

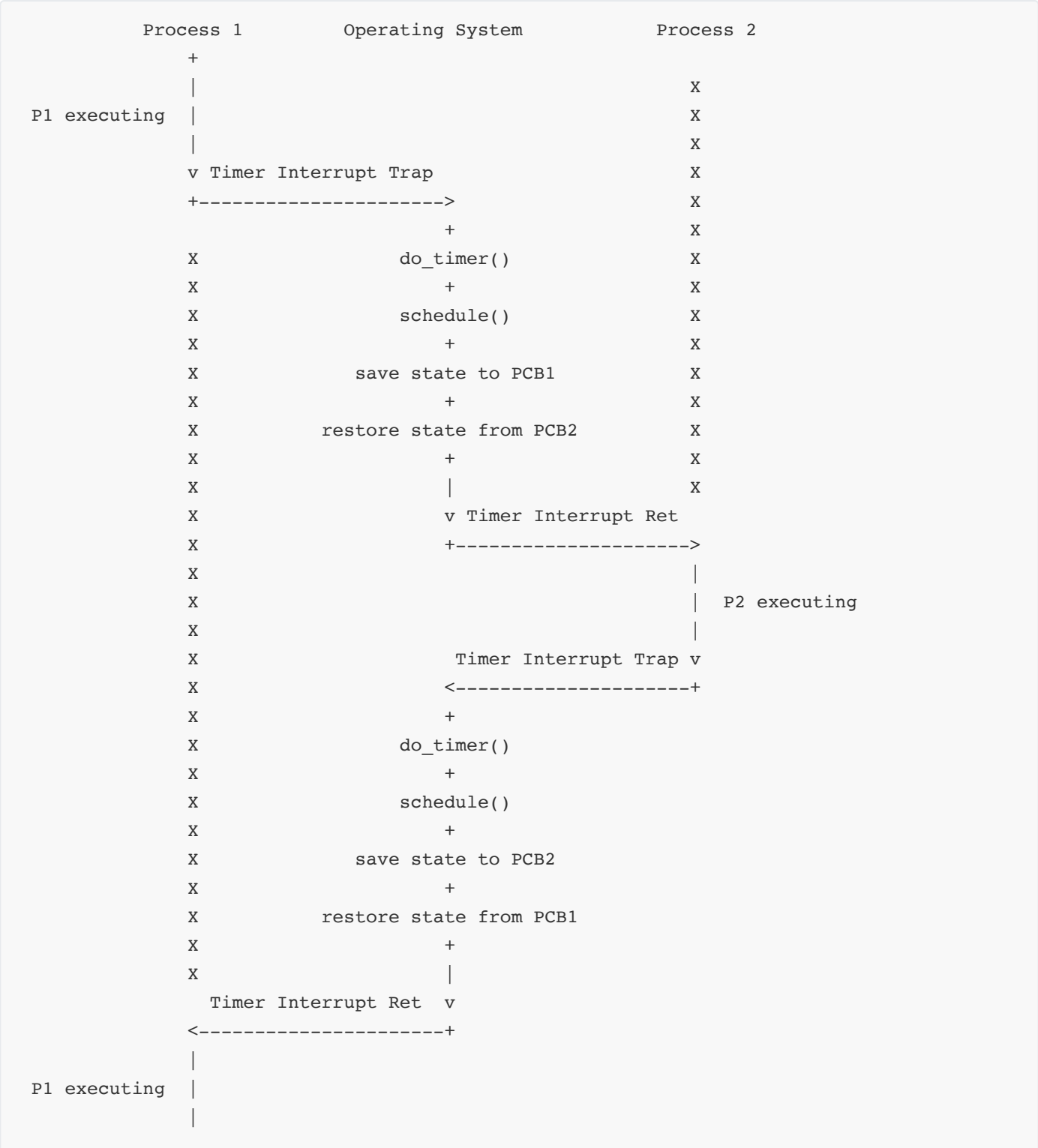
LAB2 实验报告

3210105488 刘扬

1 实验过程

本次实验需要实现的大概是：每次时钟中断触发时，当前线程的运行剩余时间-1，当当前线程运行剩余时间为0时，需要调用 `schedule()` 来找到下一个运行的线程，并通过 `switch_to()` 实现线程切换。

