

# 实验6——RV64 内核线程调度

姓名：汤尧

学号：3200106252

课程名称：计算机系统II

指导老师：申文博

## 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

- Docker in Lab3

## 3 背景知识

### 3.1 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，我们称之为程序（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CPU执行状态。

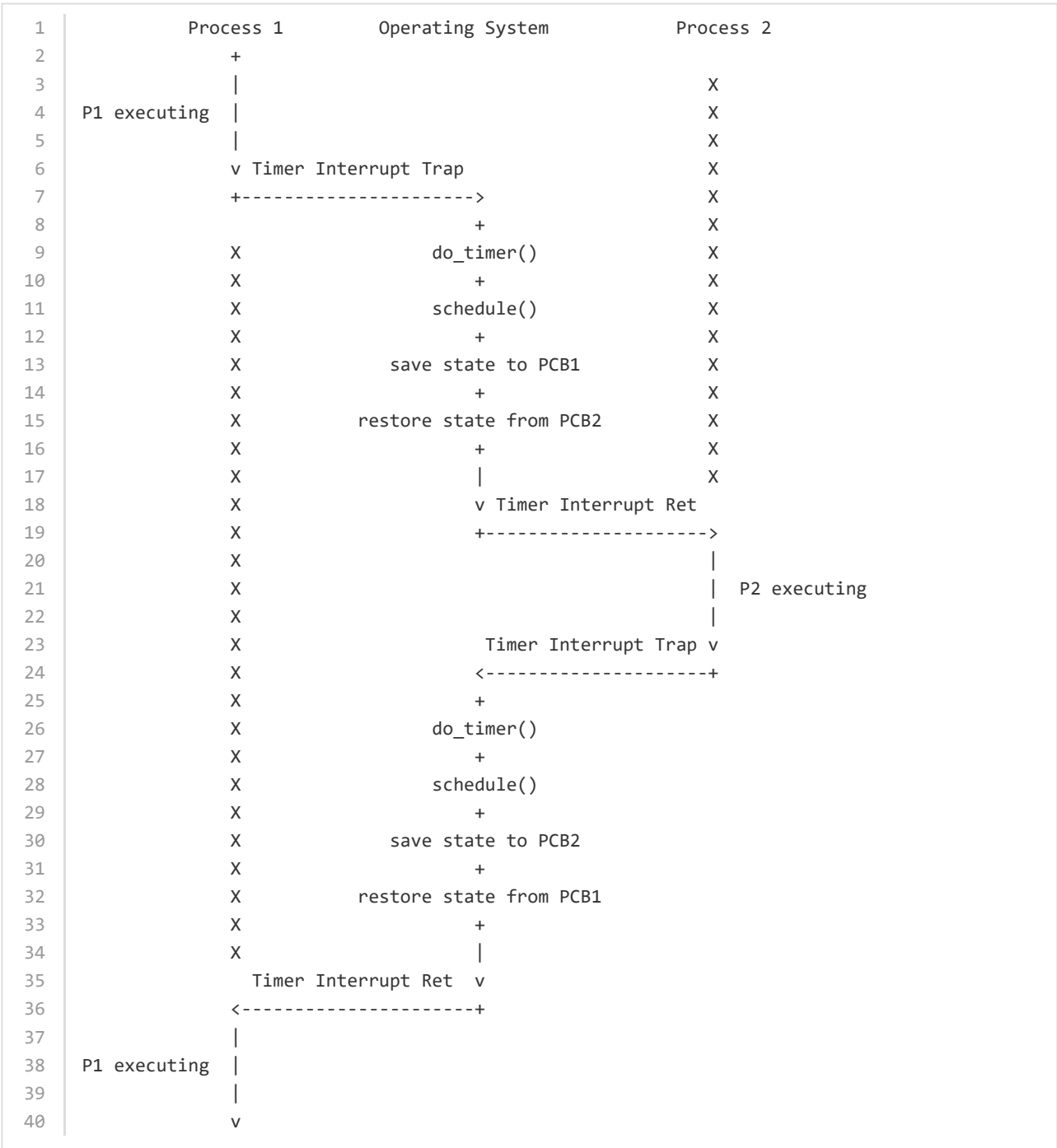
在本次实验中，为了简单起见，我们采用 single-threaded process 模型，即一个进程对应一个线程，进程与线程不做明显区分。

### 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

- 此次实验基于 lab5 所实现的代码进行。
- 从 repo 同步以下代码: rand.h/rand.c, string.h/string.c, mm.h/mm.c。并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。
- 在 lab6 中我们需要一些物理内存管理的接口, 在此我们提供了 kalloc 接口 (见 mm.c) 给同学。同学可以用 kalloc 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:

