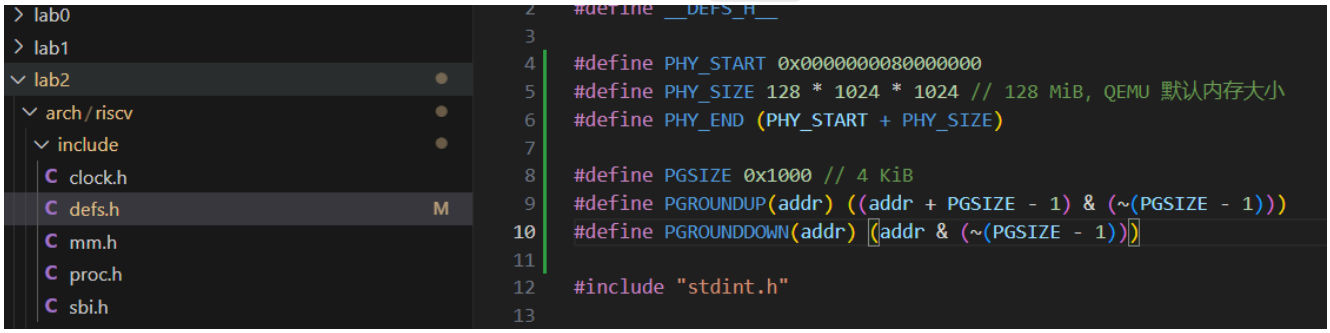
浙江大学实验报告

课程名称：操作系统
实验项目名称：RV64 内核线程调度
学生姓名：展翼飞 学号：3190102196
电子邮件地址：1007921963@qq.com
实验日期： 2024年 10 月 11 日

一、实验内容

1. 准备工程

- 从lab1中拷贝lab1完成内容至lab2中，将如下宏定义添加进`defs.h`



  尝试编译，提示找不到`memset`函数定义，进行检查，发现`\lib`中除了`printk.c`均未编译，原因为makefile文件目标仅为`printk.o`，进行修改：

```
lab2 > lib > M Makefile
# 查找源文件并生成对象文件列表
C_SRC       = $(sort $(wildcard *.c))   # 自动查找所有的 .c 文件
OBJ         = $(patsubst %.c,%.o,$(C_SRC))  # 将所有 .c 文件转换为 .o 文件

# 默认目标，编译所有的 .o 文件
all: $(OBJ)

# 编译每个 .c 文件为对应的 .o 文件
%.o: %.c
    $(GCC) $(CFLAG) -c $< -o $@

# 清理编译生成的文件
clean:
    rm -f $(OBJ)
    @echo "Cleaned lib folder."
```

  修改完成后工程正确编译

1.1 `proc.h` 数据结构定义

- 阅读添加的`proc.h`，里面内置了线程状态数据结构与线程调度函数的头文件以及相关宏定义

2. 线程调度功能实现

2.1 线程初始化

- 在`proc.c`中按照要求步骤实现线程的初始化，包含分配物理页和初始化对应线程的`task_struct`

```c
// 线程初始化函数
void task_init() {
    srand(2024);  // 设置随机种子

    // 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

    // 为 idle 分配物理页
    idle = (struct task_struct *)kalloc();
    if (idle == NULL) {
        printk("Failed to allocate page for idle task\n");
        return;
    }

    // 初始化 idle 线程
    idle->state = TASK_RUNNING;   // 设置状态为运行态
    idle->counter = 0;            // 不参与调度，counter 设置为 0
    idle->priority = 0;           // 优先级也设置为 0
    idle->pid = 0;                // pid 设置为 0

    current = idle;               // current 指向 idle
    task[0] = idle;               // task[0] 也指向 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，counter 和 priority 进行如下赋值：
    //     - counter  = 0;
    //     - priority = rand() 产生的随机数 (控制范围在 [PRIORITY_MIN, PRIORITY_MAX] 之间)
    // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中的 ra 和 sp
    //     - ra 设置为 __dummy (见 4.2.2) 的地址
    //     - sp 设置为该线程申请的物理页的高地址

    // 为其他线程分配物理页并初始化
    for (int i = 1; i < NR_TASKS; i++) {
        task[i] = (struct task_struct *)kalloc();  // 分配物理页
        if (task[i] == NULL) {
            printk("Failed to allocate page for task %d\n", i);
            return;
        }

        // 设置线程的状态和调度信息
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = (rand() % (PRIORITY_MAX - PRIORITY_MIN + 1)) + PRIORITY_MIN;  // 随机优先级

        // 设置线程的 thread_struct 中的 ra 和 sp
        task[i]->thread.ra = (uint64_t)__dummy;  // ra 设置为 __dummy 的地址
        task[i]->thread.sp = (uint64_t)task[i] + PGSIZE;  // sp 指向物理页的高地址
        task[i]->pid = i;  // pid 设置为当前的索引值
    }

    printk("...task_init done!\n");
```

