浙江大学实验报告

Lab 3：RV64内核线程调度

课程名称：　操作系统　实验类型：　综合

实验项目名称：　RV64内核线程调度

学生姓名：汪辉 专业：计算机科学与技术 学号：3190105609

同组学生姓名：个人实验　指导老师：　季江民

电子邮件：3190105609@zju.edu.cn

实验地点：　曹西503　实验日期：　2021 年 11 月 25 日

分工说明：lab 3内容组员两人都各自完成所有内容。

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 实验内容

完善代码

本实验基于lab2内容进行，需要添加以下现成文件：

`rand.h/rand.c`，`string.h/string.c`，`mm.h/mm.c`

需要基于lab2工程修改以下文件内容：

1. `arch/riscv/kernel/head.S`
2. `arch/riscv/kernel/entry.S`

还需新建和编辑以下文件：

1. `arch/riscv/include/proc.h`
2. `arch/riscv/kernel/proc.c`

修改`defs.h`

按照实验指导，在lab3中引入了简单的内存管理，在初始化时需要一些自定义的宏，因此需要在`defs.h`添加以下内容：

```
#define PHY_START 0x0000000080000000
#define PHY_SIZE 128 * 1024 * 1024 // 128MB， QEMU 默认内存大小
#define PHY_END (PHY_START + PHY_SIZE)

#define PGSIZE 0x1000 // 4KB
#define PGROUNDUP(addr) ((addr + PGSIZE - 1) & (~(PGSIZE - 1)))
#define PGROUNDDOWN(addr) (addr & (~(PGSIZE - 1)))
```

修改`head.S`

lab3中的`head.S`还需要初始化内存管理的部分，所以需要在`_start`的适当位置调用`mm_init`来初始化内存管理。

```
    .section .text.init
    .globl _start
_start:
    la sp, boot_stack_top
    call mm_init # in lab3
```

除此之外，由于线程任务也需要在开始时进行一次初始化，故在`_start`还要调用一次`task_init`来初始化线程。

```
_start:
    la sp, boot_stack_top
    call mm_init # in lab3

    la sp, boot_stack_top
    call task_init
```

修改`entry.S`

lab2中`entry.S`仅支持启用中断并完成保护上下文的任务，lab3时对中断的处理有所不同：

1. 添加`_dummy`功能段用于从中断返回到`dummy`函数。

    ```
        # -----------
            .globl __dummy
    __dummy:
        la t0, dummy
        csrw sepc, t0
        sret
        # YOUR CODE HERE
    ```

2. 添加`__witch_to`段用于完成线程切换时唤出线程寄存器的保护和换入线程寄存器的恢复

```
    # -----------

    .globl __switch_to
__switch_to:
    # save state to prev process
    sd ra, 5*8(a0)
    sd sp, 6*8(a0)
    sd s0, 7*8(a0)
    sd s1, 8*8(a0)
    sd s2, 9*8(a0)
```

```
    sd s3, 10*8(a0)
    sd s4, 11*8(a0)
    sd s5, 12*8(a0)
    sd s6, 13*8(a0)
    sd s7, 14*8(a0)
    sd s8, 15*8(a0)
    sd s9, 16*8(a0)
    sd s10, 17*8(a0)
    sd s11, 18*8(a0)
    # YOUR CODE HERE
    # restore state from next process
    ld ra, 5*8(a1)
    ld sp, 6*8(a1)
    ld s0, 7*8(a1)
    ld s1, 8*8(a1)
    ld s2, 9*8(a1)
    ld s3, 10*8(a1)
    ld s4, 11*8(a1)
    ld s5, 12*8(a1)
    ld s6, 13*8(a1)
    ld s7, 14*8(a1)
    ld s8, 15*8(a1)
    ld s9, 16*8(a1)
    ld s10, 17*8(a1)
    ld s11, 18*8(a1)
    # YOUR CODE HERE
    ret
```

添加`proc.h`

```
// arch/riscv/include/proc.h
#include "types.h"
#define NR_TASKS (1 + 7) // 用于控制 最大线程数量 （idle 线程 + 31 内核线程）
#define TASK_RUNNING 0    // 为了简化实验，所有的线程都只有一种状态
#define PRIORITY_MIN 1
#define PRIORITY_MAX 10
/* 用于记录 `线程` 的 `内核栈与用户栈指针` */
/* (lab3中无需考虑，在这里引入是为了之后实验的使用) */
struct thread_info
{
    uint64 kernel_sp;
    uint64 user_sp;
};
/* 线程状态段数据结构 */
struct thread_struct
{
    uint64 ra;
    uint64 sp;
    uint64 s[12];
};
/* 线程数据结构 */
struct task_struct
{
    struct thread_info *thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
```

```
    uint64 pid;      // 线程id
    struct thread_struct thread;
};
/* 线程初始化 创建 NR_TASKS 个线程 */
void task_init();
/* 在时钟中断处理中被调用 用于判断是否需要进行调度 */
void do_timer();
/* 调度程序 选择出下一个运行的线程 */
void schedule();
/* 线程切换入口函数*/
void switch_to(struct task_struct *next);
/* dummy funciton: 一个循环程序，循环输出自己的 pid 以及一个自增的局部变量*/
void dummy();
```

添加`proc.c`

```
// arch/riscv/kernel/proc.c
#include "proc.h"
#include "printk.h"
#include "types.h"
#include "defs.h"
// 外部调用
extern void __dummy();
extern void __switch_to(struct task_struct *prev, struct task_struct *next);
// 全局变量
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
```

