浙江大学

本科实验报告

课程名称：操作系统

姓　　名：夏尤楷

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3210104331

指导教师：夏莹杰

2023 年 11 月 5 日

浙江大学操作系统实验报告

实验名称：RV64 内核线程调度

电子邮件地址：459510812@qq.com 手机：15058004449

实验地点：玉泉曹光彪西-503 实验日期：2023 年 11 月 5 日

一、实验目的和要求

1. 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；
2. 了解如何使用时钟中断来实现线程的调度；
3. 了解线程切换原理, 并实现线程的切换；
4. 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、实验过程

（一） 准备工程

从 repo 同步以下代码：rand.h/rand.c、string.h/string.c、mm.h/mm.c、proc.h/proc.c、test.h/test_schedule.h、schedule_null.c/schedule_test.c 以及新增的一些 Makefile 的变化。将这些源文件放入 lab1 的工程文件夹中，放置方式如下：

```
.
├── arch
│   └── riscv
│       ├── include
│       │   ├── mm.h
│       │   └── proc.h
│       └── kernel
│           ├── mm.c
│           └── proc.c
├── include
│   ├── rand.h
│   ├── string.h
│   ├── test.h
│   └── schedule_test.h
│
├── test
│   ├── schedule_null.c
│   ├── schedule_test.c
```

```
│   └── Makefile
├── lib
│   ├── rand.c
│   └── string.c
└── Makefile
```

在 lab2 中我们需要一些物理内存管理的接口，我们可以用 mm.c 文件中的 kalloc 来申请 4KB 的物理页。由于引入了简单的物理内存管理，我们在 `_start` 的 `la sp,boot_stack_top` 和 `jal start_kernel` 之间插入指令 `jal mm_init`，来调用函数 mm_init( )初始化内存管理系统，代码如下：

```
1. la sp,boot_stack_top # store the address of the stack top into the register sp
2. jal mm_init
3. jal start_kernel
```

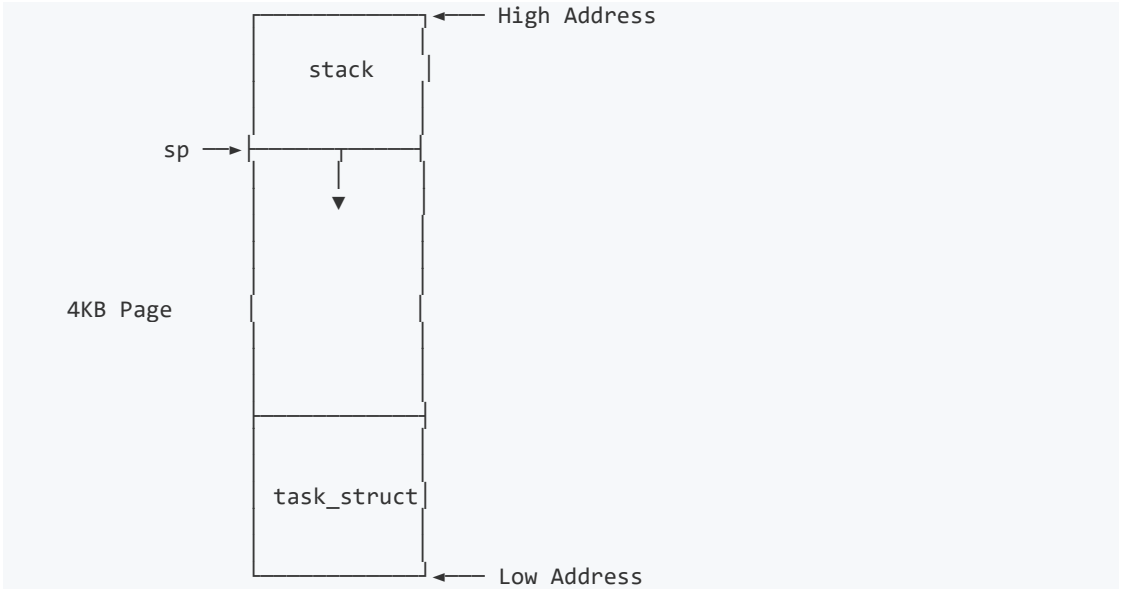
在初始化时还需要用一些自定义的宏，需要修改 defs.h，在 defs.h 添加如下内容：

```
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)))
```