其中每个线程的`task_struct`放在物理页的起始地址，而其子结构`thread_struct`紧接着`task_struct`

2.2 `__dummy` 与 `dummy` 的实现

1. `dummy`分析
   `dummy()`是除了`idle`线程外所有线程运行的代码，它记录线程被调度前的剩余运行时间`last_counter`，并在时钟中断导致线程被调度导致剩余运行时间`counter`减少时增大局部变量`auto_inc_local_var`标志线程工作进行，并使用`printk`打印线程`pid`等信息。
2. `__dummy`的实现
   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`，在 `__dummy` 中将 sepc 设置为 `dummy()` 的地址，并使用 `sret` 从 S 模式中返回

```
88      .globl __dummy          # 声明 __dummy 为全局函数
89  __dummy:
90      la t0, dummy            # 加载 dummy() 函数的地址到 t0
91      csrw sepc, t0           # 将 sepc 设置为 dummy() 的地址
92      sret                    # 从 S 模式返回，进入用户态，开始执行 dummy()
```

2.3 实现线程切换

1. `proc.c`中`switch_to`实现
   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 (next == current) return;
    else {
        struct task_struct* current_saved = current;
        current = next;
        __switch_to(current_saved, next);
    }
}
```

2. `entry.S`中`__switch_to`实现
   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` 中进行恢复：

```
.globl __switch_to
__switch_to:
    # save state to prev process
    # including ra, sp, s0-s11
    add t0, a0, 32  # offset of thread_struct in current task_struct
    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
    add t0, a1, 32  # offset of thread_struct in next task_struct
    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
```

此处`__switch_to`由`proc.c`的`switch_to`调用，`a0 a1`两个参数分别为当前线程和下一线程的物理页起始地址，而线程寄存器保存在物理页的`thread_struct`中，`thread_struct`是放在物理页起始地址的`task_struct`的成员变量，而它的偏移了四个`uint64_t`即32Bytes，因此偏移地址为32，由此可以依次存取`thread_struct`中保存的线程寄存器

2.4 实现调度入口函数

实现`do_timer()`函数，并在`trap.c`时钟中断处理函数中调用：
在`proc.c`中新增`do_timer`函数入下：

```
void do_timer(void) {
    // 1. 如果当前线程是 idle 线程或当前线程时间片耗尽则直接进行调度
    // 2. 否则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 == task[0]) schedule();
    else {
        current->counter -= 1;
        if (current->counter == 0) schedule();
    }
}
```

并在`trap.c`的`trap_handler`函数调用它，实现时钟中断发生时减少当前运行线程的剩余运行时间：

```
        // 检查是否是时钟中断
        if (interrupt_id == SUPERVISOR_TIMER_INTERRUPT_ID) {
            // 打印调试信息
            printk("[S] Supervisor Mode Timer Interrupt\n");

            // 处理时钟中断，设置下一个时钟中断
            clock_set_next_event();

            //线程运行时间自减
            do_timer();
```

2.5 线程调度算法实现

在`proc.c`中实现线程优先级调度算法`schedule`如下：

- `task_init` 的时候随机为各个线程赋予了优先级
- 调度时选择 `counter` 最大的线程运行
- 如果所有线程 `counter` 都为 0，则令所有线程 `counter = priority`
  - 即优先级越高，运行的时间越长，且越先运行
  - 设置完后需要重新进行调度
- 最后通过 `switch_to` 切换到下一个线程

```
//priority 调度
void schedule(void) {
    uint64_t maxCounter, i, next;
    struct task_struct** pointer;

    while (1) {
        maxCounter = 0;  // 初始化最大剩余时间片 (counter) 为0
        next = 0;        // 初始化下一个调度的任务索引为0 (默认 idle 任务)
        i = NR_TASKS;    // 从任务列表末尾开始遍历
        pointer = &task[NR_TASKS];  // 指针指向任务数组的末尾

        // 遍历所有任务，选择状态为 TASK_RUNNING 且剩余时间片最多的任务
        while (--i) {  // 遍历 task[NR_TASKS-1] 到 task[1]
            if (!*--pointer) continue;  // 如果任务不存在，跳过
            // 如果任务是 RUNNING 状态，且剩余时间片大于当前最大值，则更新 maxCounter 和 next
            if ((*pointer)->state == TASK_RUNNING && (*pointer)->counter > maxCounter) {
                maxCounter = (*pointer)->counter;  // 更新最大剩余时间片
                next = i;  // 记录该任务的索引，表示它是下一个要调度的任务
            }
        }

        // 如果找到了一个剩余时间片大于0的任务，退出循环
        if (maxCounter > 0) break;

        // 如果所有任务的 counter 都为 0，进行时间片重新分配
        printk("\n");

