

# 浙江大学

## 本科实验报告

课程名称	操作系统
姓 名	
学 院	计算机科学与技术学院
系	计算机科学与技术系
专 业	计算机科学与技术
学 号	
指导教师	夏莹杰

2022 年 11 月 18 日

# 浙江大学操作系统实验报告

实验名称: **Lab3: RV64内核线程调度**

实验地点: 曹西503机房

实验日期: 2022年11月3日

## 一、实验目的和要求

- 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。

## 二、实验过程

### 1. 准备工程

- 此次实验基于 lab2 所实现的代码进行。
- 我们从实验 repo 同步需要使用到的源文件 rand.h/c, string.h/c, mm.h/c。
- 我们需要在 head.S 的适当位置添加对 mm\_init() 和 task\_init() (后续说明) 的调用, 以初始化内存管理和线程池:

```
1 la sp, boot_stack_top
2 call mm_init
3 call task_init
```

- 为了使用一些预定义的宏, 需要在 defs.h 中添加以下宏定义:

```
1 #define PHY_START 0x0000000080000000
2 #define PHY_SIZE 128 * 1024 * 1024 // 128MB, QEMU 默认内存大小
3 #define PHY_END (PHY_START + PHY_SIZE)
4
5 #define PGSIZE 0x1000 // 4KB
6 #define PGROUNDUP(addr) ((addr + PGSIZE - 1) & ~(PGSIZE - 1))
7 #define PGROUNDDOWN(addr) (addr & ~(PGSIZE - 1))
```

## 2. 线程调度功能实现

### ①线程初始化

为了执行线程调度，我们首先在task\_init()中设置一系列的线程。

首先，我们需要为我们已经运行的操作系统设置一个线程idle。

为了模拟实际上的线程运作，我们预设置31个内核线程，并设置其信息、存储ra和sp。

```
1 void task_init() {
2     idle = (struct task_struct*) kalloc();
3     idle->state = TASK_RUNNING;
4     idle->counter = 0;
5     idle->priority = 0;
6     idle->pid = 0;
7     current = task[0] = idle;
8
9     for(int i = 1; i < NR_TASKS; ++i){
10         task[i] = (struct task_struct*) kalloc();
11         task[i]->state = TASK_RUNNING;
12         task[i]->counter = 0;
13         task[i]->priority = rand();
14         task[i]->pid = i;
15         task[i]->thread.ra = (uint64) __dummy;
16         task[i]->thread.sp = (uint64) task[i] + PGSIZE;
17     }
18
19     printk("...proc_init done!\n");
20 }
```

### ②实现调度入口\_\_dummy

因为进程第一次调度时不需要恢复上下文，因此我们为这些进程设置一个特殊的调度入口\_\_dummy()。

这个程序段写入sepc，然后将特权等级返回到User。

```
1     .global __dummy
2 __dummy:
3     la t0, dummy
4     csrw sepc, t0
5     sret
```

我们为进程运行状态设计的函数dummy()如下，它会输出当前进程的一些信息：

```
1 void dummy() {
2     uint64 MOD = 1000000007;
3     uint64 auto_inc_local_var = 0;
4     int last_counter = -1;
5     while(1) {
6         if (last_counter == -1 || current->counter != last_counter) {
7             last_counter = current->counter;
8             auto_inc_local_var = (auto_inc_local_var + 1) % MOD;
9             printk("[PID = %d] is running. auto_inc_local_var = %d\n",
10                 current->pid, auto_inc_local_var);
11         }
12     }
13 }
```

### ③实现线程切换

我们先用汇编实现上下文切换\_\_switch\_to()。

我们把旧进程的所有需要保存的寄存器放进task\_struct中，然后从新进程的task\_struct中读取所有保存的寄存器，这样就完成了一次完成上下文切换。

```
1 __switch_to:
2     # save state to prev process
3     sd ra, 40(a0)
4     sd sp, 48(a0)
5     sd s0, 56(a0)
6     sd s1, 64(a0)
7     sd s2, 72(a0)
8     sd s3, 80(a0)
9     sd s4, 88(a0)
10    sd s5, 96(a0)
11    sd s6, 104(a0)
12    sd s7, 112(a0)
13    sd s8, 120(a0)
14    sd s9, 128(a0)
15    sd s10, 136(a0)
16    sd s11, 144(a0)
17
18    # restore state from next process
```

```

19     ld ra, 40(a1)
20     ld sp, 48(a1)
21     ld s0, 56(a1)
22     ld s1, 64(a1)
23     ld s2, 72(a1)
24     ld s3, 80(a1)
25     ld s4, 88(a1)
26     ld s5, 96(a1)
27     ld s6, 104(a1)
28     ld s7, 112(a1)
29     ld s8, 120(a1)
30     ld s9, 128(a1)
31     ld s10, 136(a1)
32     ld s11, 144(a1)
33
34     ret