（二） 线程调度功能实现

1. 线程初始化

在初始化线程的时候，我们为每个线程分配一个 4KB 的物理页，我们将线程数据结构 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：



当我们的操作系统运行起来的时候，其本身就是一个线程（idle 线

程），但是我们并没有为它设置好 task_struct。所以第一步我们为 idle 设置 task_struct。并将 current、task[0]都指向 idle。

为了方便起见，我们将 task[1] ~ task[NR_TASKS - 1]全部初始化，与 idle 的设置不同，这里为这些线程设置线程状态段数据结构 thread_struct 中的 ra 和 sp。

这些都在 proc.c 中的函数 task_init( )中实现，最终代码如下：

```
void task_init() {
    test_init(NR_TASKS);
    // 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 task[0] 指向 idle
    idle = (struct task_struct*)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->pid = 0;
    current = task[0] = idle;

    // 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，此外，为了单元测试的需要，counter 和priority 进行如下赋值：
    //      task[i].counter  = task_test_counter[i];
    //      task[i].priority = task_test_priority[i];
    // 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++){
        task[i] = (struct task_struct*)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->counter = task_test_counter[i];
        task[i]->priority = task_test_priority[i];
        task[i]->pid = i;
        task[i]->thread.ra = (uint64)&__dummy;
        task[i]->thread.sp = (uint64)task[i] + PGSIZE;
    }
    printk("...proc_init done!\n");
}
```

我们在 head.S 中`_start` 的 `jal mm_init` 和 `jal start_kernel` 之间插入指令 `jal task_init`，以调用函数 task_init( )如下：

```
la sp,boot_stack_top # store the address of the stack top into the register sp
jal mm_init
jal task_init
jal start_kernel
```

2. `__dummy` 与 dummy

task[1]~task[NR_TASKS-1]都运行 proc.c 中的同一段代码 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，所以我们为线程第一次调度提供一个特殊的返回函数`__dummy`。这个函数被添加在 entry.S 中，在其中将 `sepc` 设

置为 dummy( )的地址，并使用 sret 从中断中返回。代码如下：

```
    .extern dummy
    .globl __dummy
__dummy:
    la t0, dummy
    csrw sepc, t0
    sret
```

3. 实现线程切换

在 proc.c 中的函数 switch_to( )中实现以下功能：

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 proc.c 中插入如下代码（其中要保证__switch_to 函数一定要在调用程序的最后进行执行，原因会在“讨论和心得”里面会讲）：

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

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

void switch_to(struct task_struct* next) {
    if (current == next)
        return;
    else{
        struct task_struct *prev = current;
        current = next;
        __switch_to(prev,next);
    }
}
```

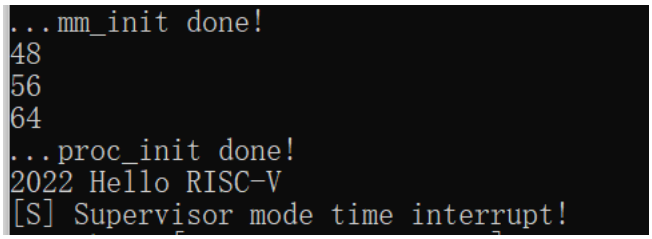
在 entry.S 中实现线程上下文切换__switch_to，它接受两个 task_struct 指针作为参数，保存当前线程的 ra、sp、s0~s11 到当前线程的 thread_struct 中，将下一个线程的 thread_struct 中的相关数据载入到 ra、sp、s0~s11 中。其中，task_struct 和 thread_struct 的结构如下：