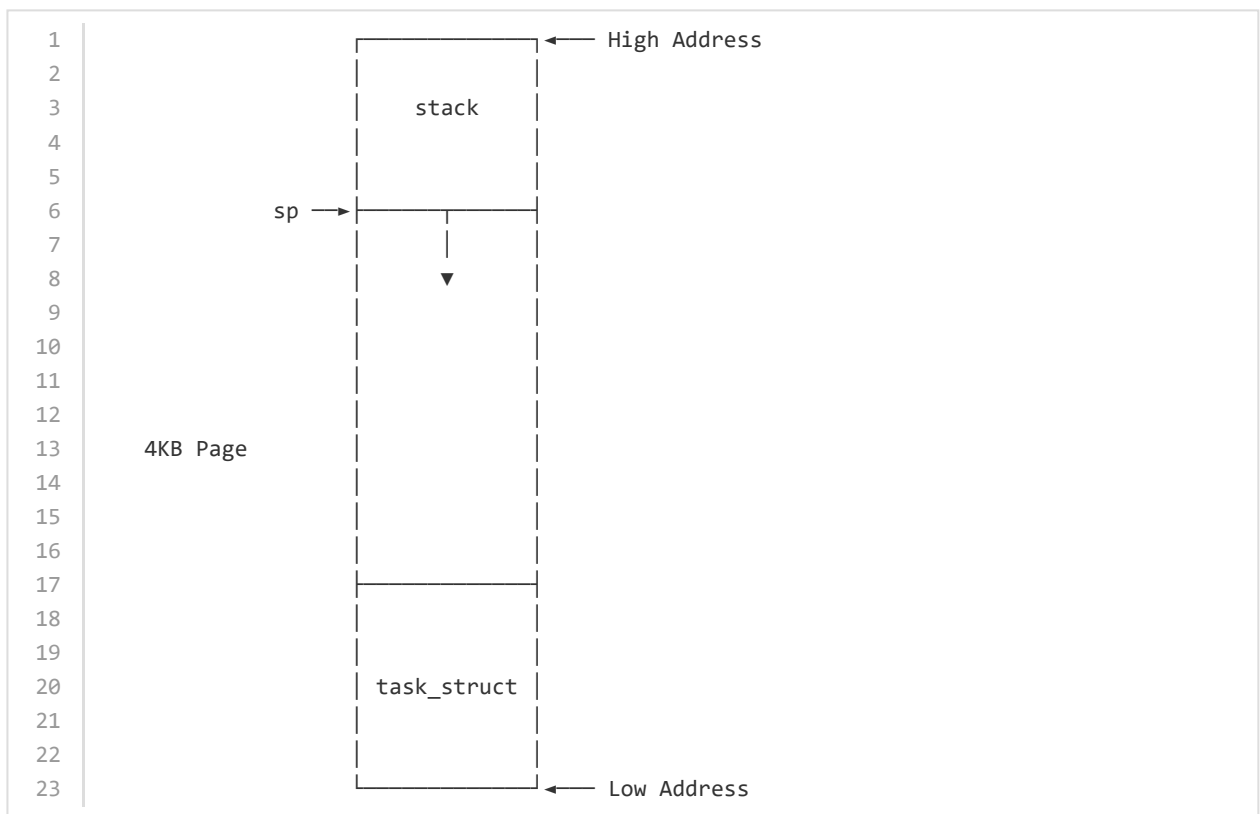
```
1 #define PHY_START 0x0000000080000000
2 #define PHY_SIZE 128 * 1024 * 1024 // 128MB, QEMU 默认内存大小
3 #define PHY_END (PHY_START + PHY_SIZE)
4
5 #define PGSIZE 0x1000 // 4KB
6 #define PGROUNDUP(addr) ((addr + PGSIZE - 1) & ~(PGSIZE - 1))
7 #define PGROUNDDOWN(addr) (addr & ~(PGSIZE - 1))
```