```

我们使用c实现线程切换用函数switch\_to()。

如果要切换的线程正好是当前线程，则什么也不做；否则调用上下文切换\_\_switch\_to()。

```

1 void switch_to(struct task_struct* next) {
2     if(next != current){
3         struct task_struct* old = current;
4         current = next;
5         __switch_to(old, next);
6     }
7 }

```

#### ④实现调度入口函数

我们需要实现调度入口函数do\_timer()。

如果当前进程是idle，或者当前进程的counter自减后为0，则进行调度。

为了方便观察进程调度情况，我们在这里调用printk()输出当前进程的priority和counter。

```

1 void do_timer(){
2     printk("current prt: %d, ctr: %d\n",
3         current->priority, current->counter);

```

```

4     if(current == idle) schedule();
5     else if(--current->counter == 0) schedule();
6 }

```

同时我们在时钟中断处理程序中调用该函数以保证每隔确定时间进行调度。

```

1 void trap_handler(unsigned long scause, unsigned long sepc) {
2     if(scause & 0x8000000000000000){
3         if((scause & 0x00000005) == 0x00000005){
4             do_timer();
5             clock_set_next_event();
6         }
7     }
8 }

```

## ⑤实现线程调度算法

我们需要在schedule()中实现线程调度算法。

实验要求完成两种线程调度算法的编写，它们是：

短作业优先调度（SJF），考虑counter值较小的任务优先运行。

```

1 #ifdef SJF
2 void schedule(){
3     struct task_struct* next = task[0];
4     while(1){
5         for(int i = 1; i < NR_TASKS; ++i){
6             if(task[i]->counter && (next == task[0] ||
7                 task[i]->counter < next->counter)) next = task[i];
8         }
9         if(next->counter == 0){
10             for(int i = 1; i < NR_TASKS; ++i){
11                 task[i]->counter = rand();
12             }
13             }else break;
14     }
15     switch_to(next);
16 }
17 #endif

```

优先级调度 (PRIORITY) , 按照优先级从小到大进行调度; 如果优先级相同, 先让 counter 值较小的任务优先运行。

```
1 #ifdef PRIORITY
2 void schedule(){
3     struct task_struct* next = task[0];
4     while(1){
5         for(int i = 1; i < NR_TASKS; ++i){
6             if(task[i]->counter && (next == task[0] ||
7                 task[i]->priority < next->priority ||
8                 (task[i]->priority == next->priority &&
9                     task[i]->counter < next->counter))) next = task[i];
10        }
11        if(next->counter == 0){
12            for(int i = 1; i < NR_TASKS; ++i){
13                task[i]->counter = rand();
14            }
15        }else break;
16    }
17    switch_to(next);
18 }
```

