浙江大學

本科实验报告

RV64 内核线程调度

课程名称：	操作系统
姓　　名：	陈杰伟
学　　院：	地球科学学院
专　　业：	地理信息科学
学　　号：	3200101205
指导老师：	寿黎但

2022 年 11 月 20 日

专业:	地理信息科学
姓名:	陈杰伟
学号:	3200101205
日期:	2022 年 11 月 20 日
地点:	玉泉曹光彪-西 503

浙江大学实验报告

课程名称:	操作系统	指导老师:	寿黎但	成绩:	
实验名称:	RV64 内核线程调度	实验类型:	编程实验	同组学生姓名:	无

一、 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

二、 实验环境

Ubuntu 20.04

三、 实验步骤

1. 线程初始化

在初始化线程的时候, 我们参考 Linux v0.11 中的实现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。

OS run 起来的时候, 其本身就是一个线程 idle 线程, 第一步为 idle 设置 task_struct。并将 current, task[0] 都指向 idle。

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.

线程初始化

```c
void task_init()
{
    idle = (struct task_struct *)kalloc();
    // 调用 kalloc() 为 idle 分配一个物理页

    idle->state = TASK_RUNNING;
    // 设置 state 为 TASK_RUNNING;

    idle->counter = 0;
    idle->counter = PRIORITY_IDLE;
    // 由于 idle 不参与调度 可以将其 counter / priority 设置为 0

```

```
    idle->pid = 0;
    // 设置 idle 的 pid 为 0

    current = idle;
    task[0] = idle;

    // 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中的 ra 和 sp,
    // 其中 ra 设置为 __dummy的地址, sp 设置为 该线程申请的物理页的高地址

    int i = 0;
    for (i = 1; i < NR_TASKS; ++i)
    {
        task[i] = (struct task_struct *)kalloc();

        task[i]->state = TASK_RUNNING;

        task[i]->counter = 0;
        task[i]->priority = rand();

        task[i]->pid = i;

        task[i]->thread.ra = (uint64)&__dummy;
        task[i]->thread.sp = (uint64)task[i] + PGSIZE;
    }

    printk("...proc_init done!\n");
    printk("Hello RISC-V\n");
    printk("idle process is running!\n");
    printk("\n");

    reset_thread();

    return;
}
```

2. 在 entry.S 添加 ___dummy

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，所以需要为线程第一次调度提供一个特殊的返回函数 ___dummy

在 ___dummy 中将 sepc 设置为 dummy() 的地址，并使用 sret 从中断中返回。

___dummy

```
__dummy:
la t0, dummy
csrw sepc, t0
sret
```

3. 实现调度入口函数

实现 do_timer(), 并在时钟中断处理函数中调用。

do_timer

```c
void do_timer(void)
{
    // 如果当前线程是 idle 线程 直接进行调度
    if (current->pid == 0)
        schedule();

    // 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
    else
    {
        current->counter--;
        if (current->counter > 0)
            return;
        else
            schedule();
    }
    return;
}
```

4. 实现线程切换

在 entry.S 中实现线程上下文切换 ___switch_to:
- ___switch_to 接受两个 task_struct 指针作为参数
- 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。
- 同时要在汇编中改变 current 指针所指向的地址，这样在 sret 触发时钟中断后不会重新再进入调度函数

___switch_to

```asm
    .globl __switch_to
__switch_to:
    # save state to prev process
    sd ra, 8*5(a0)
    sd sp, 8*6(a0)
    sd s0, 8*7(a0)
    sd s1, 8*8(a0)
    sd s2, 8*9(a0)
    sd s3, 8*10(a0)
    sd s4, 8*11(a0)
    sd s5, 8*12(a0)
    sd s6, 8*13(a0)
    sd s7, 8*14(a0)
    sd s8, 8*15(a0)
    sd s9, 8*16(a0)
    sd s10, 8*17(a0)
    sd s11, 8*18(a0)
```

```

    # restore state from next process
    ld ra, 8*5(a1)
    ld sp, 8*6(a1)
    ld s0, 8*7(a1)
    ld s1, 8*8(a1)
    ld s2, 8*9(a1)
    ld s3, 8*10(a1)
    ld s4, 8*11(a1)
    ld s5, 8*12(a1)
    ld s6, 8*13(a1)
    ld s7, 8*14(a1)
    ld s8, 8*15(a1)
    ld s9, 8*16(a1)
    ld s10, 8*17(a1)
    ld s11, 8*18(a1)

    #指针移动
    la t0, current
    sd a1, 0(t0)

    ret
```

5. 短作业优先调度算法

当需要进行调度时按照一下规则进行调度：

- 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。

短作业

```c
void schedule(void)
{
    int i = 0;
    int to_execute = -1;

    int to_reset = 1;

    for (i = 1; i < NR_TASKS; i++)
    {
        if (task[i]->state != TASK_RUNNING)
            continue;
        if (task[i]->counter > 0)
            to_reset = 0;
        else
            continue;
        if (to_execute == -1 || task[i]->counter < task[to_execute]->counter)
            to_execute = i;
```

```c
    }

    if (to_reset == 1)
    {
        reset_thread();
        schedule();
        return;
    }

    switch_to(task[to_execute]);

    return;
}
```