1. 初始化函数`task_init`：

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

// 1.参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进行初始化
    for ( int i = 1 ; i < NR_TASKS ; i++ ) {
        uint64 point = kalloc() ;
        task[i] = (struct task_struct *) point ;
        task[i]->state = TASK_RUNNING ;
        task[i]->counter = 0 ;
        task[i]->priority = rand() ;
        task[i]->pid = i ;
        task[i]->thread.ra = __dummy ;
```

```
        task[i]->thread.sp = point + PGSIZE ;
    }
// 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 */

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

2. 线程切换函数`switch_to`：

```
void switch_to(struct task_struct *next)
{
    struct task_struct *temp = current ;
    current = next ;
    if ( next != temp ) {
        __switch_to( temp , next ) ;
    }
    /* YOUR CODE HERE */
}
```

3. 调度入口函数`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
            若剩余时间任然大于0 则直接返回 否则进行调度 */
    if ( current == idle ) {
        /* code */
        schedule() ;
    } else {
        current->counter -- ;
        if ( current->counter > 0 ) return ;
        schedule() ;
    }
    /* YOUR CODE HERE */
}
```

4. 线程调度`schedule`：

lab3实现两种调度算法，最短作业优先和最高优先级优先，两种算法实现时都用到了`reset()`来重置剩下作业长度：

```
void reset()
{
    for ( int i = 1 ; i < NR_TASKS ; i++ )
        task[i]->counter = rand() ;
    PrintTask() ;
}
```

其中打印函数`PrintTask()`用来观察结果：

```
void PrintTask()
{
    for ( int i = 1 ; i < NR_TASKS ; i++ )
        printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n",
               task[i]->pid,task[i]->priority,task[i]->counter) ;
}
```

- 最短作业优先

```
#ifdef SJF
void schedule(void)
{
    struct task_struct* next = NULL ;
    int shortest = 11 ;
    for ( int i = 1 ; i < NR_TASKS ; i++ ) {
        if ( shortest > task[i]->counter && task[i]->counter != 0 ) {
            next = task[i] ;
            shortest = task[i]->counter ;
        }
    }
    if ( shortest == 11 ) {
        reset() ;
        schedule() ;
    }
    else {
        printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n",next-
>pid,next->priority,next->counter) ;
        switch_to(next) ;
    }
    /* YOUR CODE HERE */
}
```

- 优先级调度

```
#else
void schedule(void)
{
    struct task_struct* next = NULL ;
    int pre = 11 ;
    for ( int i = 1 ; i < NR_TASKS ; i++ ) {
        if ( pre > task[i]->priority && task[i]->counter != 0 ) {
            next = task[i] ;
            pre = task[i]->priority ;
        }
    }
    if ( pre == 11 ) {
        reset() ;
        schedule() ;
    } else {
        printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n",next-
>pid,next->priority,next->counter) ;
        switch_to(next) ;
    }
}
#endif
```

实验结果

`make run` 观察运行结果

首先观察最短作业优先的运行结果：

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

优先级调度运行结果：（实现默认PRIORITY数值小的优先运行）

```
SET [PID = 1 PRIORITY = 8 COUNTER = 5]
SET [PID = 2 PRIORITY = 4 COUNTER = 9]
SET [PID = 3 PRIORITY = 5 COUNTER = 4]
SET [PID = 4 PRIORITY = 1 COUNTER = 5]
SET [PID = 5 PRIORITY = 7 COUNTER = 5]
SET [PID = 6 PRIORITY = 1 COUNTER = 1]
SET [PID = 7 PRIORITY = 4 COUNTER = 10]
switch to [PID = 4 PRIORITY = 1 COUNTER = 5]
[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
switch to [PID = 6 PRIORITY = 1 COUNTER = 1]
```

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

`make debug` 调试分析

进入 `_dummy` 段，线程切换前后，查看当前线程内容 `current`：

```
(gdb) display current
1: current = (struct task_struct *) 0x87ffb000
(gdb) finish
Run till exit from #0  dummy () at proc.c:61

Breakpoint 1, dummy () at proc.c:61
1: current = (struct task_struct *) 0x87ff9000
(gdb)
```

可以看到线程内容的切换，全局变量 `current` 指针指向了不同的内容。

3 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？

   有些寄存器是 `switch` 函数的调用者默认 `switch` 函数会做修改，所以调用者已经在自己的栈上保存了这些寄存器，当函数返回时，这些寄存器会自动恢复。所以 `switch` 段里只需要保存被调用者被要求保存的那些寄存器——`Callee Saved Register`。

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。

保存恢复的函数返回点也是`-dummy`，经过`_dummy`设置返回地址`dummy()`从中断回到`dummy()`的地址继续执行，此时的寄存器内容已经是切换后的线程的内容。

4 心得体会

- 按照实验指导，明确每一步的目的后，代码实现并不复杂。
- 操作系统实验与计算机体系结构内容联系紧密，本次实验关注线程之间的切换过程，事实上这对我们理解操作系统的实现有很大的帮助，通过这次实验对线程管理有了很好的理解。
- 实验指导中的一些参考代码似乎不广泛适用于所有人的代码习惯，比如提供的`dummy`函数，在使用时发现会漏掉切换后计时还剩1的线程的运行中打印，在队友的帮助下完成了这一问题的解决，发现与`dummy`中的具体变量有关。