注意到当所有内核线程的 counter 为 0 时我们需要给所有 counter 重新随机赋值。

在编译时, 我们通过 -D 选项进行对这两个算法实现的选择性编译。

### 3. 编译及测试

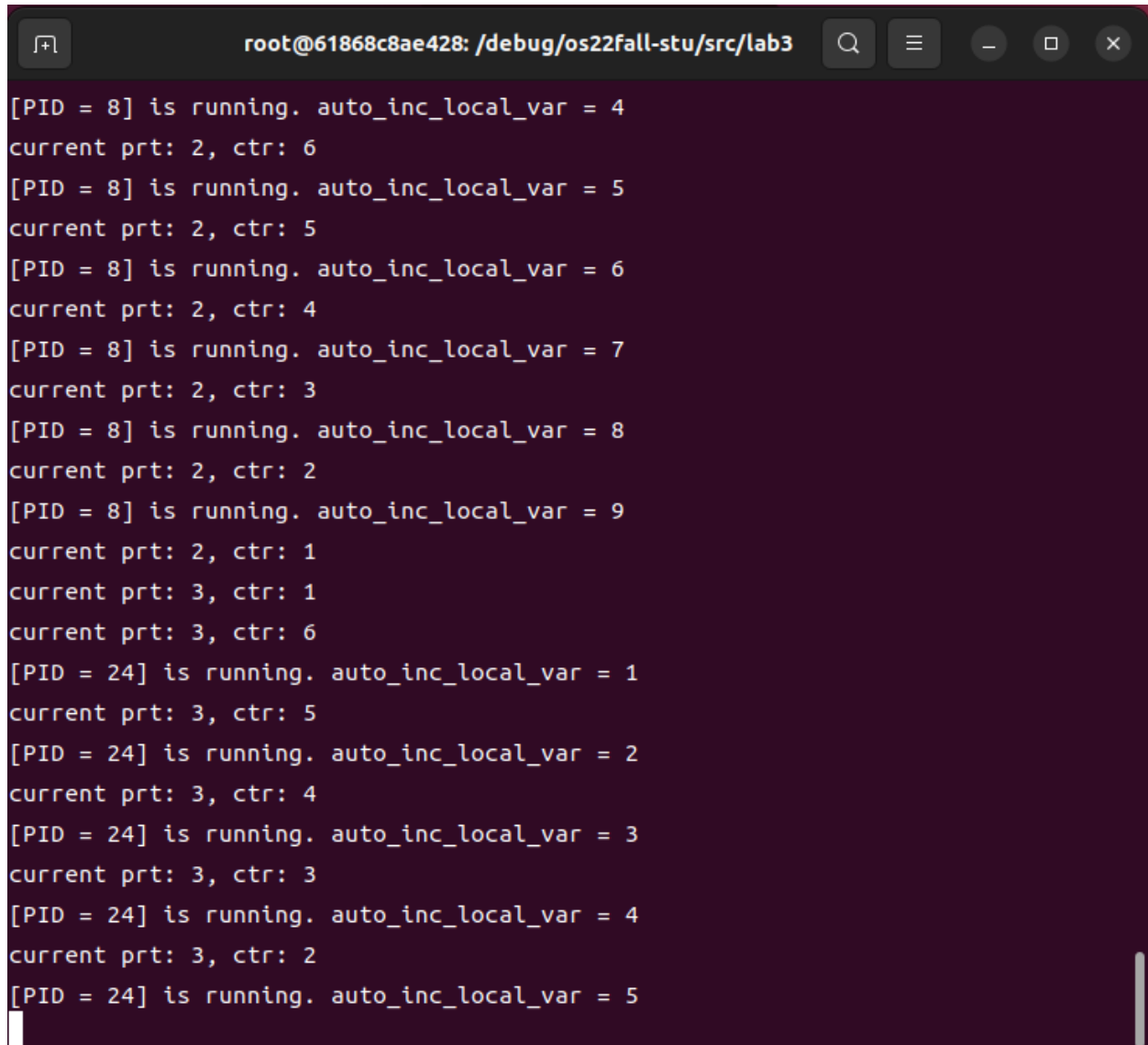
我们先在 makefile 中使用 -DSJF 选项, make run 结果如下:

```
root@61868c8ae428: /debug/os22fall-stu/src/lab3
[PID = 1] is running. auto_inc_local_var = 1
current prt: 1, ctr: 1
current prt: 4, ctr: 2
[PID = 2] is running. auto_inc_local_var = 1
current prt: 4, ctr: 1
current prt: 9, ctr: 2
[PID = 9] is running. auto_inc_local_var = 1
current prt: 9, ctr: 1
current prt: 10, ctr: 2
[PID = 14] is running. auto_inc_local_var = 1
current prt: 10, ctr: 1
current prt: 4, ctr: 3
[PID = 11] is running. auto_inc_local_var = 1
current prt: 4, ctr: 2
[PID = 11] is running. auto_inc_local_var = 2
current prt: 4, ctr: 1
current prt: 8, ctr: 3
[PID = 29] is running. auto_inc_local_var = 1
current prt: 8, ctr: 2
[PID = 29] is running. auto_inc_local_var = 2
current prt: 8, ctr: 1
current prt: 4, ctr: 4
[PID = 15] is running. auto_inc_local_var = 1
```

