浙江大學

操作系统原理与实践

lab3: RV64 内核线程调度

蒋景伟

Data: 2021-11-06

目录:

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

- ubuntu20.04 LTS
- Docker in Lab0

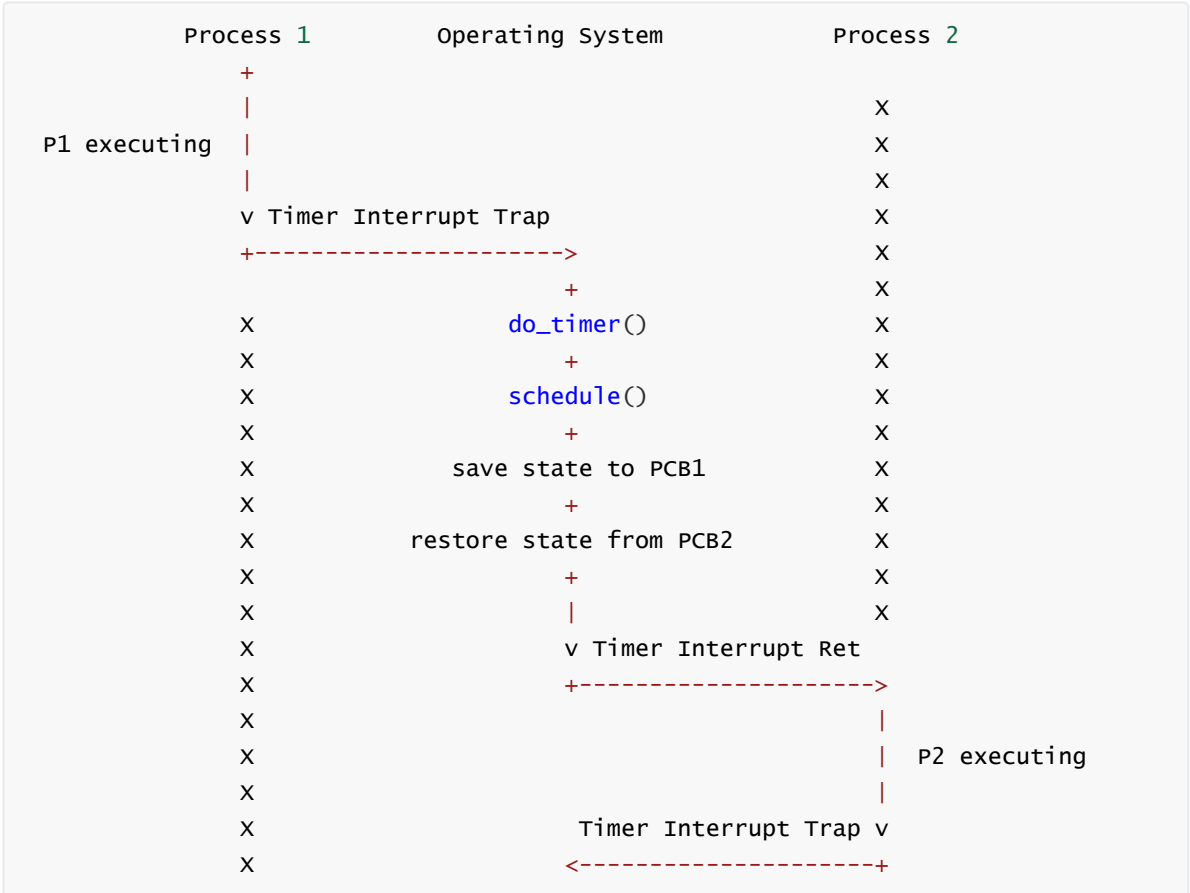
3 实验原理

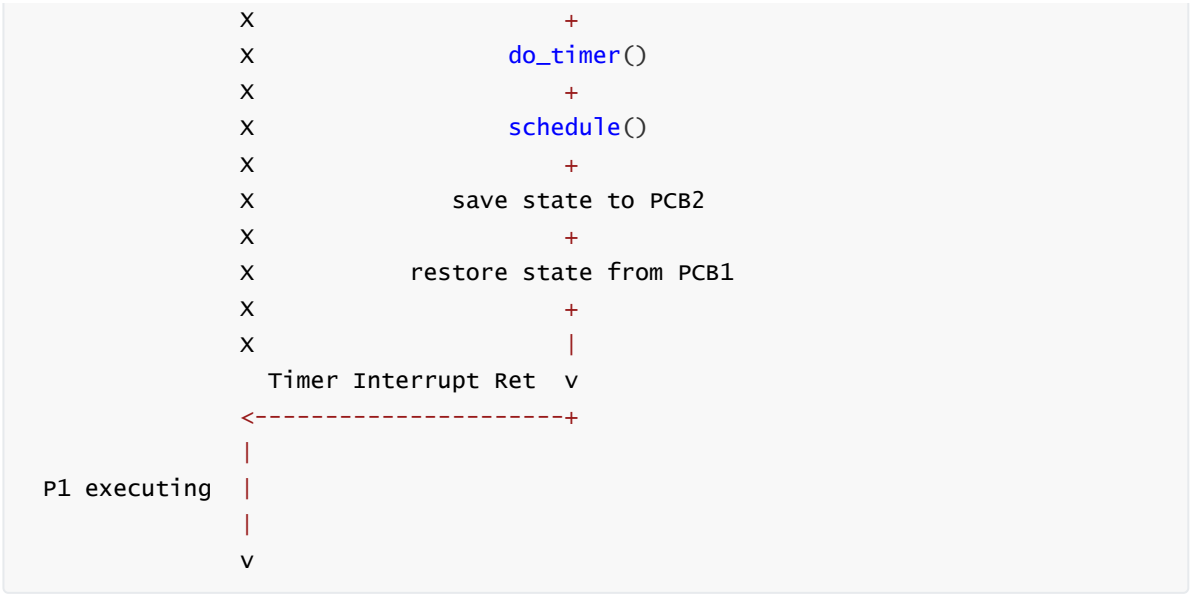
3.1 线程相关属性

在不同的操作系统中，为每个线程所保存的信息都不同。在这里，我们提供一种基础的实现，每个线程会包括：

- `线程ID`：用于唯一确认一个线程。
- `运行栈`：每个线程都必须有一个独立的运行栈，保存运行时的数据。
- `执行上下文`：当线程不在执行状态时，我们需要保存其上下文（其实就是`状态寄存器`的值），这样之后才能够将其恢复，继续运行。
- `运行时间片`：为每个线程分配的运行时间。
- `优先级`：在优先级相关调度时，配合调度算法，来选出下一个执行的线程。

3.2 线程切换流程图





- 在每次处理时钟中断时，操作系统首先会将当前线程的运行剩余时间减少一个单位。之后根据调度算法来确定是继续运行还是调度其他线程来执行。
- 在进程调度时，操作系统会遍历所有可运行的线程，按照一定的调度算法选出下一个执行的线程。最终将选择得到的线程与当前线程切换。
- 在切换的过程中，首先我们需要保存当前线程的执行上下文，再将将要执行线程的上下文载入到相关寄存器中，至此我们就完成了线程的调度与切换。

4 实验步骤

4.1 准备工程

1. 此次实验基于 lab2所实现的代码进行。
2. 在`os21fall`目录下执行`git pull`即可同步`repo`同步以下代码`rand.h/rand.c`，`string.h/string.c`，`mm.h/mm.c`。
3. 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。完成后框架如下：

```
├── arch
|   └── riscv
|       ├── include
|       |   ├── defs.h
|       |   ├── sbi.h
|       |   ├── printk.h
|       |   ├── stddef.h
|       |   ├── mm.h
|       |   └── types.h
|       ├── kernel
|       |   ├── head.S
|       |   ├── Makefile
|       |   ├── mm.c
|       |   ├── sbi.c
|       |   └── vmlinux.lds
|       └── Makefile
├── include
|   ├── print.h
```

```
|   ├── rand.h
|   ├── string.h
|   └── types.h
├── init
|   ├── main.c
|   ├── Makefile
|   └── test.c
├── lib
|   ├── Makefile
|   ├── rand.c
|   ├── string.c
|   └── printk.c
└── Makefile
```