■

## 4.3 线程调度功能实现

### 4.3.1 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。具体内存布局如下图所示:



- 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。

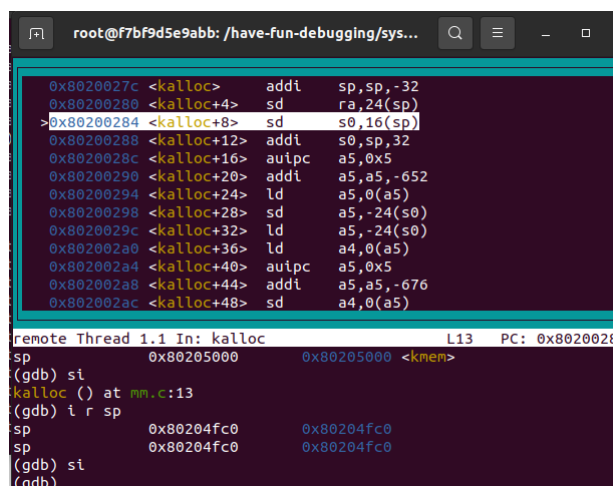
```

1 void task_init() {
2     // 1. 调用 kalloc() 为 idle 分配一个物理页
3     // 2. 设置 state 为 TASK_RUNNING;
4     // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
5     // 4. 设置 idle 的 pid 为 0
6     // 5. 将 current 和 task[0] 指向 idle
7     idle = kalloc();
8     idle->state = TASK_RUNNING;
9     idle->counter=0;
10    idle->priority=0;
11    idle->pid=0;
12    current=idle;
13    task[0]=idle;
14
15    // 1. 参考 idle 的设置，为 task[1] ~ task[NR_TASKS - 1] 进行初始化
16    // 2. 其中每个线程的 state 为 TASK_RUNNING, counter 为 0,
17    // priority 使用 rand() 来设置, pid 为该线程在线程数组中的下标。
18    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct` 中的 `ra` 和 `sp`,
19    // 4. 其中 `ra` 设置为 __dummy （见 4.3.2）的地址，
20    // `sp` 设置为 该线程申请的物理页的高地址
21
22    for(int i=1;i<NR_TASKS;i++){
23        idle = kalloc();
24        idle->state = TASK_RUNNING;
25        idle->counter=0;
26        idle->priority=rand();
27
28        idle->pid=i;
29        task[i]=idle;
30        idle->thread.ra=__dummy;
31        idle->thread.sp=PGSIZE+((uint64)idle);
32    }
33    idle = task[0];
34    printk("...proc_init done!\n");
35 }

```

- 为了方便起见，我们将 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。
- 在 \_start 适当的位置调用 `task\_init`

## 遇到bug及解决方案



The screenshot shows a GDB session with a memory dump and a stack trace. The memory dump displays addresses from 0x8020027c to 0x802002ac, with corresponding assembly instructions and their operands. The stack trace shows the current function as kalloc, with the stack pointer (sp) at 0x80205000 and the return address (ra) at 0x80204fc0.

```

root@f7bf9d5e9abb: /have-fun-debugging/sys...
0x8020027c <kalloc> addl sp,sp,-32
0x80200280 <kalloc+4> sd ra,24(sp)
->0x80200284 <kalloc+8> sd s0,16(sp)
0x80200288 <kalloc+12> addl s0,sp,32
0x8020028c <kalloc+16> auipc a5,0x5
0x80200290 <kalloc+20> addl a5,a5,-652
0x80200294 <kalloc+24> ld a5,0(a5)
0x80200298 <kalloc+28> sd a5,-24(s0)
0x8020029c <kalloc+32> ld a5,-24(s0)
0x802002a0 <kalloc+36> ld a4,0(a5)
0x802002a4 <kalloc+40> auipc a5,0x5
0x802002a8 <kalloc+44> addl a5,a5,-676
0x802002ac <kalloc+48> sd a4,0(a5)

remote Thread 1.1 In: kalloc L13 PC: 0x8020028
sp 0x80205000 0x80205000 <kmem>
(gdb) st
kalloc () at mm.c:13
(gdb) i r sp
sp 0x80204fc0 0x80204fc0
sp 0x80204fc0 0x80204fc0
(gdb) st
(gdb)

```

```

root@f7bf9d5e9abb: /have-fun-debugging/sys...
0x8020027c <kalloc>      addi    sp,sp,-32
0x80200280 <kalloc+4>      sd      ra,24(sp)
0x80200284 <kalloc+8>      sd      s0,16(sp)
0x80200288 <kalloc+12>     addi    s0,s0,32
0x8020028c <kalloc+16>     auipc   a5,0x5
0x80200290 <kalloc+20>     addi    a5,a5,-652
0x80200294 <kalloc+24>     ld      a5,0(a5)
>0x80200298 <kalloc+28>     sd      a5,-24(s0)
0x8020029c <kalloc+32>     ld      a5,-24(s0)
0x802002a0 <kalloc+36>     ld      a4,0(a5)
0x802002a4 <kalloc+40>     auipc   a5,0x5
0x802002a8 <kalloc+44>     addi    a5,a5,-676
0x802002ac <kalloc+48>     sd      a4,0(a5)

remote Thread 1.1 In: kalloc      L16      PC: 0x80200298
(gdb) si
0x0000000000000028 in _stext () at head.S:34
_stext () at head.S:36
Task_init () at proc.c:14
kalloc () at mm.c:13
(gdb) i r s0
s0      0x80204fe0      0x80204fe0
(gdb)

```

单步调试发现总是在ld a4, 0(a5)处出错，中断后查看scauses发现是2，非法指令，运行后发现a5=0，推断是函数位置错误。还没初始化空间就运用指针进行数值读取，应该将head.s中的mm\_init改写到所有函数调用的最前面。

```

root@f7bf9d5e9abb: /have-fun-debugging/sys...
Domain0 Boot HART      : 0
Domain0 HARTs          : 0*
Domain0 Region00       : 0x0000000000000000-0x0000000000001ffff ()
Domain0 Region01       : 0x0000000000000000-0xffffffffffffffff (R,
W,X)
Domain0 Next Address   : 0x0000000000000000
Domain0 Next Arg1      : 0x0000000000000000
Domain0 Next Mode      : S-mode
Domain0 SysReset       : yes

Boot HART ID           : 0
Boot HART Domain       : root
Boot HART ISA          : rv64imafdcsu
Boot HART Features     : scounteren,mcounteren,time
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      : 0x0000000000000b109
...mm_init done!
...proc_init done!