可见算法是根据counter值从小到大的顺序进行调度的，实现成功。



接下来使用-DPRIORITY选项，make run结果如下：



```
root@61868c8ae428: /debug/os22fall-stu/src/lab3

[PID = 8] is running. auto_inc_local_var = 4
current prt: 2, ctr: 6
[PID = 8] is running. auto_inc_local_var = 5
current prt: 2, ctr: 5
[PID = 8] is running. auto_inc_local_var = 6
current prt: 2, ctr: 4
[PID = 8] is running. auto_inc_local_var = 7
current prt: 2, ctr: 3
[PID = 8] is running. auto_inc_local_var = 8
current prt: 2, ctr: 2
[PID = 8] is running. auto_inc_local_var = 9
current prt: 2, ctr: 1
current prt: 3, ctr: 1
current prt: 3, ctr: 6
[PID = 24] is running. auto_inc_local_var = 1
current prt: 3, ctr: 5
[PID = 24] is running. auto_inc_local_var = 2
current prt: 3, ctr: 4
[PID = 24] is running. auto_inc_local_var = 3
current prt: 3, ctr: 3
[PID = 24] is running. auto_inc_local_var = 4
current prt: 3, ctr: 2
[PID = 24] is running. auto_inc_local_var = 5
```

可见算法是根据priority值从小到大的顺序进行调度的，当priority相等时则counter从小到大，实现成功。

### 三、讨论和心得

本次实验主要是完成操作系统内核线程调度的相关操作，实验过程中我们进一步理解了CPU上下文切换、CPU线程调度算法等机制，熟悉了riscv汇编指令以及gdb相关调试技巧。

我们在make过程中遇到困难，因为make会使用之前生成的.o文件，在修改后文件时间戳在.o文件时间戳之后的情况无法正确修改，使用gdb也无法发现这类错误。后来发现进行make clean后再进行make即可。

### 四、思考题

#### 1. 在RV64中一共32个通用寄存器，为什么context\_switch中只保存了14个？

这些寄存器大多（sp, s0-s11）是callee\_saved寄存器，需要用户进行自行保存。

其他寄存器是系统可以进行自动保存的，因此其值在线程切换时的变化不会损害运行状态。

ra寄存器与此不同，它需要保存程序的返回地址，在不同线程切换中可能会被改变到，因此必须强制保存其值。

#### 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（需要截图）。

ra保存和恢复的返回点是switch\_to()的末尾（其实也就是do\_timer()和schedule()的末尾，也就是将要进入clock\_set\_next\_event()的程序节点）。

我们考虑只有两个内核线程的情况，把程序断点设在\_\_switch\_to()的开始：

```
root@61868c8ae428: /debug/os22fall-stu/src/lab3
Domain0 HARTs      : 0*
Domain0 Region00   : 0x0000000000000000-0x0000000000001ffff (R,W,X)
Domain0 Region01   : 0x0000000000000000-0xffffffffffff (R,W,X)
Domain0 Next Address : 0x000000000200000
Domain0 Next Arg1   : 0x000000000700000
Domain0 Next Mode    : S-mode
Domain0 SysReset     : yes

Boot HART ID       : 0
Boot HART Domain    : root
Boot HART ISA       : rv64imafdcu
Boot HART Features   : scouteren,mcounteren
Boot HART PMP Count  : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 0
Boot HART MHPM Count : 0
Boot HART MIDELEG    : 0x0000000000000222
Boot HART MEDELEG     : 0x000000000000b109
...mm_init done!
...proc_init done!
2022 Hello RISC-V
current prt: 0, ctr: 0

Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vmlinux...
(gdb) target remote : 1234
Remote debugging using : 1234
0x0000000000001000 in ?? ()
(gdb) b __switch_to
Breakpoint 1 at 0x80200188: file entry.S, line 98.
(gdb) c
Continuing.

Breakpoint 1, __switch_to () at entry.S:98
98      sd ra, 40(a0)
(gdb) n
99      sd sp, 48(a0)
(gdb) i r ra
ra      0x80200700      0x80200700 <switch_to+84>
(gdb)
```

第一次调度进程1，把ra=switch\_to+84存入原线程数据；

```
root@61868c8ae428: /debug/os22fall-stu/src/lab3
Domain0 HARTs      : 0*
Domain0 Region00   : 0x0000000000000000-0x0000000000001ffff (R,W,X)
Domain0 Region01   : 0x0000000000000000-0xffffffffffff (R,W,X)
Domain0 Next Address : 0x000000000200000
Domain0 Next Arg1   : 0x000000000700000
Domain0 Next Mode    : S-mode
Domain0 SysReset     : yes

Boot HART ID       : 0
Boot HART Domain    : root
Boot HART ISA       : rv64imafdcu
Boot HART Features   : scouteren,mcounteren
Boot HART PMP Count  : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count : 0
Boot HART MHPM Count : 0
Boot HART MIDELEG    : 0x0000000000000222
Boot HART MEDELEG     : 0x000000000000b109
...mm_init done!
...proc_init done!
2022 Hello RISC-V
current prt: 0, ctr: 0

103      sd s3, 80(a0)
(gdb) n
104      sd s4, 88(a0)
(gdb) n
105      sd s5, 96(a0)
(gdb) n
106      sd s6, 104(a0)
(gdb) n
107      sd s7, 112(a0)
(gdb) n
108      sd s8, 120(a0)
(gdb) n
109      sd s9, 128(a0)
(gdb) n
110      sd s10, 136(a0)
(gdb) n
111      sd s11, 144(a0)
(gdb) n
114      ld ra, 40(a1)
(gdb) n
115      ld sp, 48(a1)
(gdb) i r ra
ra      0x80200178      0x80200178 <__dummy>
(gdb)
```