4. 实验提供了`kalloc`接口，可以用来申请`4KB`的物理页，由于引入了简单的物理内存管理，需要在`_start`的适当位置调用`mm_init`，来初始化内存管理系统，并且在初始化时需要用一些自定义的宏，需要修改`defs.h`，在`defs.h`添加如下内容：

```
#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)))
```

5. 该部分最关键的是找到在适当的位置调用`mm_init`，一开始直接放在了`_start`的开头，发现会一直卡住不输出程序无法执行，进入debug模式，发现每执行到下面这个语句时就会自动退出：

```
   0x80200340 <mm_init>    addi    sp,sp,-32
   0x80200344 <mm_init+4>  lui     a5,0xfffff
>  0x80200348 <mm_init+8>  sd      s0,16(sp)
```

发现是`sp`并没有初始化，出错了，所以应将该函数放在`sp`初始化后面。

6. 完成后执行可得到以下结果，表示工程可以正常运行：

```
...mm_init done!
```

4.2 `proc.h` 数据结构定义

在之前增加的`proc.h`文件中增加以下数据结构定义：

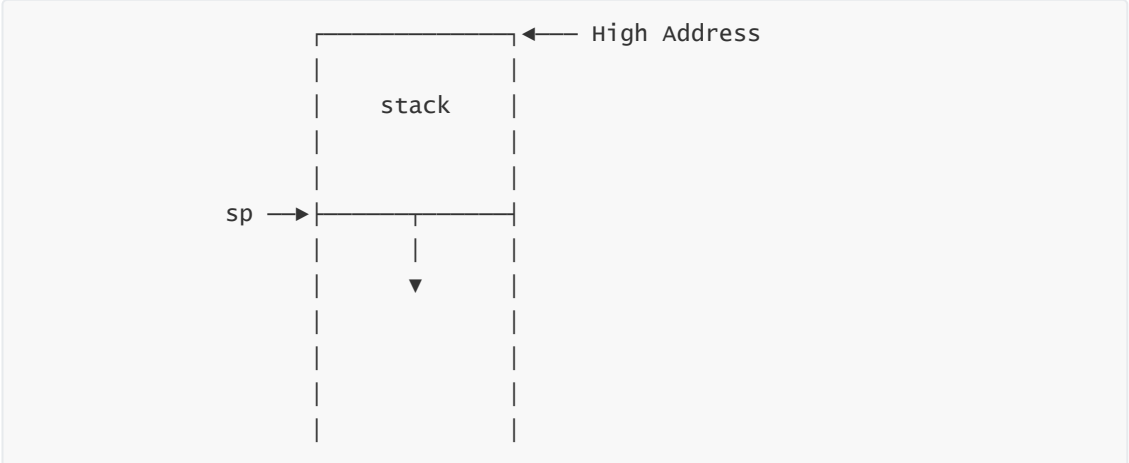
```
// arch/riscv/include/proc.h

#include "types.h"

#define NR_TASKS  (1 + 31) // 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 {
```