```
/* 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;
};
```

为了获取thread_struct内的各元素的首地址相对task_struct的首地址的偏移，在 task_init( )的“printk("...proc_init done!\n");”之前插入以下代码：

```
#define OFFSET(TYPE , MEMBER) ((unsigned long)(&(((TYPE *)0)->MEMBER)))

const uint64 OffsetOfThreadInTask = (uint64)OFFSET(struct task_struct, thread);
const uint64 OffsetOfRaInTask = OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, ra);
const uint64 OffsetOfSpInTask = OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, sp);
const uint64 OffsetOfSInTask = OffsetOfThreadInTask+(uint64)OFFSET(struct thread_struct, s);
printk("%d\n",OffsetOfRaInTask);
printk("%d\n",OffsetOfSpInTask);
printk("%d\n",OffsetOfSInTask);
```

调整代码使得能够编译程序并运行，发现下图情况：



说明 thread_struct 中的 ra、sp、s[12]的首地址相对 task_struct 的偏移量分别为 48、56、64。故在 entry.S 中编写__switch_to 如下：

```
    .globl __switch_to
__switch_to:
    # save state to prev process
    addi t0, a0, 48
    sd ra, 0*8(t0)
    sd sp, 1*8(t0)
    sd s0, 2*8(t0)
    sd s1, 3*8(t0)
    sd s2, 4*8(t0)
    sd s3, 5*8(t0)
    sd s4, 6*8(t0)
    sd s5, 7*8(t0)
    sd s6, 8*8(t0)
    sd s7, 9*8(t0)
    sd s8, 10*8(t0)
    sd s9, 11*8(t0)
    sd s10, 12*8(t0)
    sd s11, 13*8(t0)

    # restore state from next process
    addi t0, a1, 48
    ld ra, 0(t0)
    ld sp, 8(t0)
    ld s0, 2*8(t0)
    ld s1, 3*8(t0)
    ld s2, 4*8(t0)
    ld s3, 5*8(t0)
    ld s4, 6*8(t0)
    ld s5, 7*8(t0)
    ld s6, 8*8(t0)
    ld s7, 9*8(t0)
```

```
32.         ld s8, 10*8(t0)
33.         ld s9, 11*8(t0)
34.         ld s10, 12*8(t0)
35.         ld s11, 13*8(t0)
36.
37.         ret
```

4. 实现调度入口函数

在 proc.c 中实现 do_timer( )（其中要保证 schedule ( )函数一定要在调用程序的最后进行执行，原因会在“讨论和心得”里面会讲），即插入代码如下：

```
1.  // arch/riscv/kernel/proc.c
2.
3.  void do_timer(void) {
4.      // 1. 如果当前线程是 idle 线程 直接进行调度
5.      // 2. 如果当前线程不是 idle 对当前线程的运行剩余时间减1 若剩余时间仍然大于0 则直接返回 否则进行调度
6.
7.      if (current == idle)
8.          schedule();
9.      else {
10.         if ((long)(--(current->counter)) > 0)
11.             return;
12.         else {
13.             current->counter = 0;
14.             schedule();
15.         }
16.     }
17. }
```

在时钟中断处理函数（即 trap.c 中的 trap_handler( )）中调用 do_timer( )（其中要保证 do_timer( )函数一定要在调用程序的最后进行执行，原因会在“讨论和心得”里面会讲），调用后 trap.c 如下：