第一次调度进程1，从线程读取ra=\_\_dummy；

```
root@61868c8ae428: /debug/os22fall-stu/src/lab3
[PID = 1] is running. auto_inc_local_var = 2
current prt: 1, ctr: 8
[PID = 1] is running. auto_inc_local_var = 3
current prt: 1, ctr: 7
[PID = 1] is running. auto_inc_local_var = 4
current prt: 1, ctr: 6
[PID = 1] is running. auto_inc_local_var = 5
current prt: 1, ctr: 5
[PID = 1] is running. auto_inc_local_var = 6
current prt: 1, ctr: 4
[PID = 1] is running. auto_inc_local_var = 7
current prt: 1, ctr: 3
[PID = 1] is running. auto_inc_local_var = 8
current prt: 1, ctr: 2
[PID = 1] is running. auto_inc_local_var = 9
current prt: 1, ctr: 1
current prt: 4, ctr: 4
[PID = 2] is running. auto_inc_local_var = 1
current prt: 4, ctr: 3
[PID = 2] is running. auto_inc_local_var = 2
current prt: 4, ctr: 2
[PID = 2] is running. auto_inc_local_var = 3
current prt: 4, ctr: 1

(gdb) n
108      sd s8, 120(a0)
(gdb) n
109      sd s9, 128(a0)
(gdb) n
110      sd s10, 136(a0)
(gdb) n
111      sd s11, 144(a0)
(gdb) n
114      ld ra, 40(a1)
(gdb) n
115      ld sp, 48(a1)
(gdb) i r ra
ra      0x80200178      0x80200178 <__dummy>
(gdb) c
Continuing.

Breakpoint 1, __switch_to () at entry.S:98
98      sd ra, 40(a0)
(gdb) n
99      sd sp, 48(a0)
(gdb) i r ra
ra      0x80200700      0x80200700 <switch_to+84>
(gdb)
```

第二次调度进程1，存入ra=switch\_to+84；

```
root@61868c8ae428: /debug/os22fall-stu/src/lab3
[PID = 1] is running. auto_inc_local_var = 2
current prt: 1, ctr: 8
[PID = 1] is running. auto_inc_local_var = 3
current prt: 1, ctr: 7
[PID = 1] is running. auto_inc_local_var = 4
current prt: 1, ctr: 6
[PID = 1] is running. auto_inc_local_var = 5
current prt: 1, ctr: 5
[PID = 1] is running. auto_inc_local_var = 6
current prt: 1, ctr: 4
[PID = 1] is running. auto_inc_local_var = 7
current prt: 1, ctr: 3
[PID = 1] is running. auto_inc_local_var = 8
current prt: 1, ctr: 2
[PID = 1] is running. auto_inc_local_var = 9
current prt: 1, ctr: 1
current prt: 4, ctr: 4
[PID = 2] is running. auto_inc_local_var = 1
current prt: 4, ctr: 3
[PID = 2] is running. auto_inc_local_var = 2
current prt: 4, ctr: 2
[PID = 2] is running. auto_inc_local_var = 3
current prt: 4, ctr: 1

root@61868c8ae428: /debug/os22fall-stu/src/lab3
103      sd s3, 80(a0)
(gdb) n
104      sd s4, 88(a0)
(gdb) n
105      sd s5, 96(a0)
(gdb) n
106      sd s6, 104(a0)
(gdb) n
107      sd s7, 112(a0)
(gdb) n
108      sd s8, 120(a0)
(gdb) n
109      sd s9, 128(a0)
(gdb) n
110      sd s10, 136(a0)
(gdb) n
111      sd s11, 144(a0)
(gdb) n
114      ld ra, 40(a1)
(gdb) n
115      ld sp, 48(a1)
(gdb) i r ra
ra      0x80200700  0x80200700 <switch_to+84>
(gdb)
```

第二次调度进程1，读出ra=switch\_to+84。