```

### 4.3.2 \_\_dummy 与 dummy 介绍

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

```

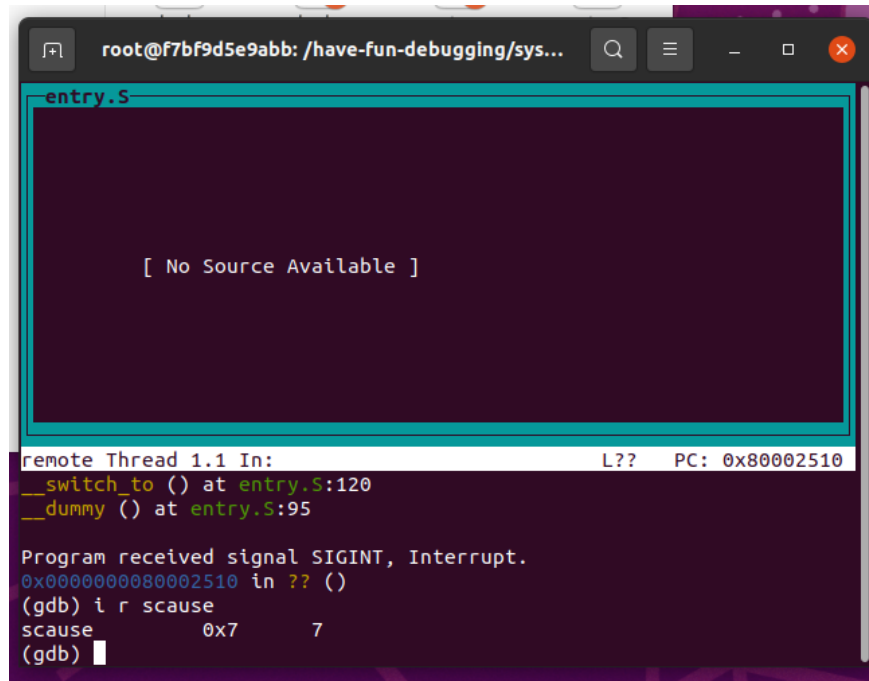
1 // arch/riscv/kernel/proc.c
2
3 void dummy() {
4     uint64 MOD = 1000000007;
5     uint64 auto_inc_local_var = 0;
6     int last_counter = -1;
7     while(1) {
8         if (last_counter == -1 || current->counter != last_counter) {
9             last_counter = current->counter;
10            auto_inc_local_var = (auto_inc_local_var + 1) % MOD;
11            printk("[PID = %d] is running. auto_inc_local_var = %d\n", current->pid,
12                auto_inc_local_var);
13        }
14    }
15 }

```

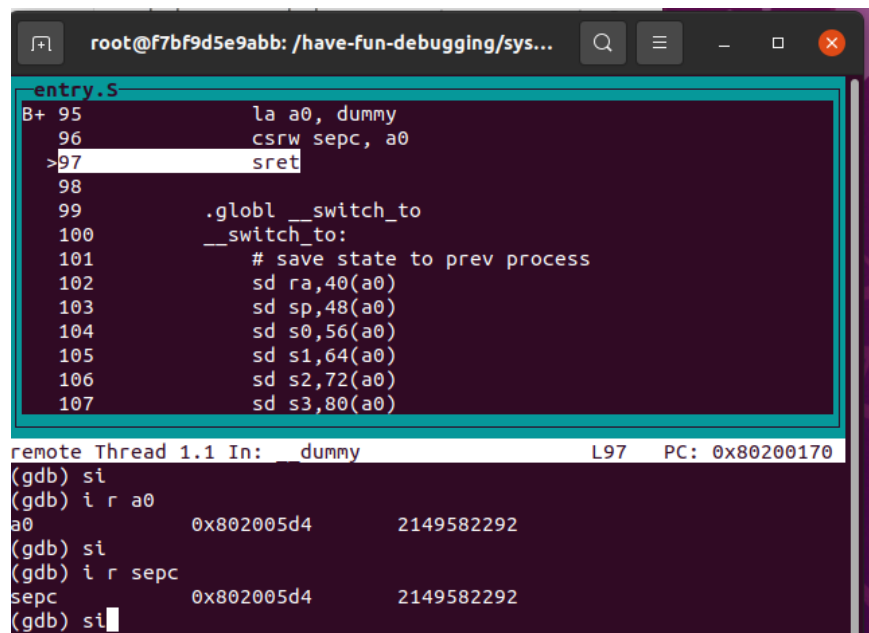
- 当线程在运行时，由于时钟中断的触发，会将当前运行线程的上下文环境保存在栈上（lab5 中实现的 \_traps）。当线程再次被调度时，会将上下文从栈上恢复，但是当我们创建一个新的线程，此时线程的栈为空，当这个线程被调度时，是没有上下文需要被恢复的，所以我们需要为线程第一次调度提供一个特殊的返回函数 \_\_dummy
- 在 entry.S 添加 \_\_dummy
  - 在\_\_dummy 中将 sepc 设置为 dummy() 的地址, 并使用 sret 从中断中返回。