```
1. #include "printk.h"
2.
3. extern void clock_set_next_event();
4. extern void do_timer(void);
5.
6. void trap_handler(unsigned long scause, unsigned long sepc) {
7.     if (scause >> 63){ // 通过 `scause` 判断trap类型
8.         if (scause % 8 == 5) { // 如果是interrupt，判断是否是timer interrupt
9. //如果是timer interrupt，则打印输出相关信息，并通过`clock_set_next_event()`设置下一次时钟中断
10.             printk("[S] Supervisor mode time interrupt!\n");
11.             clock_set_next_event();
12.             do_timer();
13.         }
14.     }
15. }
```

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。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）下运行剩余时间不为 0 的线程中，选择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]的运行剩余时间使用 rand( )重新赋值，之后再重新进行调度。

（2）优先级调度算法。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）下运行剩余时间不为 0 的线程中，选择优先级最高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]的运行剩余时间根据它们的优先级高低重新赋值，优先级高的赋给运行时间的值就大，之后再重新进行调度。

在 proc.c 中编写 schedule( )函数，其中使用“#ifdef，#endif”来控制编译哪一种算法的代码。其中，短作业优先调度算法对应宏“SJF”，优先级调度算法对应宏“PRIORITY”。修改顶层 Makefile 为 CFLAG = ${CF} ${INCLUDE} -DSJF 或 CFLAG = ${CF} ${INCLUDE} -DPRIORITY，即可选择编译短作业优先调度算法或优先级调度算法。

最终 schedule( )代码如下：

```
1.  void schedule(){
2.      #ifdef PRIORITY
3.      int c,i,next;
4.      static int isInitialized = 0;
5.      while (1) {
6.          c = -1;
7.          next = 0;
8.          i = NR_TASKS;
9.          while (--i) {
10.         if (!task[i])
11.             continue;
12.         if (task[i]->state == TASK_RUNNING && (long)(task[i]->counter) > c)
13.             c = task[i]->counter, next = i;
14.         }
15.         if (c) {
16.             printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n",
                    next, task[next]->priority, task[next]->counter);
17.             break;
18.         }
19.         for(i = 1; i < NR_TASKS ; ++i)
20.             if (task[i]) {
21.                 task[i]->counter = (task[i]->counter >> 1) + task[i]->priority / 10;
22.                 printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", i,
```

```
                        task[i]->priority, task[i]->counter);
            }
        }
        switch_to(task[next]);
        #endif

        #ifdef SJF
        int selected_task_id = -1;
        int min_remaining_time = 1e10;
        int is_all_zero = 1;

        // Check if all running task counters are zero
        for(int i = 1; i < NR_TASKS; ++i) {
            if(task[i]->state == TASK_RUNNING && task[i]->counter > 0) {
                is_all_zero = 0;
                break;
            }
        }

        // If all running task counters are zero, reset them to random values
        if(is_all_zero) {
            for(int i = 1; i < NR_TASKS; ++i) {
                task[i]->counter = rand();
                printk("SET [PID = %d COUNTER = %d]\n", task[i]->pid, task[i]->counter);
            }
        }

        // Find the running task with the smallest remaining time
        for(int i = 1; i < NR_TASKS; ++i) {
            if(task[i]->state == TASK_RUNNING && task[i]->counter > 0
               && task[i]->counter < min_remaining_time) {
                min_remaining_time = task[i]->counter;
                selected_task_id = i;
            }
        }

        // If a task is found, switch to it
        if(selected_task_id != -1) {
            printk("switch to [PID = %d COUNTER = %d]\n",
                task[selected_task_id]->pid, task[selected_task_id]->counter);
            switch_to(task[selected_task_id]);
        } else {
            // No task to schedule
            printk("No runnable tasks with remaining time,
                system is idle or re-schedule\n");
        }
        #endif
    }
```

（三） 测试

1. NR_TASK = 4 时，短作业优先调度算法的测试输出：

```
BBBBD[S] Supervisor mode time interrupt!
D
BBBBDD[S] Supervisor mode time interrupt!
D
BBBBDDD[S] Supervisor mode time interrupt!
D
BBBBDDDD[S] Supervisor mode time interrupt!
D
BBBBDDDDD[S] Supervisor mode time interrupt!
D
BBBBDDDDDD[S] Supervisor mode time interrupt!
D
BBBBDDDDDDD[S] Supervisor mode time interrupt!
D
BBBBDDDDDDDD[S] Supervisor mode time interrupt!
D
BBBBDDDDDDDDD[S] Supervisor mode time interrupt!
switch to [PID = 2 COUNTER = 9]
C
BBBBDDDDDDDDDC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCCCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCCCCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCCCCCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCCCCCCC[S] Supervisor mode time interrupt!
C
BBBBDDDDDDDDDCCCCCCCCC
NR_TASKS = 4, SJF test passed!