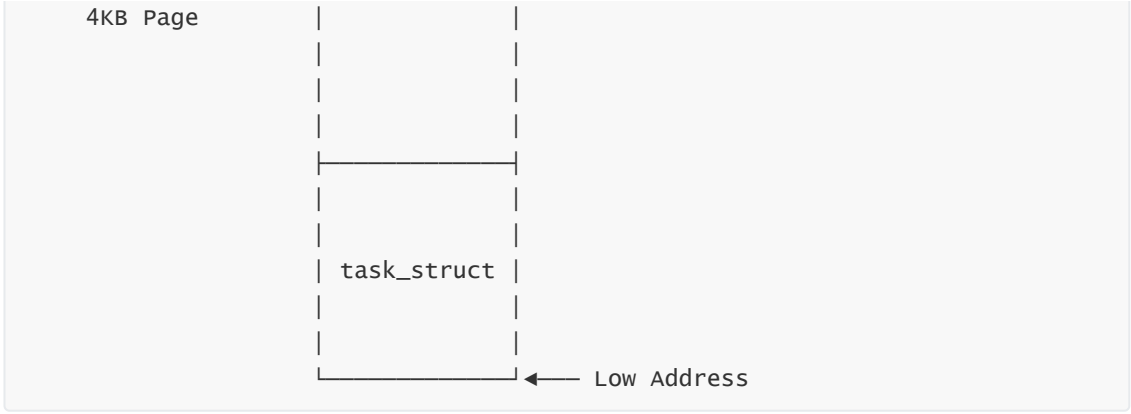
```
    uint64 kernel_sp;
    uint64 user_sp;
};

/* 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();
```

4.3 线程调度功能实现

4.3.1 线程初始化

1. 在初始化线程的时候，参考Linux v0.11中的实现为每个线程分配一个 4KB 的物理页，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：





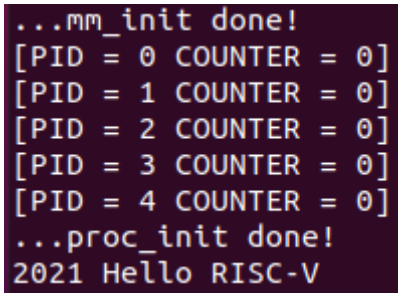
2. 当我们的 OS run 起来时候，其本身就是一个线程 `idle 线程`，但是我们并没有为它设计好 `task_struct`。所以第一步我们要为 `idle` 设置 `task_struct`。并将 `current`, `task[0]` 都指向 `idle`。该部分的初始化代码如下,这里要注意的的是为`idle`分配空间：

```
// 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 taks[0] 指向 idle
idle = (struct task_struct*)kalloc();
idle->state = TASK_RUNNING;
idle->counter = 0;
idle->priority = 0;
idle->pid = 0;
current = idle;
task[0] = idle;
```

3. 为了方便起见，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`，该部分代码如下：

```
// 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

for( int i = 1; i < NR_TASKS; i++ ){
    struct task_struct* tmp = (struct task_struct*)kalloc();
    tmp->state = TASK_RUNNING;
    tmp->counter = 0;
    tmp->priority = rand();
    tmp->pid = i;
    struct thread_struct thread_tmp;
    thread_tmp.ra = (uint64)__dummy;
    thread_tmp.sp = PGROUNDUP((uint64)tmp);
    tmp->thread = thread_tmp;
    task[i] = tmp;
}
```

4. 在调用线程初始化之前需要先调用`mm_init`函数所以需要在`head.S`中的`call mm_init`后面调用，增加一些`debug`信息，运行后可得到以下结果，说明该部分初始化已完成：



4.3.2 _dummy与dummy

1. `task[1]` ~ `task[NR_TASKS - 1]` 都运行同一段代码 `dummy()` 我们在 `proc.c` 添加 `dummy()`：

```
// arch/riscv/kernel/proc.c

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;
            printk("[PID = %d] is running. auto_inc_local_var = %d\n",
current->pid, auto_inc_local_var);
        }
    }
}
```

2. 在 `entry.S` 添加 `__dummy`
   - 在`__dummy` 中将 sepc 设置为 `dummy()` 的地址, 并使用 `sret` 从中断中返回。
   - `__dummy` 与 `_traps`的 `restore`部分相比， 其实就是省略了从栈上恢复上下文的过程 （但是手动设置了 `sepc` ），添加如下代码：

```
     .global __dummy
__dummy:
    la  t0, dummy  #获取dummy地址
    csrrw x0, sepc, t0  #将地址给sepc
    sret          #从中断返回
```