- `__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`）。

```
1 .extern dummy
2 .global __dummy
3 __dummy:
4     la a0, dummy
5     csw sepc, a0
6     sret
```



进入 `dummy` 的时候，异常中断，查看 `scause` 是 7，为 `store access fault` 储存访问故障。



再次复现 bug，查看变量时发现 `sepc` 数值正常，`sp` 为 0，再次复现 bug 时，发现从 `__switch_to` 开始 `sp` 为 0，是进程切换的问题，`sp` 赋值失败。

```
1 (gdb) x/64xb 0x87ffd000
2 (gdb) x/64xb 0x87ffb000
```

使用以上两个指令使用将a1的储存地址和值打印出来可以发现结构体储存地址和数值，将相对地址更改正确得到正确的答案。

remote Thread 1.1 In: switch to L105 PC: 0x80200174								
0x87ffb000:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb008:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb010:	0x0a	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb018:	0x0a	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb020:	0x03	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb028:	0x64	0x01	0x20	0x80	0x00	0x00	0x00	0x00
0x87ffb030:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb038:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb040:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb048:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb050:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb058:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb060:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb068:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb070:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb078:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x87ffb080:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

--Type <RET> for more, q to quit, c to continue without paging--

### 4.3.3 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。

```

1 extern void __switch_to(struct task_struct* prev, struct task_struct* next);
2 void switch_to(struct task_struct* next) {
3     if(next!=current) {
4         printk("switch to [PID = %d COUNTER = %d PRIORITY = %d]\n",next->pid,next-
5 >counter,next->priority);
6         struct task_struct* temp=current;
7         current=next;
8         __switch_to(temp,next);
9     }
10 }

```

#### 遇到bug及解决方案

在进入\_\_switch\_to之后，current的值并没有改变，在\_\_switch\_to中，寄存器的值被更改到next的进程数值，但是a1，a0本身寄存器的值（结构体指针）没有变化，故在调用\_\_switch\_to之前将current指针指向next。

- 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。

```

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

```

```

16     sd s10,136(a0)
17     sd s11,144(a0)
18
19     # restore state from next process
20     ld ra,40(a1)
21     ld sp,48(a1)
22     ld s0,56(a1)
23     ld s1,64(a1)
24     ld s2,72(a1)
25     ld s3,80(a1)
26     ld s4,88(a1)
27     ld s5,96(a1)
28     ld s6,104(a1)
29     ld s7,112(a1)
30     ld s8,120(a1)
31     ld s9,128(a1)
32     ld s10,136(a1)
33     ld s11,144(a1)
34
35     ret

```

### 遇到bug及其解决方案

在switch\_to(next)的时候发生异常中断，scause提示为异常指令，此时跟踪代码发现是next没有成功赋值，导致切换到了空指针。查看汇编代码发现是在比较task[i]->counter时，因为task[i]->counter是uint64类型，编译器将其翻译成了无符号比较，这时永远不可能满足 task[i]->counter > (-1)补，此时在task[i]->counter前加上类型强制转换，转换为int类型进行比较。

### 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 中调用。

```

1 void do_timer(void) {
2     /* 1. 如果当前线程是 idle 线程 或者 当前线程运行剩余时间为0 进行调度 */
3     /* 2. 如果当前线程不是 idle 且 运行剩余时间不为0 则对当前线程的运行剩余时间减1 直接返回 */
4     if(current!=idle&&current->counter!=0){
5         current->counter--;
6     }
7     if(current==idle||current->counter==0){
8         schedule();
9     }
10 }

```

### 4.3.5 实现线程调度

本次实验我们需要实现两种调度算法：1.短作业优先调度算法，2.优先级调度算法。

#### 4.3.5.1 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。

```

1 #ifdef SJF
2
3 void schedule(void) {

```



```

4      //printk("Enter SJF schedule\n");
5
6      int scheMin=999999;
7      struct task_struct* next;
8      next=task[0];
9      while(1){
10         for(int i=1;i<NR_TASKS;i++){
11             // printk("look [PID = %d COUNTER = %d]\n",task[i]->pid,task[i]->counter);
12             if(task[i]->state==TASK_RUNNING && task[i]->counter && (int)task[i]->counter <
scheMin){
13                 scheMin= task[i]->counter ;
14                 next=task[i];
15             }
16         }
17         if(next==idle){
18             for(int i=1;i<NR_TASKS;i++){
19                 task[i]->counter=rand();
20                 printk("SET [PID = %d COUNTER = %d]\n",task[i]->pid,task[i]->counter);
21             }
22         }
23         else
24             break;
25     }
26     switch_to(next);
27 }
28 #endif