整体流程如下：



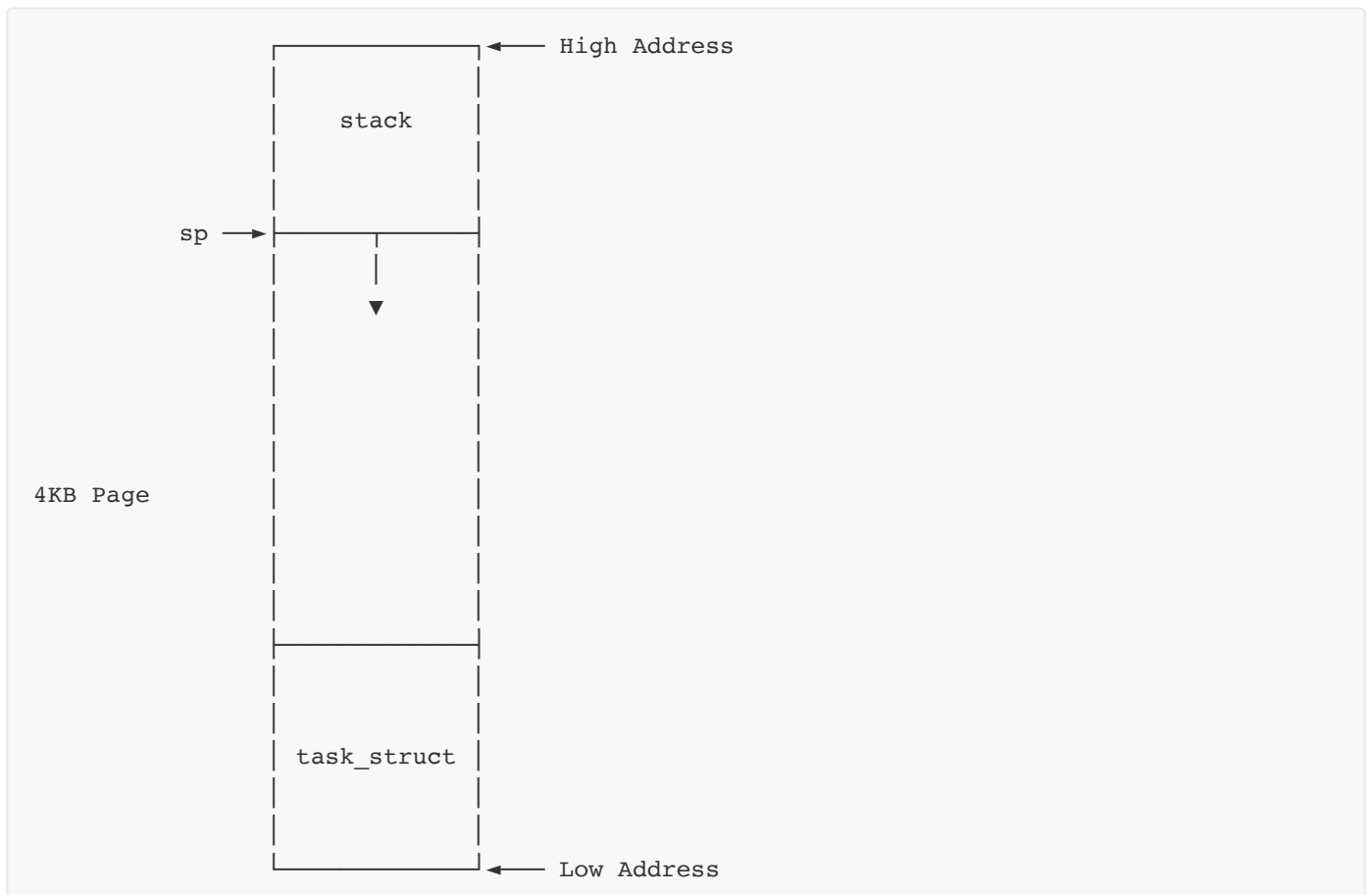
1.1 准备工作

将lab1的代码merge到lab2，修改 `MakeFile` 和 `defs.h`，不赘述。

1.2 线程调度功能实现

1.2.1 线程初始化

在初始化线程时，物理页的大小为4KB，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：



首先初始化 `idle` 即 `task[0]`，给 `idle` 设置 `task_struct`，并将 `current` 和 `task[0]` 都指向 `idle`。

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

然后将 `task[1]~task[NR_TASKS - 1]` 全部初始化，需要设置好 `ra` 和 `sp`。其中 `ra` 设置为 `__dummy` 的地址，`sp` 设置为该线程申请的物理页的高地址。初始化 `ra` 的目的是让每个线程在第一次时钟中断时能够正确地进入 `dummy()`

```

for (int i=0; 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;
}

```

1.2.2 `__dummy` 与 `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，所以我们需要为线程 **第一次调度** 提供一个特殊的返回函数 `__dummy`

1.2.2.1 `dummy`

`dummy` 已经在实验指导中给出，在不触发时钟中断时，该线程会一直在 `while` 循环中执行，一旦触发中断，`current->counter` 改变，`auto_inc_local_var` 自增。

```

void dummy() {
    schedule_test();
    uint64 MOD = 1000000007;
    uint64 auto_inc_local_var = 0;
    int last_counter = -1;
    while(1) {
        if ((last_counter == -1 || current->counter != last_counter) && current->counter >
0) {
            if(current->counter == 1){
                --(current->counter);    // forced the counter to be zero if this thread is
going to be scheduled
            }                            // in case that the new counter is also 1, leading
the information not printed.
            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);
        }
    }
}

```

1.2.2.2 `__dummy`