[S] Supervisor mode time interrupt!
SET [PID = 1 COUNTER = 1]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 10]
```

根据图中显示，测试通过！

2. NR_TASK = 4 时，短作业优先调度算法的测试输出：

```
C
CCCCCC[S] Supervisor mode time interrupt!
C
CCCCCCC[S] Supervisor mode time interrupt!
C
CCCCCCCC[S] Supervisor mode time interrupt!
C
CCCCCCCCC[S] Supervisor mode time interrupt!
switch to [PID = 3 PRIORITY = 52 COUNTER = 8]
D
CCCCCCCCCD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDDD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDDDD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDDDDD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDDDDDD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDDDDDDD[S] Supervisor mode time interrupt!
D
CCCCCCCCCDDDDDDDD[S] Supervisor mode time interrupt!
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
B
CCCCCCCCCDDDDDDDDB[S] Supervisor mode time interrupt!
B
CCCCCCCCCDDDDDDDDBB[S] Supervisor mode time interrupt!
B
CCCCCCCCCDDDDDDDDBBB[S] Supervisor mode time interrupt!
B
CCCCCCCCCDDDDDDDDBBBB
NR_TASKS = 4, PRIORITY test passed!

[S] Supervisor mode time interrupt!
SET [PID = 1 PRIORITY = 37 COUNTER = 3]
SET [PID = 2 PRIORITY = 88 COUNTER = 8]
SET [PID = 3 PRIORITY = 52 COUNTER = 5]
```

根据图中显示，测试通过！

三、讨论和心得

这个实验让我更加理解了线程调度的原理，提高了我编写汇编和 C 代码的能力。

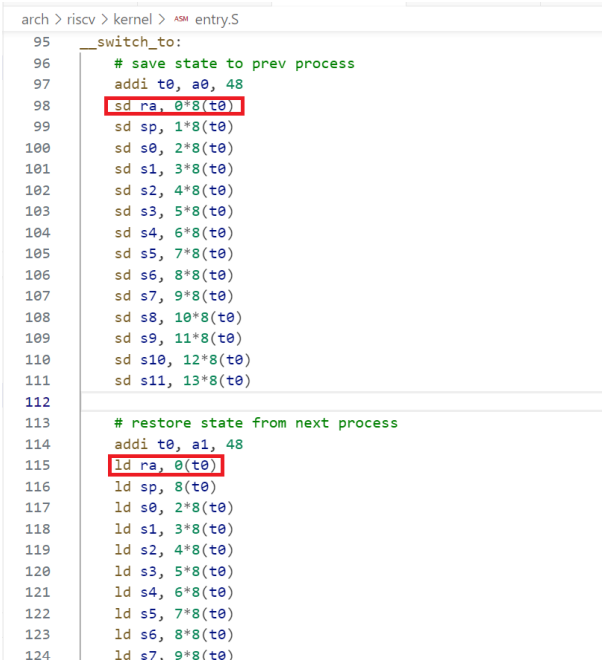
在实验过程中，有一个问题我必须说说。一般来说，在 C 语言代码里面，在某处调用并执行完某个函数时，在代码上紧随其后的语句会非常自然地被执行。但是这个程序里面有个例外：在某个线程内，C 语言调用并执行__switch_to 期间，由于要切换到其他线程，调用执行完后程序要返回的指令的地址在__switch_to 中被汇编语言改变了，所以在 C 语言代码里，调用__switch_to 的语句之后的语句就不会被程序立刻返回到并执行了，而其执行得等到程序返回该线程之后了。所以，调用__switch_to（或是调用要调用__switch_to 的函数）的 C 代码块必须要把调用__switch_to 的代码放到该块代码的最后部分，以免程序在切换线程前并没有执行我们原本预想中会执行的操作。

（因为一开始没发现这个问题，我在编写和调试程序的时候吃了好些苦头啊）

四、思考题

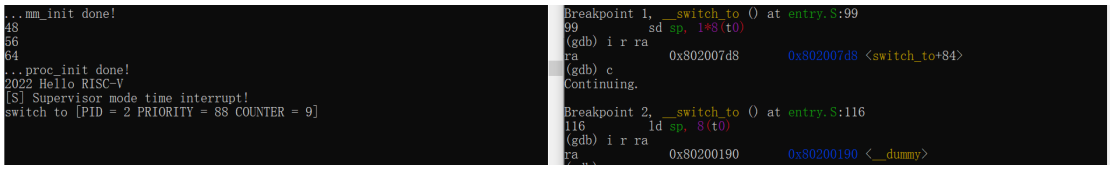
1. 因为__switch_to 函数是在 C 语言的 switch_to( )函数中调用的。而 C 语言在调用__switch_to 函数的时候，会将 RISC-V 的通用寄存器中由调用者保存的寄存器值压入栈中进行保存，所以__switch_to 中只需要保存 C 语言没有保存的寄存器值，即由被调用者保存的通用寄存器（sp 以及 s0~s11）内的值。而为了能够在从其他线程切换回该线程时，能够返回至正确的地址继续运行程序，还要额外保存 ra 寄存器的值。所以总共只需要保存 14 个。

2. 在每次切换线程而运行函数__switch_to 的时候，保存 ra 的指令在 entry.S 的 98 行，恢复 ra 的指令在 entry.S 的 115 行，如下图：

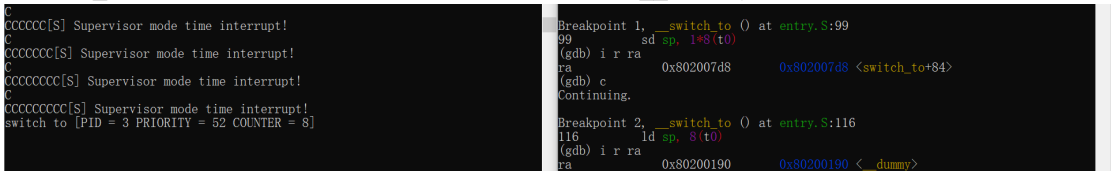