```

### 遇到bug及解决方案

开始没有判断counter是否等于0，导致next一直为第一个进程，无法switch\_to下一个进程，调试时一直输出SET [PID = 1 COUNTER = 0]。后在判断里加入判断counter是否为0。

### 4.3.5.2 优先级调度算法

- 参考 [Linux v0.11 调度算法实现](#) 实现。

```

1  #ifdef PRIORITY
2
3  void schedule(void) {
4      int c,pri=0;
5      struct task_struct* next;
6      while(1){
7          c=-1;
8          next=task[0];
9          for(int i=1;i<NR_TASKS;i++){
10             if(task[i]->state==TASK_RUNNING && (int)task[i]->counter){
11                 if(task[i]->priority>next->priority){
12                     next=task[i];
13                 }
14             }
15         }
16         if(next==task[0]){
17             for(int i=1;i<NR_TASKS;i++){
18                 task[i]->counter = rand();
19                 printk("SET [PID = %d COUNTER = %d PRIORITY = %d]\n",task[i]-
>pid,task[i]->counter,task[i]->priority);
20             }
21             printk("\n");
22         }
23     }

```

```

24         else
25             break;
26     }
27     switch_to(next);
28 }
29 #endif

```

## 4.4 编译及测试结果

先开启

```

1 make debug
2 riscv64-unknown-linux-gnu-gdb vmlinux
3 (gdb)
4 target remote:1234

```

符号表

```

1 0000000080200000 A BASE_ADDR
2 0000000080203000 G TIMECLOCK
3 000000008020015c T __dummy
4 000000008020016c T __switch_to
5 0000000080205fe8 B _ebss
6 0000000080203000 G _edata
7 0000000080205fe8 B _kernel
8 0000000080202148 R _erodata
9 0000000080201404 T _etext
10 0000000080204000 B _sbss
11 0000000080203000 G _sdata
12 0000000080200000 T _skernel
13 0000000080202000 R _srodata
14 0000000080200000 T _start
15 0000000080200000 T _stext
16 0000000080200040 T _traps
17 0000000080204000 B boot_stack
18 0000000080205000 B boot_stack_top
19 000000008020020c T clock_set_next_event
20 0000000080205010 B current
21 00000000802006f4 T do_timer
22 00000000802005dc T dummy
23 00000000802001e0 T get_cycles
24 0000000080205008 B idle
25 0000000080205018 B initialize
26 0000000080200274 T kalloc
27 00000000802002d0 T kfree
28 0000000080200348 T kfreerange
29 0000000080205000 B kmem
30 0000000080201394 T memset
31 00000000802003c8 T mm_init
32 0000000080201004 T printk
33 0000000080200ad8 T putc
34 0000000080205048 B r
35 0000000080201084 T rand
36 0000000080200964 T sbi_ecall

```

```

37 0000000080200798 T schedule
38 0000000080200a88 T start_kernel
39 000000008020067c T switch_to
40 000000008020501c B t
41 0000000080205020 B task
42 000000008020040c T task_init
43 0000000080200ac8 T test
44 0000000080200a30 T trap_handler
45 0000000080200b28 t vprintfmt

```

在 M 模式运行期间可能发生的同步例外有五种：

- 访问错误异常 当物理内存的地址不支持访问类型时发生（例如尝试写入 ROM）。
- 断点异常 在执行 **ebreak** 指令，或者地址或数据与调试触发器匹配时发生。
- 环境调用异常 在执行 **ecall** 指令时发生。
- 非法指令异常 在译码阶段发现无效操作码时发生。
- 非对齐地址异常 在有效地址不能被访问大小整除时发生，例如地址为 0x12 的 **amoadd.w**。

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

图 10.3: RISC-V 异常和中断的原因。中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位标识了中断或异常的具体原因。只有在实现了监督者模式时才能处理监督者模式中断和页面错误异常（参见第 10.5 节）。（来自[Waterman and Asanovic 2017]中的表 3.6。）

- 由于加入了一些新的 .c 文件，可能需要修改一些 Makefile 文件，请同学自己尝试修改，使项目可以编译并运行。
- 由于本次实验需要完成两个调度算法，因此需要两种调度算法可以使用 **gcc -D** 选项进行控制。

```

1 $ gcc -D DEBUG myfile.c -o myfile
2 $ ./myfile
3 Debug run