为了线程在第一次调度时能够正确地进入 `dummy`，简单来说，就是时钟中断在完成后的 `sret` 需要返回到 `dummy` 中。

```

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

```

1.2.3 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。

```

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

```

sd ra, 48(a0) # Offset of thread_struct = 48
sd sp, 56(a0)
sd s0, 64(a0)
...
sd s11, 152(a0)

```

- 将下一个线程的 `thread_struct` 中的相关数据载入到 `ra`, `sp`, `s0~s11` 中。

```

ld ra, 48(a1)
ld sp, 56(a1)
ld s0, 64(a1)
...
ld s11, 152(a1)
ret

```

`thread_struct` 的偏移量是根据 `task_struct` 的数据结构算出的:

在 `thread_struct` 前有6个 `uint64`, $6 * 8 = 48$ 。

```

struct thread_info {
    uint64 kernel_sp;
    uint64 user_sp;
};

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;
};

```

1.2.4 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。

注意 `current->counter` 的类型是 `uint64`，需要做类型转换。

```

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 若剩余时间仍然大于0 则直接返回 否则进行调度
    if (current == idle){
        schedule();
    }
    else {
        if(current->counter){
            --(current->counter);
        }
        if ((int)current->counter > 0){ //current counter uint!!
            return;
        }
        else {
            schedule();
        }
    }
}

```

1.2.5 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。

1.2.5.1 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。

参考[Linux v0.11 调度算法实现](#)。区别是Linux的调度算法是找到最大 `counter` 的线程，SJF是找到最小 `counter` 的线程。注意如果有多个线程的 `counter` 一样小，调度 `pid` 小的线程。