4.3.3 实现线程切换

1. `switch_to` **函数。**判断下一个执行的线程 `next` 与当前的线程 `current` 是否为同一个线程，如果是同一个线程，则无需做任何处理，否则调用 `__switch_to` 进行线程切换，**这里需要注意的是，在调用**`__switch_to` **之前，需要先将当前线程设置为需要切换到的线程**，代码如下：

```
void switch_to(struct task_struct* next) {
    if(current->pid == next->pid)return; //如果是同一个线程则返回
    //输出切换的线程信息
    printk("switch to [PID = %d COUNTER = %d]\n",next->pid, next->counter);
    struct task_struct* tmp = current;
    current = next;  //将当前线程切换到需要切换到线程
    __switch_to(tmp, current);
    return;
}
```

2. `__switch_to`**函数**，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

   首先计算`thread_struct`的偏移地址，前面有一个指针，四个64位变量，根据地址对齐，起始地址为相对于`a0`偏移40，该部分代码如下：

```
__switch_to:
    sd ra, 40(a0)    #储存原线程信息
    sd sp, 48(a0)
    sd s0, 56(a0)
    sd s1, 64(a0)
    sd s2, 72(a0)
    sd s3, 80(a0)
    sd s4, 88(a0)
    sd s5, 96(a0)
    sd s6, 104(a0)
    sd s7, 112(a0)
    sd s8, 120(a0)
    sd s9, 128(a0)
    sd s10, 136(a0)
    sd s11, 144(a0)

    ld ra, 40(a1)    #加载新线程信息
    ld sp, 48(a1)
    ld s0, 56(a1)
    ld s1, 64(a1)
    ld s2, 72(a1)
    ld s3, 80(a1)
    ld s4, 88(a1)
    ld s5, 96(a1)
    ld s6, 104(a1)
    ld s7, 112(a1)
    ld s8, 120(a1)
    ld s9, 128(a1)
    ld s10, 136(a1)
    ld s11, 144(a1)
    ret
```

4.3.4 实现调度入口函数

实现 `do_timer()`，并在 `时钟中断处理函数trap()` 中调用，该部分代码如下：

- `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->pid == 0 ){
        printk("idle process is running!\n\n");
        schedule();
    }
    else{
        if(--(current->counter) > 0){
```

```
        //printk("%d\n",current->counter);
            return;//dummy();
            }
        else {
            printk("\n");
            schedule();
        }
    }
}
```

- `trap()`

```
#include "printk.h"
#include "sbi.h"
#include "proc.h"
extern void clock_set_next_event();
void trap_handler(unsigned long scause, unsigned long sepc) {
    // 通过 `scause` 判断trap类型
    // 如果是interrupt 判断是否是timer interrupt
    // 如果是timer interrupt 则打印输出相关信息, 并通过 `clock_set_next_event()`
设置下一次时钟终端
    // `clock_set_next_event()` 见 4.5 节
    // 其他interrupt / exception 可以直接忽略
    // YOUR CODE HERE
    if(scause == 0x8000000000000005){
            //printk("trap\n");
            clock_set_next_event();
            do_timer();

    }
    return;
}
```

4.3.5 实现线程调度

4.3.5.1 短作业优先调度算法

当需要进行调度时按照以下规则进行调度：

- 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
- 该部分代码如下：

```
void schedule(void) {
    //printk("schedule\n");
    int id = 0;
    uint64 min = 100;
    //找到最短的作业
    for( int i = 1; i < NR_TASKS; i++ ){
        if( task[i]->counter > 0 && task[i]->counter < min ){
            id = i;
            min = task[i]->counter;
        }
    }
```

