实验六-RV64 内核线程调度

姓名:王晶晶 学号:3200104880 学院:计算机科学与技术学院 课程名称：计算机系统Ⅱ

实验时间：2021.12.23 实验地点：紫金港机房中心 指导老师：申文博

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、实验内容和原理

1、进程与线程

源代码经编译器一系列处理（编译、链接、优化等）后得到的可执行文件，称之为程序（Program）。而通俗地说，进程就是正在运行并使用计算机资源的程序。进程与程序的不同之处在于，进程是一个动态的概念，其不仅需要将其运行的程序的代码/数据等加载到内存空间中，还需要拥有自己的运行栈。同时一个进程可以对应一个或多个线程，线程之间往往具有相同的代码，共享一块内存，但是却有不同的CPU执行状态。
在本次实验中，为了简单起见， 采用 single-threaded process 模型， 即一个进程对应一个线程，进程与线程不做明显区分。

2、线程相关属性

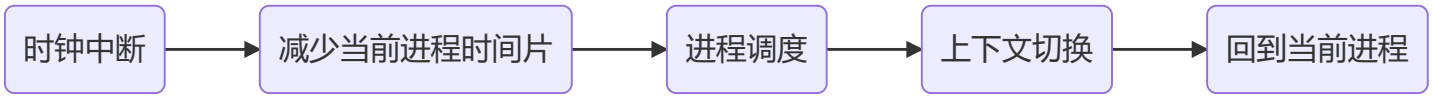
在不同的操作系统中，为每个线程所保存的信息都不同。本次实验中，每个线程会包括：

- 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、线程切换

- 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，至此就完成了线程的调度与切换。



三、主要仪器设备

- Docker in Lab3

四、操作方法与实现步骤

4.1 实验步骤

- 引入proc、mm、rand等函数与头文件
- 实现线程调度功能
  - 线程初始化：
    - 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。
    - 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，并为这些线程设置 `thread_struct` 中的 `ra` 和 `sp`

        - 在 _start 适当的位置调用 task_init
    - 设置进程入口__dummy 与 dummy
        - 在 proc.c 添加 dummy()
        - 为线程第一次调度提供一个特殊的返回函数 __dummy ,并在 entry.S 中添加 __dummy
- 实现线程切换
    - 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。
    - 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
    - 实现调度入口函数 do_timer() ，并在时钟中断中调度
    - 实现调度算法
        - 短作业优先调度算法
        - 优先级调度算法
- 编辑 Makefile 及 proc.c 文件，实现用宏定义控制两种算法的编译运行

4.2 实验结果

4.2.1 head.S

```
    .section .text.entry
    .align 2
    .globl _traps
    .globl __dummy
    .globl __switch_to
    .extern trap_handler
    .extern dummy
    .extern clock_set_next_event
_traps:
    # YOUR CODE HERE
    # -----------

        # 1. save 32 registers and sepc to stack
     sd sp, -8(sp)
     sd ra, -16(sp)
     sd gp, -24(sp)
     sd tp, -32(sp)
     sd t0, -40(sp)
     sd t1, -48(sp)
     sd t2, -56(sp)
     sd s0, -64(sp)
     sd s1, -72(sp)
     sd a0, -80(sp)
     sd a1, -88(sp)
     sd a2, -96(sp)
     sd a3, -104(sp)
     sd a4, -112(sp)
     sd a5, -120(sp)
     sd a6, -128(sp)
     sd a7, -136(sp)
     sd s2, -144(sp)
     sd s3, -152(sp)
     sd s4, -160(sp)
     sd s5, -168(sp)
     sd s6, -176(sp)
     sd s7, -184(sp)
     sd s8, -192(sp)
     sd s9, -200(sp)
     sd s10, -208(sp)
     sd s11, -216(sp)
     sd t3, -224(sp)
     sd t4, -232(sp)
     sd t5, -240(sp)
     sd t6, -248(sp)
     csrr t0,sepc
```

```
 sd t0,-256(sp)
 addi sp,sp,-256

# -----------

    # 2. call trap_handler
 csrr a0,scause
 csrr a1,sepc
 call trap_handler

# -----------

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
 ld t0,0(sp)
 csrw sepc,t0
 ld t6,8(sp)
 ld t5,16(sp)
 ld t4,24(sp)
 ld t3,32(sp)
 ld s11,40(sp)
 ld s10,48(sp)
 ld s9,56(sp)
 ld s8,64(sp)
 ld s7,72(sp)
 ld s6,80(sp)
 ld s5,88(sp)
 ld s4,96(sp)
 ld s3,104(sp)
 ld s2,112(sp)
 ld a7,120(sp)
 ld a6,128(sp)
 ld a5,136(sp)
 ld a4,144(sp)
 ld a3,152(sp)
 ld a2,160(sp)
 ld a1,168(sp)
 ld a0,176(sp)
 ld s1,184(sp)
 ld s0,192(sp)
 ld t2,200(sp)
 ld t1,208(sp)
 ld t0,216(sp)
 ld tp,224(sp)
 ld gp,232(sp)
 ld ra,240(sp)
```

```
     ld sp,248(sp)

    # -----------

        # 4. return from trap
     sret
    # -----------
__dummy:
    la t0,dummy
    csrw sepc,t0
    sret
__switch_to:
    #unable the time interrupt
    #csrr t1,sie
    #andi t1,t1,0xFFFFFFFFFFFFFFDF
    #csrw sie,t1
    #save state to prev process
    addi t0,a0,40
    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

    addi t0,a1,40
    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
```