```
struct task_struct* next;
while(1){
    int c = __INT_MAX__;
    int i = 0;
    int zero_cnt = 0;
    while(++i < NR_TASKS){
        if(task[i]->state == TASK_RUNNING && task[i]->counter == 0){
            zero_cnt++;
            continue;
        }
        if(task[i]->state == TASK_RUNNING && (int)task[i]->counter < c){
            c = task[i]->counter;
            next = task[i];
        }
    }
    if(zero_cnt != NR_TASKS - 1) break;
    i = NR_TASKS;
    while(--i){
        task[i]->counter = rand();
        printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]->pid, task[i]->priority, task[i]->counter);
    }
}
switch_to(next);
```

1.2.5.2 优先级调度算法

参考[Linux v0.11 调度算法实现](#)。

```
struct task_struct* next;
while(1){
    int c = -1;
    int i = NR_TASKS;
    while(--i){
        if(task[i]->state == TASK_RUNNING && (int)task[i]->counter > c){
            c = task[i]->counter;
            next = task[i];
        }
    }
    if(c) break;
    i = NR_TASKS;
    while(--i){
        task[i]->counter = (task[i]->counter >> 1) + task[i]->priority;
        printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", task[i]->pid, task[i]->priority, task[i]->counter);
    }
}
```

```
switch_to(next);
```

1.3 编译及测试

[illegible]

```

FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNL[S] Supervisor Mode Timer Interrupt
L