```
    //初始化
    if( id == 0 ){
        for( int i = 1; i < NR_TASKS; i++ ){
            task[i]->counter = rand(); //初始化counter
            printk("SET [PID = %d COUNTER = %d]\n",i,task[i]->counter);
        }
        printk("\n");
        //for( int i = 1; i < NR_TASKS; i++ ) printk("%d",task[i]->counter);
        schedule();
    }else{
        switch_to(task[id]); //切换到最短的作业
    }
    return;
}
```

4.3.5.2 优先级调度算法

按照以下规则进行调度：

- 随机初始化所有线程的`priority`，并将`counter`设置为`priority`
- 每次调度`counter`不为0，且`priority`最大的线程
- 均为0后重新将`counter`设置为`priority`
- 该部分代码如下：

```
void schedule(void) {
    int id = 0;
    uint64 max = 0;
    //找到counter不为0中最大的优先级
    for( int i = 1; i < NR_TASKS; i++ ){
        if( task[i]->counter > 0 && task[i]->priority > max ){
            id = i;
            max = task[i]->priority;
        }
    }
    //初始化
    if( id == 0 ){
        for( int i = 1; i < NR_TASKS; i++ ){
            task[i]->counter = task[i]->priority;
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n",i,task[i]-
>priority,task[i]->counter);
        }
        printk("\n");
        //for( int i = 1; i < NR_TASKS; i++ ) printk("%d",task[i]->counter);
        schedule();
    }else{
        switch_to(task[id]); //调度
    }
    return;
}
```

4.3.5.3 综合

综合以上代码，可得到以下结果：

```
void schedule(void) {
    //printk("schedule\n");
    #ifdef SJF
        int id = 0;
        uint64 min = 100;
        for( int i = 1; i < NR_TASKS; i++ ){
            if( task[i]->counter > 0 && task[i]->counter < min ){
                id = i;
                min = task[i]->counter;
            }
        }
        if( id == 0 ){
            for( int i = 1; i < NR_TASKS; i++ ){
            task[i]->counter = rand();
            printk("SET [PID = %d COUNTER = %d]\n",i,task[i]->counter);
            }
            printk("\n");
            //for( int i = 1; i < NR_TASKS; i++ ) printk("%d",task[i]->counter);
            schedule();
        }else{
            switch_to(task[id]);
        }
    #endif
    #ifdef PRIORITY
        int id = 0;
        uint64 max = 0;
        for( int i = 1; i < NR_TASKS; i++ ){
            if( task[i]->counter > 0 && task[i]->priority > max ){
                id = i;
                max = task[i]->priority;
            }
        }
        if( id == 0 ){
            for( int i = 1; i < NR_TASKS; i++ ){
            task[i]->counter = task[i]->priority;
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n",i,task[i]-
>priority,task[i]->counter);
            }
            printk("\n");
            //for( int i = 1; i < NR_TASKS; i++ ) printk("%d",task[i]->counter);
            schedule();
        }else{
            switch_to(task[id]);
        }
    #endif
    return;
}
```

4.4 编译及测试

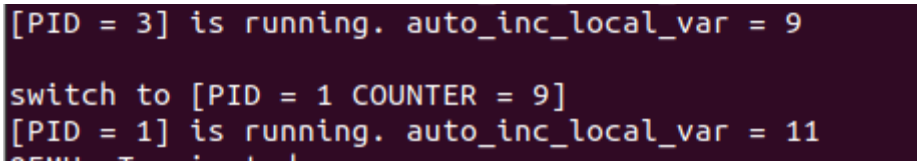
4.4.1 SJF

在`Makefile`中添加：

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

运行后可得到以下结果：

```
...proc_init done!
2021 Hello RISC-V
idle process is running!