4.2.2 proc.c

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

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

extern void __dummy();
extern void __switch_to(struct task_struct* prev, struct task_struct* next);

struct task_struct* idle;           // idle process
struct task_struct* current;        //point to the current process
struct task_struct* task[NR_TASKS]; // store all processes

void task_init() {

    uint64 i;

    idle=(struct task_struct*)kalloc();
    idle->thread.sp=(uint64)idle+PGSIZE;
    idle->state=TASK_RUNNING;
    idle->counter=0;
    idle->priority=0;
    idle->pid=0;

    current=idle;
    task[0]=idle;

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

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

void switch_to(struct task_struct* next) {
    struct task_struct *tmp=current;
    if(current==next);
    else{
        printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n",next->pid,next->pr
        current=next;
        __switch_to(tmp,next);
    }
    return;
}

void do_timer(void) {
    if((current==idle)||(!(--current->counter)))
        schedule();
    return;
}
//shortest-job first
#ifdef SJF
int Findtask(void){
    uint64 i,index=-1;
    for(i=1;i<NR_TASKS;i++){
        if((task[i]->counter)&&(task[i]->state==TASK_RUNNING)){
            if((index==-1)||(task[i]->counter<task[index]->counter))
                index=i;
        }
    }
    return index;
}
#endif
//priority
#ifdef PRIORITY
int Findtask(void){
    uint64 i,index=-1;
    for(i=1;i<NR_TASKS;i++){
```

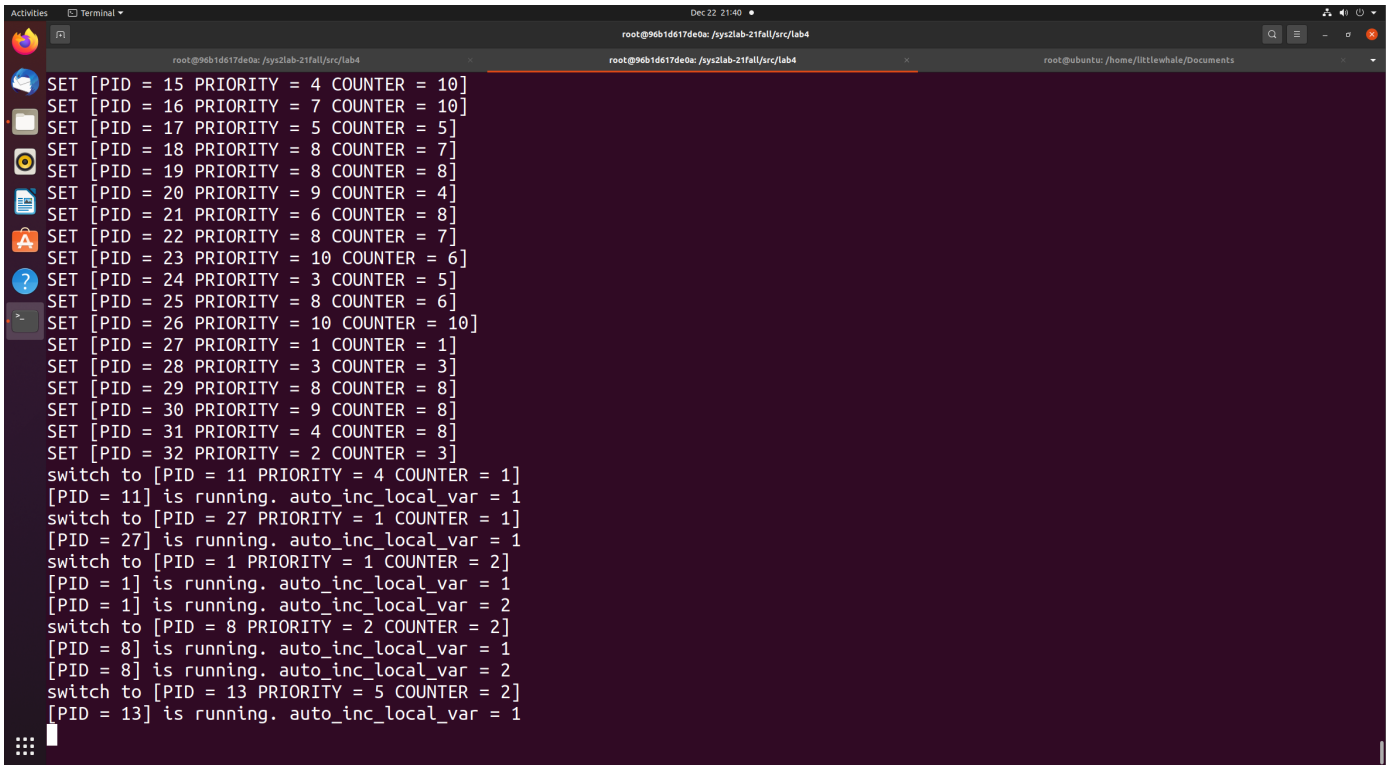
```
        if((task[i]->counter)&&(task[i]->state==TASK_RUNNING)){
            if((index==-1)||(task[i]->priority>task[index]->priority))
                index=i;
        }
    }
   return index;
}
#endif
void schedule(void){
    uint64 index,i;
    uint64 cnt=0;
    while((index=Findtask())==-1){
        for(i=1;i<NR_TASKS;i++){
            if(task[i]->state==TASK_RUNNING){
                task[i]->counter=(uint64)rand();
            }
        }
        cnt++;
    }
    if(cnt){
        for(i=1;i<NR_TASKS;i++){
            printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n",task[i]->pid,task[i]
        }
    }
    //printk("%x",(uint64)task[index]);
    //before=current;
    //current=task[index];
    switch_to(task[index]);
    return;
}
```

4.2.3 arch/riscv/kernel/Makefile

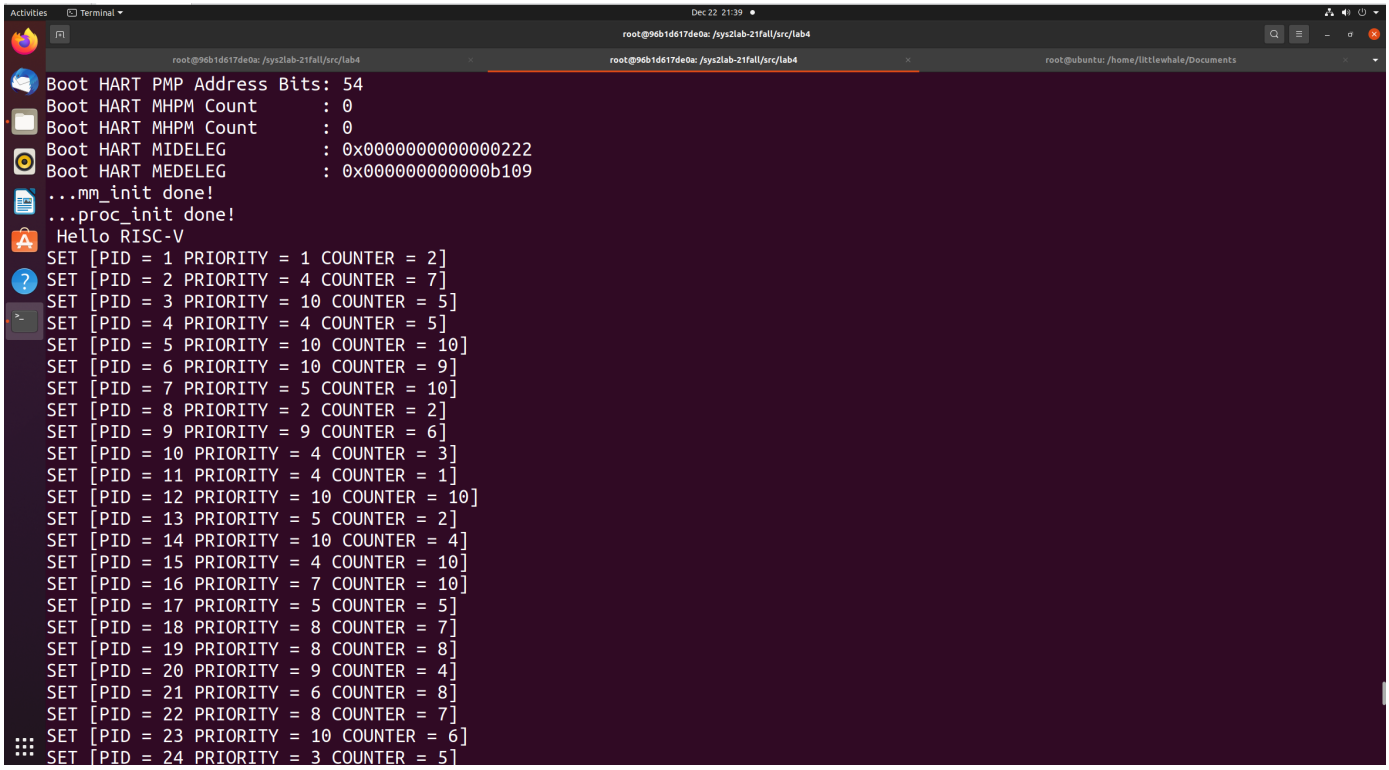
```
ASM_SRC         = $(sort $(wildcard *.S))
C_SRC       = $(sort $(wildcard *.c))
OBJ                 = $(patsubst %.S,%.o,$(ASM_SRC)) $(patsubst %.c,%.o,$(C_SRC))