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLL[S] Supervisor Mode Timer Interrupt
L
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLLL[S] Supervisor Mode Timer Interrupt
switch to [PID = 14 PRIORITY = 6 COUNTER = 2]
0
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLLLO[S] Supervisor Mode Timer Interrupt
0
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLLLO0[S] Supervisor Mode Timer Interrupt
switch to [PID = 7 PRIORITY = 5 COUNTER = 2]
H
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLLLO0H[S] Supervisor Mode Timer Interrupt
H
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLLLO0HH[S] Supervisor Mode Timer Interrupt
switch to [PID = 8 PRIORITY = 25 COUNTER = 1]
I
FFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGCCCCCCCCDDDDDDDDJJJJJJPPPPPEEEEEEMMMBBBBNNNLLLO0HHI
NR_TASKS = 16, PRIORITY test passed!

```

2 思考题

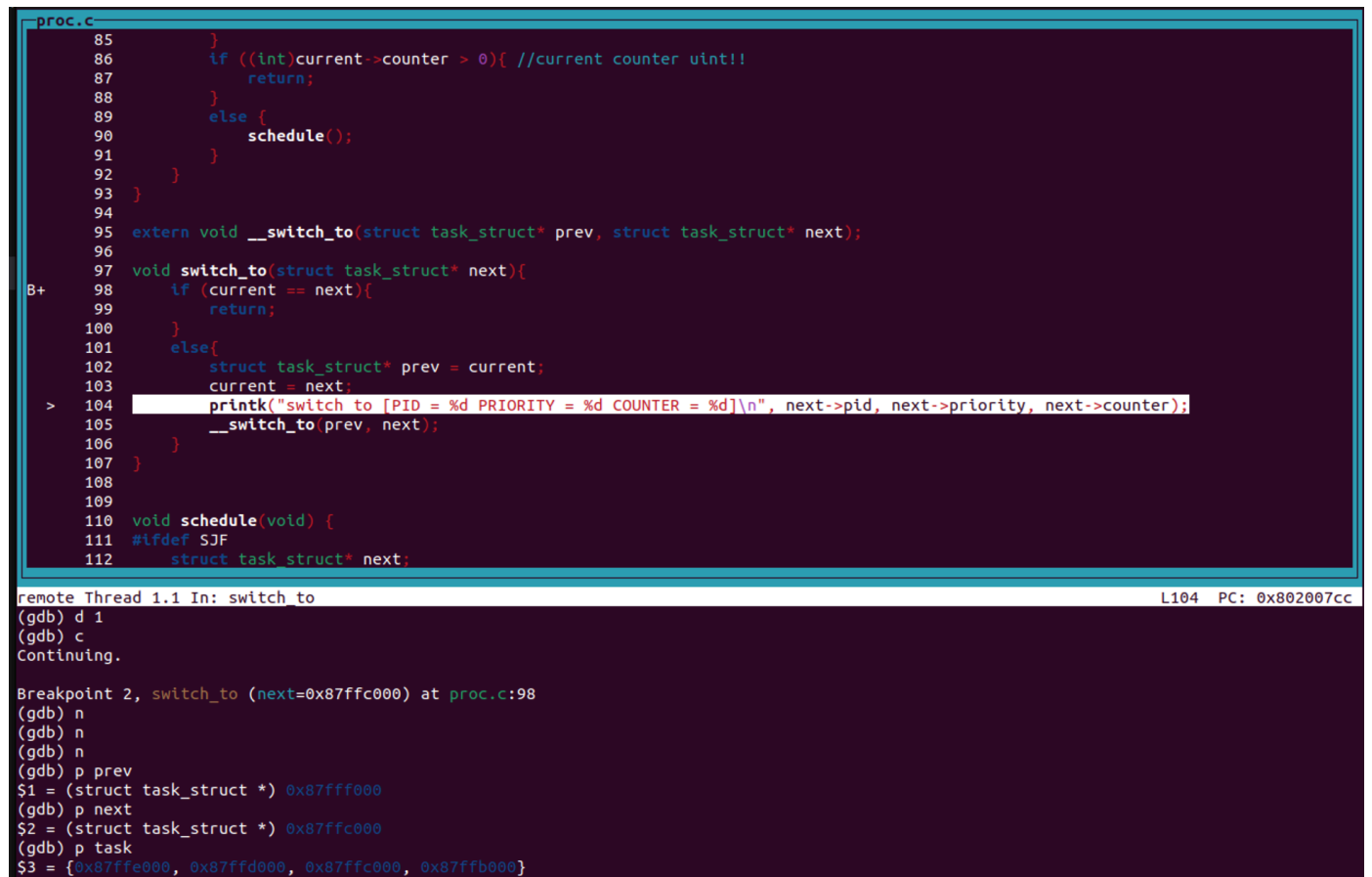
1.在 RV64 中一共用 32 个通用寄存器，为什么 `context_switch` 中只保存了14个？

每个线程都有自己的 `ra` 和 `sp`，因此在做 `context_switch` 时需要保存上一个线程的 `ra` 和 `sp`，并载入下一个线程的 `ra` 和 `sp`。`s0-s11` 是 callee saved，在做 `context_switch` 时不会由 caller 进行保存，因此我们需要在 callee 中将 `s0-s11` 保存起来。

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 (需要截图)。

是否需要线程切换是由 `do_timer` 判断的, 如果需要线程切换, 即当前线程运行剩余时间为0时, 会调用 `schedule` 来选择下一个线程, 并在 `schedule` 中调用 `switch_to` 来切换线程。如果该线程是第一次调用, `ra` 保存的是 `__dummy` 的地址。我们在 `switch_to` 打上断点, 然后进行观察:

第一次线程切换时, 由 `task[0]` 切换为 `task[2]`:



```
proc.c
85     }
86     if ((int)current->counter > 0){ //current counter uint!!
87         return;
88     }
89     else {
90         schedule();
91     }
92 }
93 }
94
95 extern void __switch_to(struct task_struct* prev, struct task_struct* next);
96
97 void switch_to(struct task_struct* next){
98     if (current == next){
99         return;
100     }
101     else{
102         struct task_struct* prev = current;
103         current = next;
104         printk("switch to [PID = %d PRIORITY = %d COUNTER = %d]\n", next->pid, next->priority, next->counter);
105         __switch_to(prev, next);
106     }
107 }
108
109 void schedule(void) {
110     #ifdef SJF
111         struct task_struct* next;
112         ...
113     }
114 }
```

```
remote Thread 1.1 In: switch_to L104 PC: 0x802007cc
(gdb) d 1
(gdb) c
Continuing.

Breakpoint 2, switch_to (next=0x87ffc000) at proc.c:98
(gdb) n
(gdb) n
(gdb) n
(gdb) p prev
$1 = (struct task_struct *) 0x87fff000
(gdb) p next
$2 = (struct task_struct *) 0x87ffc000
(gdb) p task
$3 = {0x87ffe000, 0x87ffd000, 0x87ffc000, 0x87ffb000}
(gdb) n
```

进入 `__switch_to`, 查看到 `ra` 的值是 `__dummy` 的地址:


```

entry.S
102     sd s6, 112(a0)
103     sd s7, 120(a0)
104     sd s8, 128(a0)
105     sd s9, 136(a0)
106     sd s10, 144(a0)
107     sd s11, 152(a0)
108     # load next ra, sp, s0-s11
B+ 109     ld ra, 48(a1)
> 110     ld sp, 56(a1)
111     ld s0, 64(a1)
112     ld s1, 72(a1)
113     ld s2, 80(a1)
114     ld s3, 88(a1)
115     ld s4, 96(a1)
116     ld s5, 104(a1)
117     ld s6, 112(a1)
118     ld s7, 120(a1)
119     ld s8, 128(a1)
120     ld s9, 136(a1)
121     ld s10, 144(a1)
122     ld s11, 152(a1)
123     ret
124
125
126
127
128
129

remote Thread 1.1 In: __switch_to L110 PC: 0x802001b8
(gdb) si
(gdb) si
(gdb) n

Breakpoint 3, __switch_to () at entry.S:94
(gdb) b 109
Breakpoint 4 at 0x802001b4: file entry.S, line 109.
(gdb) c
Continuing.

Breakpoint 4, __switch_to () at entry.S:109
(gdb) n
(gdb) i r ra
ra 0x8020016c 0x8020016c <__dummy>
(gdb)

```

因为 ra 是 __dummy 的地址，所以在 ret 之后进入 __dummy：

```

entry.S
73     ld x6, 208(sp)
74     ld x5, 216(sp)
75     ld x4, 224(sp)
76     ld x3, 232(sp)
77     ld x1, 240(sp)
78     ld x0, 248(sp)
79     ld sp, 256(sp)
80     # 4. return from trap
81     sret
82
83     # __dummy
84     .globl __dummy
85     dummy:
> 86     la t0, dummy
87     csrw sepc, t0
88     sret
89
90     # __switch_to
91     .globl __switch_to
92     __switch_to:
93     # save prev ra, sp, s0-s11
B+ 94     sd ra, 48(a0) # Offset of thread_struct = 48
95     sd sp, 56(a0)
96     sd s0, 64(a0)
97     sd s1, 72(a0)
98     sd s2, 80(a0)
99     sd s3, 88(a0)
100     sd s4, 96(a0)

remote Thread 1.1 In: dummy L86 PC: 0x8020016c
(gdb) n
(gdb) i r ra
ra 0x8020016c 0x8020016c <__dummy>
(gdb) d 4
(gdb) b 123
Breakpoint 5 at 0x802001ec: file entry.S, line 123.
(gdb) c
Continuing.

Breakpoint 5, __switch_to () at entry.S:123
(gdb) b dummy
Breakpoint 6 at 0x80200618: file proc.c, line 60.
(gdb) n
__dummy () at entry.S:86
(gdb)

```

__dummy 跳转到 dummy，第一次线程切换完成，之后在 dummy 中每次触发时钟中断都会让 counter-1。之后对于 task[1] 和 task[3] 的调度也同上，因为都是第一次调度，所以 ra 都是默认的 __dummy 的地址。

直接跳到第二次调度 `task[2]`，进入 `__switch_to` 后观察 `ra` 的值：