```

- DSJF（短作业优先调度）。
- DPRIORITY（优先级调度）。
- 在 proc.c 中使用 **#ifdef**, **#endif** 来控制代码。修改 Makefile 中的 **CFLAG = \${CF} \${INCLUDE} -DSJF / -DPRIORITY** (作业提交的时候 Makefile 选择任意一个都可以)
- 短作业优先调度输出示例 (为了便于展示，这里一共只初始化了 4 个线程) 同学们最后提交时需要保证 **NR\_TASKS** 为 32 不变

```

1  OpenSBI v0.9
2
3  / _ \      / _ \      / _ \
4  | | | | _ _ \      _ _ \      ( | | |
5  | | | | ' _ \ / _ \ / _ \      < | |
6  | | | | | _ \ | _ \ | _ \      | | |
7  \ _ \ / | . _ \ \ _ \ | _ \ / _ \
8      | |
9      | _
10
11  ...
12
13  Boot HART MIDELEG          : 0x0000000000000222
14  Boot HART MEDELEG         : 0x000000000000b109
15
16  ...mm_init done!
17  ...proc_init done!
18
19  Hello RISC-V
20  idle process is running!
21
22  SET [PID = 1 COUNTER = 10]
23  SET [PID = 2 COUNTER = 10]
24  SET [PID = 3 COUNTER = 5]
25  SET [PID = 4 COUNTER = 2]
26
27  switch to [PID = 4 COUNTER = 2]
28  [PID = 4] is running. auto_inc_local_var = 1
29  [PID = 4] is running. auto_inc_local_var = 2
30
31  switch to [PID = 3 COUNTER = 5]
32  [PID = 3] is running. auto_inc_local_var = 1
33  ....
34  [PID = 3] is running. auto_inc_local_var = 5
35
36  switch to [PID = 2 COUNTER = 10]
37  [PID = 2] is running. auto_inc_local_var = 1
38  ...
39  [PID = 2] is running. auto_inc_local_var = 10
40
41  switch to [PID = 1 COUNTER = 10]
42  [PID = 1] is running. auto_inc_local_var = 1
43  ...
44  [PID = 1] is running. auto_inc_local_var = 10
45
46  SET [PID = 1 COUNTER = 9]
47  SET [PID = 2 COUNTER = 4]
48  SET [PID = 3 COUNTER = 4]
49  SET [PID = 4 COUNTER = 10]
50
51  switch to [PID = 3 COUNTER = 4]
52  [PID = 3] is running. auto_inc_local_var = 6
53  ...
54  [PID = 3] is running. auto_inc_local_var = 9

```

NR\_TASKS = 5 时

```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/lab6
Boot HART MEDELEG      : 0x000000000000b109
...mm_init done!
...proc_init done!
2021 Hello RISC-V
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
```

NR\_TASKS = 32 时

```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/lab6
SET [PID = 15 COUNTER = 4]
SET [PID = 16 COUNTER = 10]
SET [PID = 17 COUNTER = 10]
SET [PID = 18 COUNTER = 5]
SET [PID = 19 COUNTER = 7]
SET [PID = 20 COUNTER = 8]
SET [PID = 21 COUNTER = 4]
SET [PID = 22 COUNTER = 8]
SET [PID = 23 COUNTER = 7]
SET [PID = 24 COUNTER = 6]
SET [PID = 25 COUNTER = 5]
SET [PID = 26 COUNTER = 6]
SET [PID = 27 COUNTER = 10]
SET [PID = 28 COUNTER = 1]
SET [PID = 29 COUNTER = 3]
SET [PID = 30 COUNTER = 8]
SET [PID = 31 COUNTER = 8]
switch to [PID = 12 COUNTER = 1]
[PID = 12] is running. auto_inc_local_var = 1
switch to [PID = 28 COUNTER = 1]
[PID = 28] is running. auto_inc_local_var = 1
switch to [PID = 1 COUNTER = 2]
[PID = 1] is running. auto_inc_local_var = 1
[PID = 1] is running. auto_inc_local_var = 2
switch to [PID = 2 COUNTER = 2]
[PID = 2] is running. auto_inc_local_var = 1
[PID = 2] is running. auto_inc_local_var = 2
switch to [PID = 9 COUNTER = 2]
[PID = 9] is running. auto_inc_local_var = 1
[PID = 9] is running. auto_inc_local_var = 2
switch to [PID = 14 COUNTER = 2]
[PID = 14] is running. auto_inc_local_var = 1
[PID = 14] is running. auto_inc_local_var = 2
switch to [PID = 11 COUNTER = 3]
[PID = 11] is running. auto_inc_local_var = 1
```