6. 优先级调度算法

参考 Linux v0.11 调度算法实现实现。

优先级

```c
void schedule(void)
{
    int i, next, c;
    struct task_struct **p;

    c = -1;
    next = 0;
    i = NR_TASKS;
    p = &task[NR_TASKS];

    while (--i)
    {
        if (!*--p)
            continue;
        if (((*p)->state == TASK_RUNNING) && ((int)(*p)->counter > c))
        {
            c = (*p)->counter;
            next = i;
        }
    }

    if (!c)
    {
        reset_thread();
        schedule();
        return;
    }

    switch_to(task[next]);
    return;
```

```
    }
```

7. 运行结果

短作业调度算法结果如图 1，能正确进行切换并读取保存上下文

优先级调度算法结果如图 2，能正确进行切换并读取保存上下文

四、 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?

sp 指示了线程运行栈的位置

ra 代表了线程所执行函数的地址

sp, ra, s0-s11 是 Callee, 保存了线程当前的执行状态，而剩下的寄存器都是 Caller，其中保存的信息在函数执行过程中可变，并没有必要将其存储到栈中

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)。

第一次调用时 ra 保存值为 __dummy 的地址如图 3，

之后调用时保存的是自己正在执行的任务，即 dummy 函数中循环的地址，如图 4

```
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 5]
SET [PID = 3 COUNTER = 2]
SET [PID = 4 COUNTER = 9]
SET [PID = 5 COUNTER = 4]


switch to [PID = 3 COUNTER = 2]
[PID = 3] is running. auto_inc_local_var = 1

switch to [PID = 5 COUNTER = 4]
[PID = 5] is running. auto_inc_local_var = 1
[PID = 5] is running. auto_inc_local_var = 2
[PID = 5] is running. auto_inc_local_var = 3

switch to [PID = 2 COUNTER = 5]
[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

switch to [PID = 4 COUNTER = 9]
[PID = 4] is running. auto_inc_local_var = 1
[PID = 4] is running. auto_inc_local_var = 2
[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

switch to [PID = 1 COUNTER = 10]
[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
[PID = 1] is running. auto_inc_local_var = 8
[PID = 1] is running. auto_inc_local_var = 9

SET [PID = 1 COUNTER = 4]
```

图 1: 短作业结果

```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.10
Boot HART Base ISA        : rv64imafdc
Boot HART ISA Extensions  : none
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

...mm_init done!
...proc_init done!
Hello RISC-V
idle process is running!


SET [PID = 1 PRIORITY = 1 COUNTER = 1]
SET [PID = 2 PRIORITY = 4 COUNTER = 4]
SET [PID = 3 PRIORITY = 10 COUNTER = 10]
SET [PID = 4 PRIORITY = 4 COUNTER = 4]
SET [PID = 5 PRIORITY = 10 COUNTER = 10]

switch to [PID = 5 PRIORITY = 10 COUNTER = 10]
[PID = 5] is running. auto_inc_local_var = 1
[PID = 5] is running. auto_inc_local_var = 2
[PID = 5] is running. auto_inc_local_var = 3
[PID = 5] is running. auto_inc_local_var = 4
[PID = 5] is running. auto_inc_local_var = 5
[PID = 5] is running. auto_inc_local_var = 6
[PID = 5] is running. auto_inc_local_var = 7
[PID = 5] is running. auto_inc_local_var = 8
[PID = 5] is running. auto_inc_local_var = 9

switch to [PID = 3 PRIORITY = 10 COUNTER = 10]
[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
[PID = 3] is running. auto_inc_local_var = 5
```

图 2: 优先级结果

```
Breakpoint 2, switch_to (next=0x87fe8000) at proc.c:161
161         if (next->pid == current->pid)
=> 0x000000008020078c <switch_to+20>:   83 37 84 fe     ld      a5,-24(s0)
   0x0000000080200790 <switch_to+24>:   03 b7 07 02     ld      a4,32(a5)
1: next->thread.ra = 2149581188
2: current->thread.ra = 2149581188
(gdb) 
```

图 3: 第一次 ra

```
Continuing.

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

Breakpoint 2, switch_to (next=0x87ffd000) at proc.c:161
161         if (next->pid == current->pid)
=> 0x000000008020078c <switch_to+20>:   83 37 84 fe     ld      a5,-24(s0)
   0x0000000080200790 <switch_to+24>:   03 b7 07 02     ld      a4,32(a5)
1: next->thread.ra = 2149582840
2: current->thread.ra = 2149582840
(gdb) 
                                                                    行 5、
```

图 4: 之后 ra