter = 5]
set [pid = 2, priority = 5, counter = 8]
set [pid = 3, priority = 1, counter = 8]
set [pid = 4, priority = 5, counter = 9]
switch to [pid = 1, priority = 2, counter = 5]
[PID = 1] is running. auto_inc_local_var = 15
[PID = 1] is running. auto_inc_local_var = 16
[PID = 1] is running. auto_inc_local_var = 17
[PID = 1] is running. auto_inc_local_var = 18
[PID = 1] is running. auto_inc_local_var = 19
```

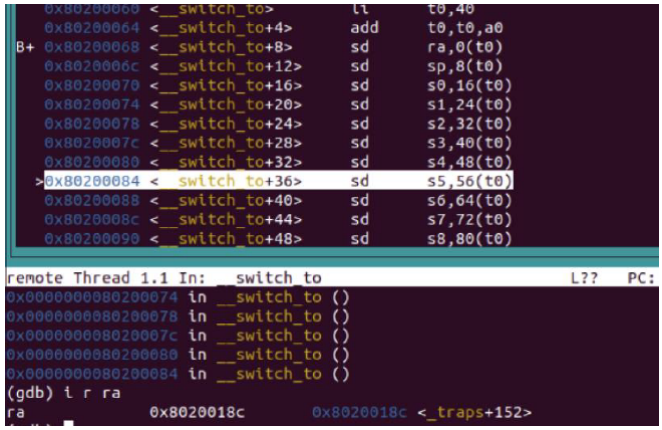
```
   0x802005f0 <schedule+40>        addi    s1,s1,-1484
   0x802005f4 <schedule+44>        auipc   s4,0x5
   0x802005f8 <schedule+48>        addi    s4,s4,-1460
   0x802005fc <schedule+52>        li      s3,10
   0x80200600 <schedule+56>        auipc   s2,0x2
   0x80200604 <schedule+60>        addi    s2,s2,-1384
B+>0x80200608 <schedule+64>        ld      a5,8(s1)
   0x8020060c <schedule+68>        ld      a5,16(a5)
   0x80200610 <schedule+72>        beqz    a5,0x80200620 <schedule+88>
   0x80200614 <schedule+76>        bgeu    a5,s5,0x802006f4 <schedule+300>
   0x80200618 <schedule+80>        mv      s5,a5
   0x8020061c <schedule+84>        li      a5,1
   0x80200620 <schedule+88>        ld      a4,16(s1)
remote Thread 1.1 In: schedule                                L??   PC: 0x80200
Breakpoint 1, 0x000000008020074c in do_timer ()
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000080200608 in schedule ()
(gdb) i r ra
ra             0x8020018c       0x8020018c <_traps+152>
```

```
[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 = 4, priority = 5, counter = 7]
[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
[PID = 4] is running. auto_inc_local_var = 6
[PID = 4] is running. auto_inc_local_var = 7
[PID = 4] is running. auto_inc_local_var = 8
[PID = 4] is running. auto_inc_local_var = 9
set [pid = 1, priority = 2, counter = 5]
set [pid = 2, priority = 5, counter = 8]
set [pid = 3, priority = 1, counter = 8]
set [pid = 4, priority = 5, counter = 9]
switch to [pid = 1, priority = 2, counter = 5]
[PID = 1] is running. auto_inc_local_var = 15
[PID = 1] is running. auto_inc_local_var = 16
[PID = 1] is running. auto_inc_local_var = 17
[PID = 1] is running. auto_inc_local_var = 18
[PID = 1] is running. auto_inc_local_var = 19
switch to [pid = 2, priority = 5, counter = 8]
```

```
   0x80200088 <__switch_to+40>     sd      s6,64(t0)
   0x8020008c <__switch_to+44>     sd      s7,72(t0)
   0x80200090 <__switch_to+48>     sd      s8,80(t0)
   0x80200094 <__switch_to+52>     sd      s9,88(t0)
   0x80200098 <__switch_to+56>     sd      s10,96(t0)
   0x8020009c <__switch_to+60>     sd      s11,104(t0)
   0x802000a0 <__switch_to+64>     li      t0,40
   0x802000a4 <__switch_to+68>     add     t0,t0,a1
   0x802000a8 <__switch_to+72>     ld      ra,0(t0)
   0x802000ac <__switch_to+76>     ld      sp,8(t0)
>0x802000b0 <__switch_to+80>     ld      s0,16(t0)
   0x802000b4 <__switch_to+84>     ld      s1,24(t0)
   0x802000b8 <__switch_to+88>     ld      s2,32(t0)
remote Thread 1.1 In: __switch_to                             L??   PC: 0x
0x00000000802000a0 in __switch_to ()
0x00000000802000a4 in __switch_to ()
0x00000000802000a8 in __switch_to ()
0x00000000802000ac in __switch_to ()
0x00000000802000b0 in __switch_to ()
(gdb) i r ra
ra             0x8020018c       0x8020018c <_traps+152>
```

```
[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 = 4, priority = 5, counter = 7]
[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
[PID = 4] is running. auto_inc_local_var = 6
[PID = 4] is running. auto_inc_local_var = 7
[PID = 4] is running. auto_inc_local_var = 8
[PID = 4] is running. auto_inc_local_var = 9
set [pid = 1, priority = 2, counter = 5]
set [pid = 2, priority = 5, counter = 8]
set [pid = 3, priority = 1, counter = 8]
set [pid = 4, priority = 5, counter = 9]
switch to [pid = 1, priority = 2, counter = 5]
[PID = 1] is running. auto_inc_local_var = 15
[PID = 1] is running. auto_inc_local_var = 16
[PID = 1] is running. auto_inc_local_var = 17
[PID = 1] is running. auto_inc_local_var = 18
[PID = 1] is running. auto_inc_local_var = 19
switch to [pid = 2, priority = 5, counter = 8]
```



- traps中随后会把之前压入栈的所有寄存器全部拿回寄存器中，ra中的值恢复，接着traps结束返回到ra保存的dummy中的某个位置执行。

心得：

在这次实验中，主要实现的是两种调度方式。由因为是非抢占式的调度，因此两种调度其实非常的像，一个是根据随机生成的较短时间，去选择还未实现的进程中的较短进程运行；一个是挑选优先级最高的进行。而在处理细节上，比较重要的我觉得是线程切换，这里我进一步体会了上下文切换中对寄存器值的保存和加载。此外，在验收的过程中我还对_dummy有了更多的理解。当线程再次被调度时，会将上下文从栈上恢复，但是当我们创建一个新的线程，此时线程的栈为空，所以需要提供一个特殊的返回机制，使得能够跳转pc；也明白了ret指令是保存下pc进行无条件跳转的作用。

通过这次实验，我对指令、调试和课堂的理论知识都更加明白了。