## ■ 优先级调度输出示例

```

1  OpenSBI v0.9
2
3  / _ \      / _ \      / _ \
4  | | | | _ _ _ _ _ _ _ _ _ _ ( _ | | ) | | |
5  | | | | ' _ \ / _ \ ' _ \ \ _ \ | | < | |
6  | | | | | ) | _ / | | | _ ) | | ) | | |
7  \ _ \ / | _ / \ _ \ | | | _ / | _ / |
8      | |
9      | |
10
11  ...
12
13  Boot HART MIDELEG      : 0x0000000000000222
14  Boot HART MEDELEG     : 0x000000000000b109
15
16  ...mm_init done!
17  ...proc_init done!
18
19  Hello RISC-V
20  idle process is running!
21
22  SET [PID = 1 PRIORITY = 1 COUNTER = 1]
23
24  SET [PID = 2 PRIORITY = 4 COUNTER = 4]
25
26  SET [PID = 3 PRIORITY = 10 COUNTER = 10]
27
28  SET [PID = 4 PRIORITY = 4 COUNTER = 4]
29
30  switch to [PID = 3 PRIORITY = 10 COUNTER = 10]
31
32  [PID = 3] is running. auto_inc_local_var = 1
33
34  ...
35  [PID = 3] is running. auto_inc_local_var = 10
36
37  switch to [PID = 4 PRIORITY = 4 COUNTER = 4]
38  [PID = 4] is running. auto_inc_local_var = 1
39
40  ...
41  [PID = 4] is running. auto_inc_local_var = 4
42
43  switch to [PID = 2 PRIORITY = 4 COUNTER = 4]
44  [PID = 2] is running. auto_inc_local_var = 1
45
46  ...
47  [PID = 2] is running. auto_inc_local_var = 4
48
49  switch to [PID = 1 PRIORITY = 1 COUNTER = 1]
50  [PID = 1] is running. auto_inc_local_var = 1
51
52  SET [PID = 1 PRIORITY = 1 COUNTER = 1]
53  SET [PID = 2 PRIORITY = 4 COUNTER = 4]
54  SET [PID = 3 PRIORITY = 10 COUNTER = 10]
55  SET [PID = 4 PRIORITY = 4 COUNTER = 4]
56
57  switch to [PID = 3 PRIORITY = 10 COUNTER = 10]
58  [PID = 3] is running. auto_inc_local_var = 11
59
60  ...

```

NR\_TASKS = 5 时

```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/lab6
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...mm_init done!
...proc_init done!
2021 Hello RISC-V
SET [PID = 1 COUNTER = 10 PRIORITY = 1]
SET [PID = 2 COUNTER = 10 PRIORITY = 4]
SET [PID = 3 COUNTER = 5 PRIORITY = 10]
SET [PID = 4 COUNTER = 2 PRIORITY = 4]

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 = 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
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 = 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
```

NR\_TASKS = 32 时

```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/lab6

SET [PID = 29 COUNTER = 3 PRIORITY = 8]
SET [PID = 30 COUNTER = 8 PRIORITY = 9]
SET [PID = 31 COUNTER = 8 PRIORITY = 4]

switch to [PID = 3 COUNTER = 7 PRIORITY = 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
switch to [PID = 5 COUNTER = 5 PRIORITY = 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
switch to [PID = 6 COUNTER = 10 PRIORITY = 10]
[PID = 6] is running. auto_inc_local_var = 1
[PID = 6] is running. auto_inc_local_var = 2
[PID = 6] is running. auto_inc_local_var = 3
[PID = 6] is running. auto_inc_local_var = 4
[PID = 6] is running. auto_inc_local_var = 5
[PID = 6] is running. auto_inc_local_var = 6
[PID = 6] is running. auto_inc_local_var = 7
[PID = 6] is running. auto_inc_local_var = 8
[PID = 6] is running. auto_inc_local_var = 9
[PID = 6] is running. auto_inc_local_var = 10
switch to [PID = 12 COUNTER = 1 PRIORITY = 10]
[PID = 12] is running. auto_inc_local_var = 1
switch to [PID = 14 COUNTER = 2 PRIORITY = 10]
[PID = 14] is running. auto_inc_local_var = 1
[PID = 14] is running. auto_inc_local_var = 2
switch to [PID = 23 COUNTER = 7 PRIORITY = 10]
[PID = 23] is running. auto_inc_local_var = 1
```

## 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？

因为进程是分配内存的单位，同一进程的不同线程共用同一个堆、代码区、数据区，有自己不同的栈，且使用相同的寄存器。故保存的时候只需保存与栈的切换有关以及重要的寄存器值，不需要切换更多的寄存器。

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。

ra 第一次返回点是 \_\_dummy。在 dummy 中，将 sepc 设置为了 dummy 的地址，所以以后进行 context\_switch 时，ra 保存的返回点是 dummy。查看 sepc 和 system\_map 即可确定 sepc 被设置为了 dummy 的地址。



```
root@f7bf9d5e9abb: /have-fun-debugging/sys...
entry.S
B+ 95          la a0, dummy
    96          csrw sepc, a0
>97          sret
    98
    99          .globl __switch_to
   100          __switch_to:
   101              # save state to prev process
   102              sd ra,40(a0)
   103              sd sp,48(a0)
   104              sd s0,56(a0)
   105              sd s1,64(a0)
   106              sd s2,72(a0)
   107              sd s3,80(a0)

remote Thread 1.1 In:  dummy          L97  PC: 0x80200170
(gdb) si
(gdb) i r a0
a0          0x802005d4          2149582292
(gdb) si
(gdb) i r sepc
sepc        0x802005d4          2149582292
(gdb) si
```

## 作业提交

见附件。