CFLAG = ${CF}${INCLUDE} -DPRIORITY
# CFLAG = ${CF}${INCLUDE} -DSJF

all:$(OBJ)


%.o:%.S
        ${GCC}  ${CFLAG} -c $<

%.o:%.c
        ${GCC}  ${CFLAG} -c $<

clean:
        $(shell rm *.o 2>/dev/null)
```

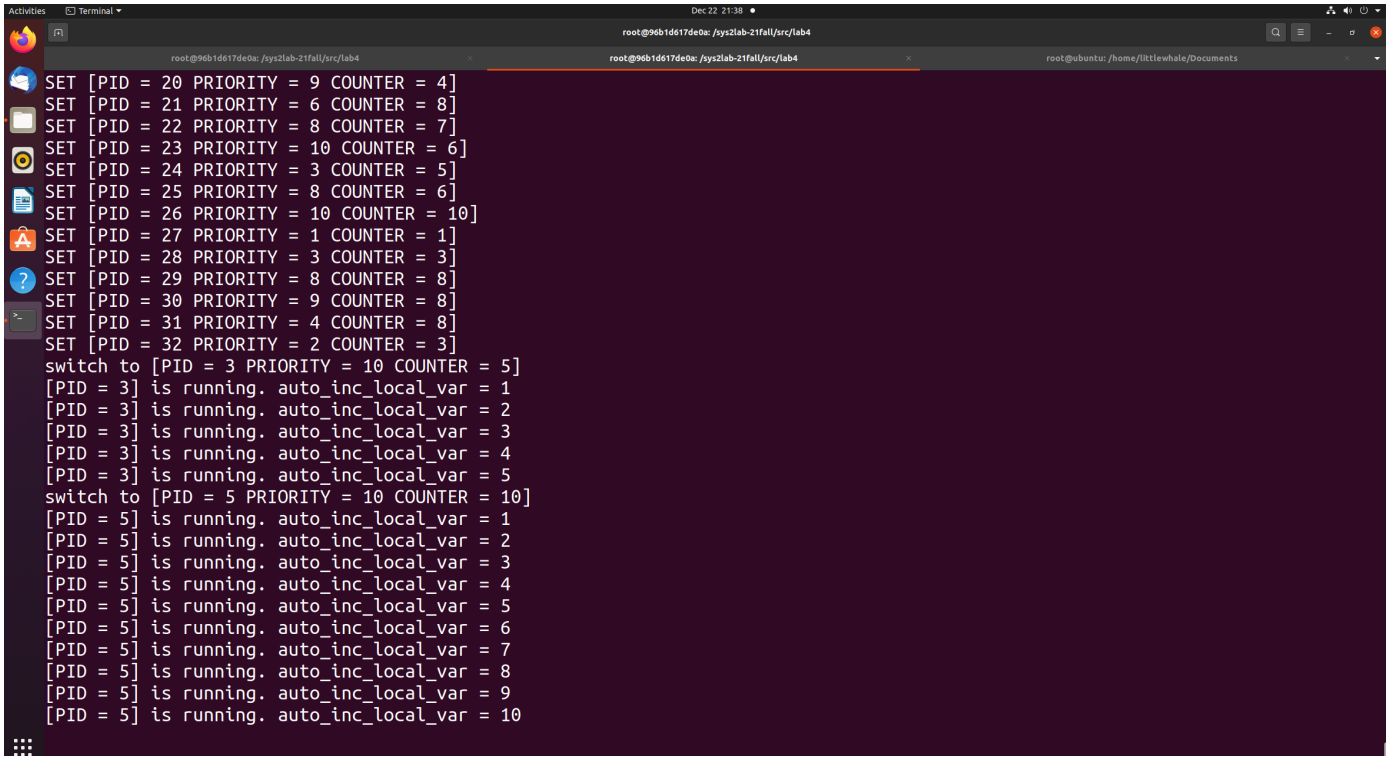
4.4.4 make run

- SJF



- PRIORITY





五、思考与心得

- 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?

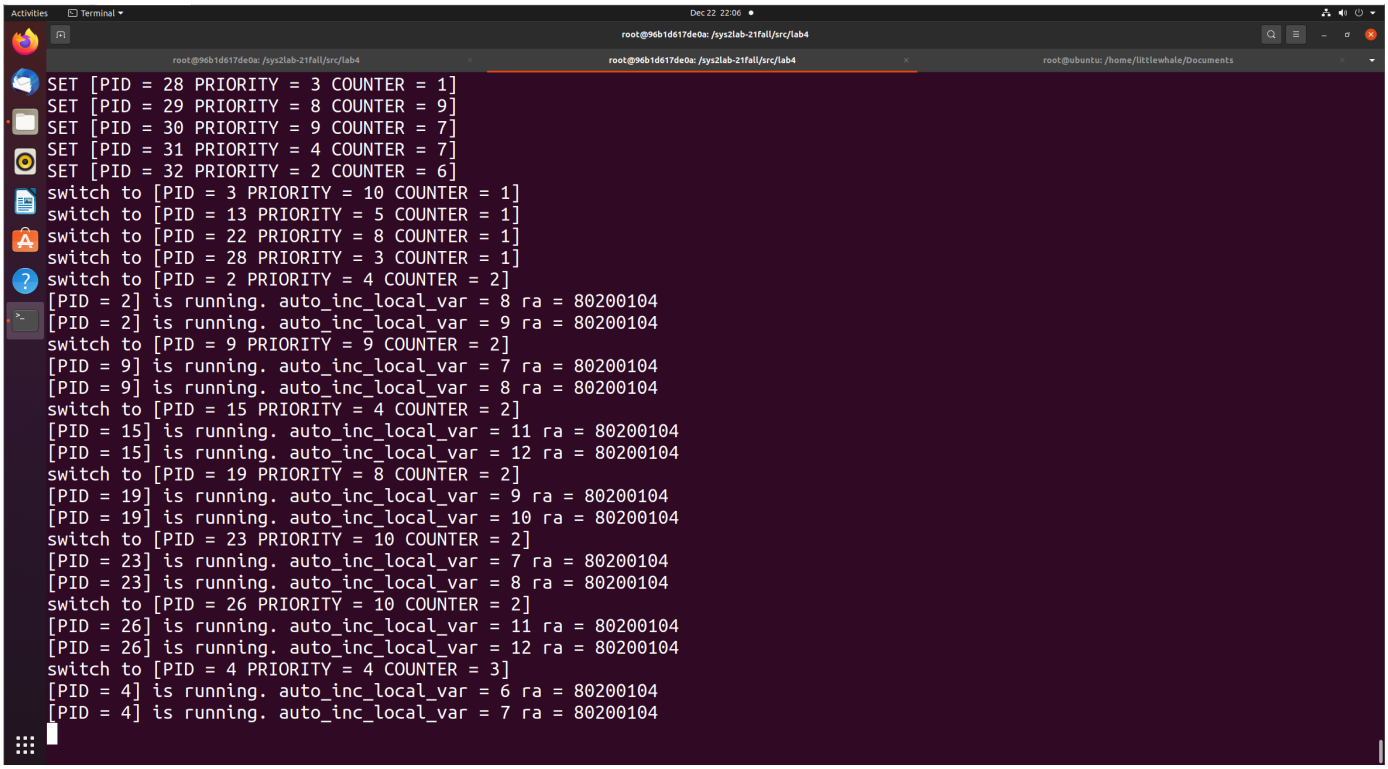
  时钟中断发生时，进入中断处理程序实现进程的调度。在上下文切换时，只要把sp、ra恢复，上下文切换完成后，即可返回到中断处理入口程序，重新恢复进程保存在

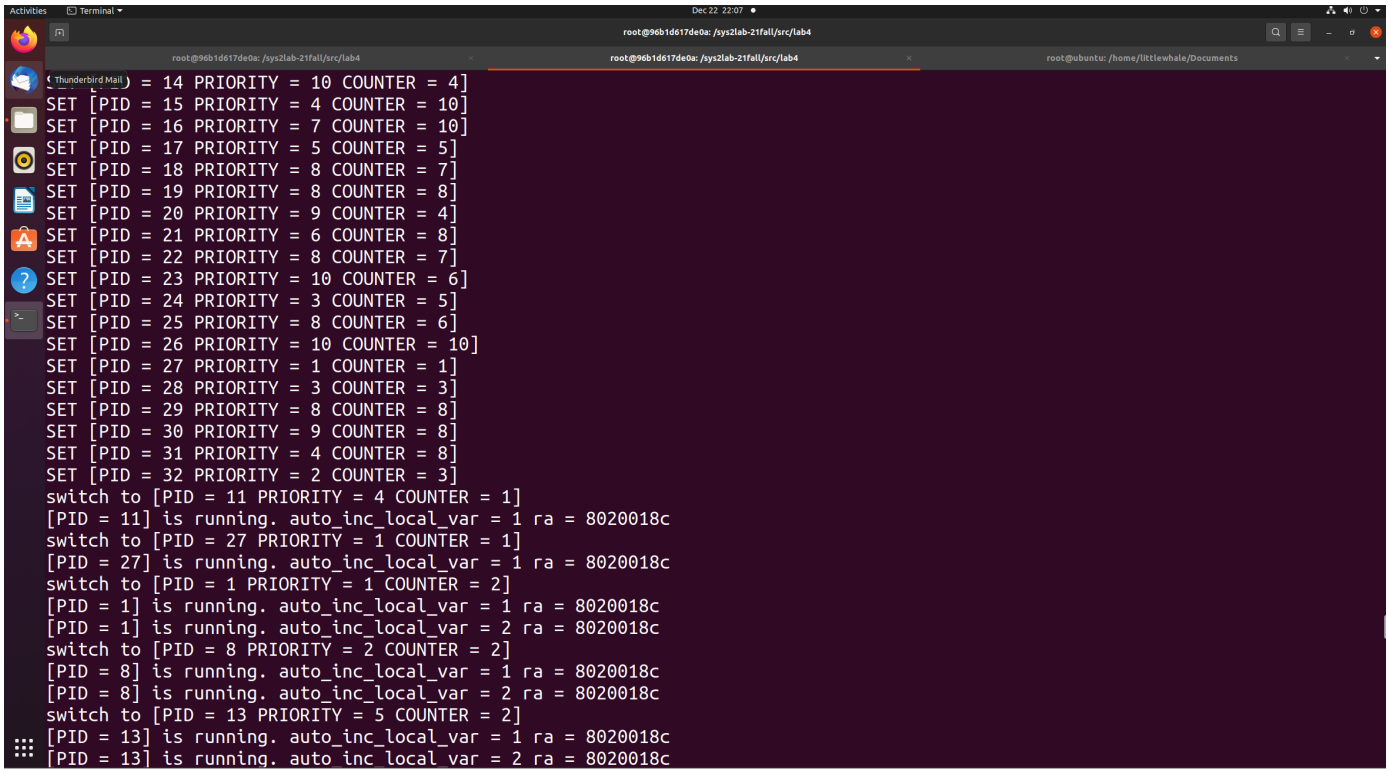
栈上的其它相关参数。故而其它寄存器如临时变量寄存器等在上下文切换时候不是必要保存的。

同时根据RV64的寄存器约定，在context switch所必须保存的寄存器均为被调用者必须保存的寄存器。