因此，在使用 gdb 调试时，依次输入 b entry.S:99 和 b entry.S:115，把断点打在保存/恢复 ra 寄存器的命令的下一条指令，以方便查看保存/恢复的情况。

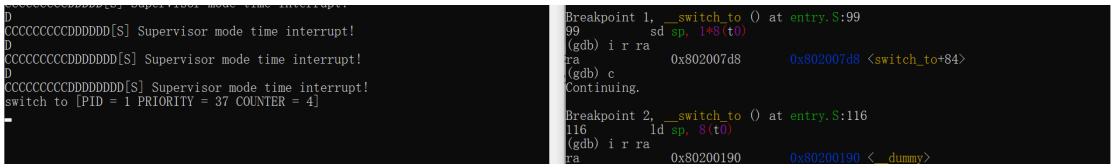
第一次线程调用是从 task[0]切换到 task[2]，可见保存的 task[0]的 ra 为 0x802007d8 <switch_to+84>，恢复的 task[2]的 ra 为 0x80200190<__dummy>。如下图：



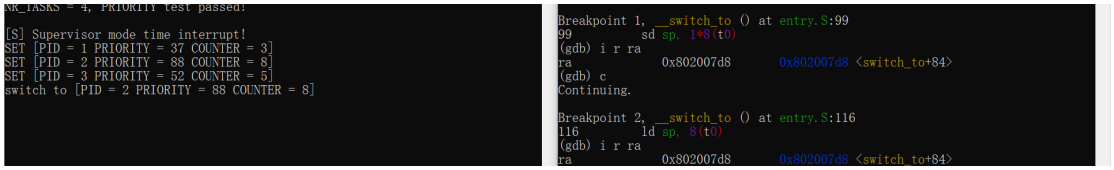
第二次线程调用是从 task[2]切换到 task[3]，可见保存的 task[2]的 ra 为 0x802007d8 <switch_to+84>，恢复的 task[3]的 ra 为 0x80200190<__dummy>。如下图：



第三次线程调用是从 task[3]切换到 task[1]，可见保存的 task[3]的 ra 为 0x802007d8 <switch_to+84>，恢复的 task[1]的 ra 为 0x80200190<__dummy>。如下图：

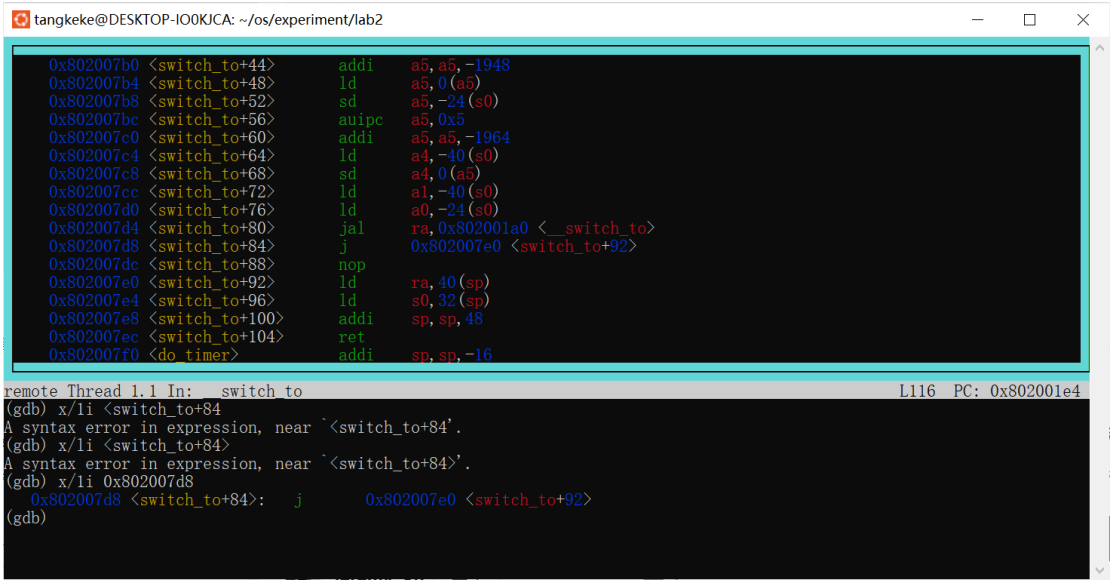


第四次线程调用是从 task[1]切换到 task[2]，可见保存的 task[1]的 ra 为 0x802007d8 <switch_to+84>，恢复的 task[2]的 ra 为 0x802007d8<switch_to+84>。如下图：



不难推想，在之后的线程调用中，保存的上一线程的 ra 和恢复的下一线程的 ra 都将是 0x802007d8<switch_to+84>。

查看该地址的汇编代码（如下图），不难看出，这是 switch_to( )函数的末尾，在调用完 __switch_to 后跳出该函数的指令的地址。



一切运行过的线程在被切换离开的时候，最后一次 C 语言代码中的函数调用都是在这里的__switch_to，所以被切换离开时，保存的 ra 的值就是地址上紧随调用__switch_to 的指令的下一条指令的地址（即 0x802007d8）。当这个线程又被调度的时候，恢复的 ra 的值也是这个地址（即 0x802007d8）了。

而当线程是初次被调度到时，它没执行过任何函数的调用，因此保存的 ra 的值就是一开始初始化的 0x80200190<__dummy>了。

五、附录

无。