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

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

switch to [PID = 3 COUNTER = 5]
[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

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
[PID = 1] is running. auto_inc_local_var = 10

switch to [PID = 2 COUNTER = 10]
[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
[PID = 2] is running. auto_inc_local_var = 10

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

[PID = 2] is running. auto_inc_local_var = 11
[PID = 2] is running. auto_inc_local_var = 12
[PID = 2] is running. auto_inc_local_var = 13
[PID = 2] is running. auto_inc_local_var = 14

switch to [PID = 3 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 6
[PID = 3] is running. auto_inc_local_var = 7
[PID = 3] is running. auto_inc_local_var = 8
```



4.4.1 PRIORITY

在`Makefile`中添加：

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

运行后可得到以下结果：

```
...mm_init done!
[PID = 0 COUNTER = 0]
[PID = 1 COUNTER = 0]
[PID = 2 COUNTER = 0]
[PID = 3 COUNTER = 0]
[PID = 4 COUNTER = 0]
...proc_init done!
2021 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]

switch to [PID = 3 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
[PID = 3] is running. auto_inc_local_var = 6
[PID = 3] is running. auto_inc_local_var = 7
[PID = 3] is running. auto_inc_local_var = 8
[PID = 3] is running. auto_inc_local_var = 9
[PID = 3] is running. auto_inc_local_var = 10

switch to [PID = 2 COUNTER = 4]
[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 = 4]
[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

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

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]

switch to [PID = 3 COUNTER = 10]
[PID = 3] is running. auto_inc_local_var = 11
[PID = 3] is running. auto_inc_local_var = 12
[PID = 3] is running. auto_inc_local_var = 13
[PID = 3] is running. auto_inc_local_var = 14
[PID = 3] is running. auto_inc_local_var = 15
```

5 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？

   本质上，`context_switch` 仍然是一个普通函数，需要符合 `RISC-V` 标准过程调用文档。`RISC-V` 规定，`x8~x9,x18~x27` 是属于callee-saved registers，也就是说，在 `switch_to` 函数调用 `__switch_to` 函数这个过程中， `__switch_to` 函数要保证这些寄存器值是和调用 `__switch_to` 函数之前一模一样的。除此之外，线程有自己独立的栈，所以栈指针 `sp` 也需要保存，还有自己独立的 `pc`，保存在 `ra` 中，共14个，其他寄存器由调用函数来保存即可。

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。

   调用完一次之后，`ra` 保存的值为 `0x802000d4<_traps+152>`，debug过程如下所示：

   ```
   (gdb) b __dummy
   Breakpoint 2 at 0x8020016c
   (gdb) c
   Continuing.

   Breakpoint 2, 0x000000008020016c in __dummy ()
   (gdb) si
   0x0000000080200170 in __dummy ()
   0x0000000080200174 in __switch_to ()
   (gdb) i r ra
   ra             0x802000d4       0x802000d4 <_traps+152>
   ```

   查看 `ra` 保存的地址的指令，如下所示：

   ```
   0x802000d0 <_traps+148> jal     ra,0x80200808 <trap_handler>
   0x802000d4 <_traps+152> ld      t0,248(sp)
   0x802000d8 <_traps+156> csrw    sepc,t0
   0x802000dc <_traps+160> ld      zero,240(sp)
   0x802000e0 <_traps+164> ld      ra,232(sp)
   0x802000e4 <_traps+168> ld      gp,224(sp)
   ```

   发现 `ra` 为 `traps` 函数调用了中断处理函数之后，而在 `__dummy` 中，我们已经将 `dummy()` 的地址存到 `sepc` 中了，在进行 `context_switch` 后，跳转到该线程对应的 `dummy()` 继续执行。

6 遇到的问题

1. 最开始没有注意到需要在 `_start` 的适当位置调用 `mm_init`，来初始化内存管理系统这句话，不知从何下手，卡了很久，后来发现了问题，并利用 `debug` 追踪才找到了应该放在初始化了 `sp` 指针后；
2. 在实现 `switch_to` 函数时，判断完不相等后，直接调用 `__switch_to` 函数，发现 `sret` 会直接跳过，后来发现是没有切换线程，所以需要在实现 `switch_to` 函数时先切换好线程；
3. 在 `makefile` 修改时，一开始增加 `-DSJF` 发现程序出问题了，查找解决方案后，发现 `-D` 是 `-define` 的简写，所以要在中间加上空格。