- 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。

  ra会变成_traps的第一个恢复栈语句。




Activities
Terminal
Dec 22 22:07
root@96b1d617de0a: /sys2lab-21fall/src/lab4
root@96b1d617de0a: /sys2lab-21fall/src/lab4
root@96b1d617de0a: /sys2lab-21fall/src/lab4
root@ubuntu: /home/littlewhale/Documents
Thunderbird Mail
) = 14 PRIORITY = 10 COUNTER = 4]
SET [PID = 15 PRIORITY = 4 COUNTER = 10]
SET [PID = 16 PRIORITY = 7 COUNTER = 10]
SET [PID = 17 PRIORITY = 5 COUNTER = 5]
SET [PID = 18 PRIORITY = 8 COUNTER = 7]
SET [PID = 19 PRIORITY = 8 COUNTER = 8]
SET [PID = 20 PRIORITY = 9 COUNTER = 4]
SET [PID = 21 PRIORITY = 6 COUNTER = 8]
SET [PID = 22 PRIORITY = 8 COUNTER = 7]
SET [PID = 23 PRIORITY = 10 COUNTER = 6]
SET [PID = 24 PRIORITY = 3 COUNTER = 5]
SET [PID = 25 PRIORITY = 8 COUNTER = 6]
SET [PID = 26 PRIORITY = 10 COUNTER = 10]
SET [PID = 27 PRIORITY = 1 COUNTER = 1]
SET [PID = 28 PRIORITY = 3 COUNTER = 3]
SET [PID = 29 PRIORITY = 8 COUNTER = 8]
SET [PID = 30 PRIORITY = 9 COUNTER = 8]
SET [PID = 31 PRIORITY = 4 COUNTER = 8]
SET [PID = 32 PRIORITY = 2 COUNTER = 3]
switch to [PID = 11 PRIORITY = 4 COUNTER = 1]
[PID = 11] is running. auto_inc_local_var = 1 ra = 8020018c
switch to [PID = 27 PRIORITY = 1 COUNTER = 1]
[PID = 27] is running. auto_inc_local_var = 1 ra = 8020018c
switch to [PID = 1 PRIORITY = 1 COUNTER = 2]
[PID = 1] is running. auto_inc_local_var = 1 ra = 8020018c
[PID = 1] is running. auto_inc_local_var = 2 ra = 8020018c
switch to [PID = 8 PRIORITY = 2 COUNTER = 2]
[PID = 8] is running. auto_inc_local_var = 1 ra = 8020018c
[PID = 8] is running. auto_inc_local_var = 2 ra = 8020018c
switch to [PID = 13 PRIORITY = 5 COUNTER = 2]
[PID = 13] is running. auto_inc_local_var = 1 ra = 8020018c
[PID = 13] is running. auto_inc_local_var = 2 ra = 8020018c