```
entry.S
90  #__switch_to
91  .globl __switch_to
92  __switch_to:
93  # save prev ra,sp,s0-s11
B+ 94  sd ra, 48(a0) # Offset of thread_struct = 48
95  sd sp, 56(a0)
96  sd s0, 64(a0)
97  sd s1, 72(a0)
98  sd s2, 80(a0)
99  sd s3, 88(a0)
100 sd s4, 96(a0)
101 sd s5, 104(a0)
102 sd s6, 112(a0)
103 sd s7, 120(a0)
104 sd s8, 128(a0)
105 sd s9, 136(a0)
106 sd s10, 144(a0)
107 sd s11, 152(a0)
108 # load next ra,sp,s0-s11
B+ 109 ld ra, 48(a1)
> 110 ld sp, 56(a1)
111 ld s0, 64(a1)
112 ld s1, 72(a1)
113 ld s2, 80(a1)
114 ld s3, 88(a1)
115 ld s4, 96(a1)
116 ld s5, 104(a1)
117 ld s6, 112(a1)

remote Thread 1.1 In: __switch_to L110 PC: 0x802001b8
Breakpoint 2, __switch_to (next=0x87ffc000) at proc.c:98
(gdb) c
Continuing.

Breakpoint 3, __switch_to () at entry.S:94
(gdb) b 109
Breakpoint 8 at 0x802001b4: file entry.S, line 109.
(gdb) c
Continuing.

Breakpoint 8, __switch_to () at entry.S:109
(gdb) n
(gdb) i r ra
ra 0x80200804 0x80200804 <switch_to+128>
(gdb) c
```

可以看到 `ra` 变成了 `0x80200804`，即 `switch_to` 函数中，调用 `__switch_to` 的下一行。因为在第一次调用时，我们 `sd` 了 `ra` 的值，即调用 `__switch_to` 后的返回地址在 `thread_struct` 中，因此我们 `ld` 得到的是 `__switch_to` 的返回地址。

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

这时候函数会逐层返回

```
__switch_to --> switch_to --> schedule --> do_timer --> trap_handler --> _traps
```

再由 `_traps` 中保存的 `context` 返回到正确的地址，即 `sret` 回到 `dummy` 中正在运行的行数。

在 `_traps` 的 `sret` 前打上断点即可验证：

```

entry.S
68 ld x11, 168(sp)
69 ld x10, 176(sp)
70 ld x9, 184(sp)
71 ld x8, 192(sp)
72 ld x7, 200(sp)
73 ld x6, 208(sp)
74 ld x5, 216(sp)
75 ld x4, 224(sp)
76 ld x3, 232(sp)
77 ld x1, 240(sp)
78 ld x0, 248(sp)
79 ld sp, 256(sp)
80 # 4, return from trap
B+> 81 sret
82
83 # __dummy
84 .globl __dummy
85 __dummy:
86 la t0, dummy
87 csw sepc, t0
88 sret
89
90 # switch to
~/lab2/arch/riscv/kernel/head.o switch_to
92 __switch_to:
93 # save prev ra, sp, s0-s11
B+ 94 sd ra, 48(a0) # Offset of thread_struct = 48
95 sd sp, 56(a0)

remote Thread 1.1 In: traps L81 PC: 0x80200168
(gdb) n
(gdb) n
(gdb) b 81
Breakpoint 12 at 0x80200168: file entry.S, line 81.
(gdb) c
Continuing.

Breakpoint 12, _traps () at entry.S:81
(gdb) i r sret
Invalid register `sret'
(gdb) i r sepc
sepc 0x80200640 2149582400
(gdb) b * 0x80200640
Breakpoint 13 at 0x80200640: file proc.c, line 65.
(gdb)

```

proc.c 的65行即:

```

59 void dummy() {
60     schedule_test();
61     uint64 MOD = 1000000007;
62     uint int last_counter var = 0;
63     int last_counter = -1;
64     while(1) {
65         if ((last_counter == -1 || current->counter != last_counter) && current->counter > 0)
66             if(current->counter == 1){
67                 --(current->counter); // forced the counter to be zero if this thread is go
68             } // in case that the new counter is also 1, leading th
69         last_counter = current->counter;
70         auto_inc_local_var = (auto_inc_local_var + 1) % MOD;
71         printk("[PID = %d] is running. auto_inc_local_var = %d\n", current->pid, auto_inc
72     }
73 }
74 }

```