        // 遍历所有任务，重新分配时间片
        for (pointer = &task[NR_TASKS-1]; pointer > &task[0]; --pointer) {
            if (*pointer) {
                // 重新计算时间片：右移1位相当于除以2，然后加上任务的优先级
                (*pointer)->counter = ((*pointer)->counter >> 1) + (*pointer)->priority;
                // 输出调度信息，打印该任务的 PID、优先级和重新分配后的 counter
                printk("SET [PID = %d PRIORITY = %d COUNTER = %d]\n", (*pointer)->pid, (*pointer)->priority, (*pointer)->counter);
            }
        }
    }

    // 输出即将切换的任务信息，包括 PID、优先级和剩余时间片
    printk("\nswitch to [PID = %d PRIORITY = %d COUNTER = %d]\n", task[next]->pid, task[next]->priority, task[next]->counter);

    // 切换到选中的任务
    switch_to(task[next]);
}
```

每次调度时遍历`task`选择剩余运行时间最大的线程运行，并在所有线程没有剩余运行时间时重新根据优先级分配时间片。在线程切换与重新分配时间片时打印相应调试信息。

3. 编译及测试

在根目录`Makefile`中添加如下变量

```
INCLUDE :=  -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include
CF      :=  -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builtin -ffunct
CFLAG   :=  $(CF) $(INCLUDE) -DTEST_SCHED=$(TEST_SCHED)
```

其中编译选项`-DTEST_SCHED`会在编译时添加相应宏定义`TEST_SCHED`

运行测试`make TEST_SCHED=1 run`:

```
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[PID = 1] is running. auto_inc_local_var = 8 current_counter = 7
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. auto_inc_local_var = 9 current_counter = 6
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. auto_inc_local_var = 10 current_counter = 5
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. auto_inc_local_var = 11 current_counter = 4
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. auto_inc_local_var = 12 current_counter = 3
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. auto_inc_local_var = 13 current_counter = 2
[S] Supervisor Mode Timer Interrupt
[PID = 1] is running. auto_inc_local_var = 14 current_counter = 1
[S] Supervisor Mode Timer Interrupt

switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 5 current_counter = 4
Test passed!
    Output: 22222222221111111333342222222222111111113
```

测试顺利通过

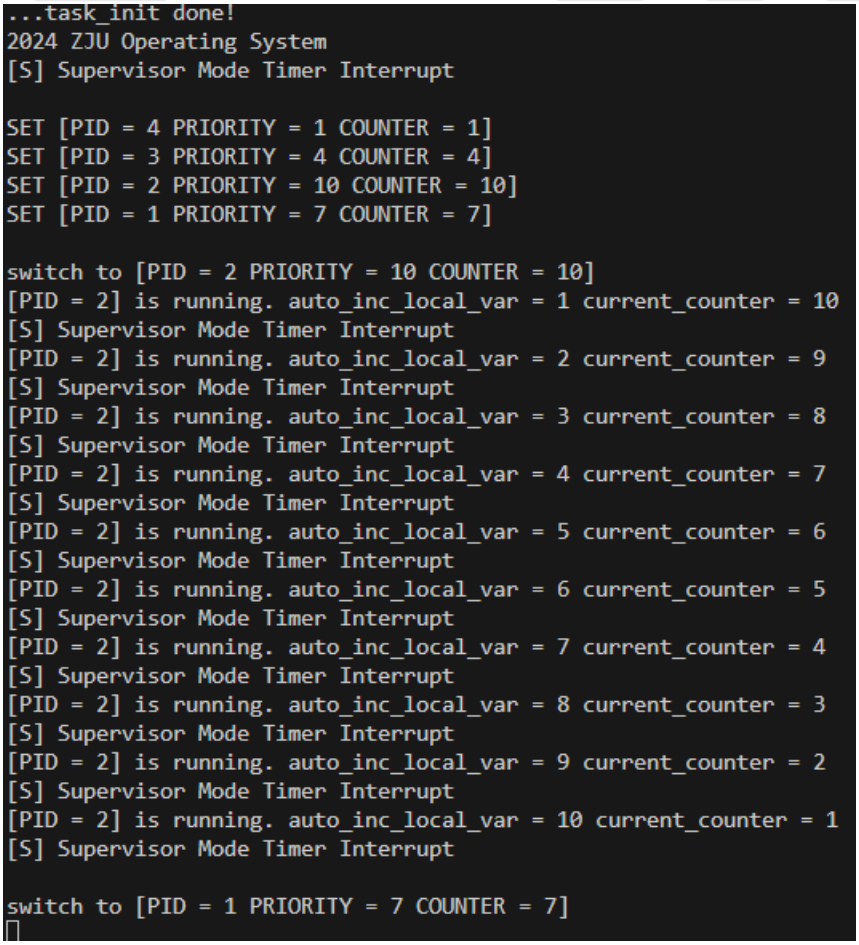
二、思考题

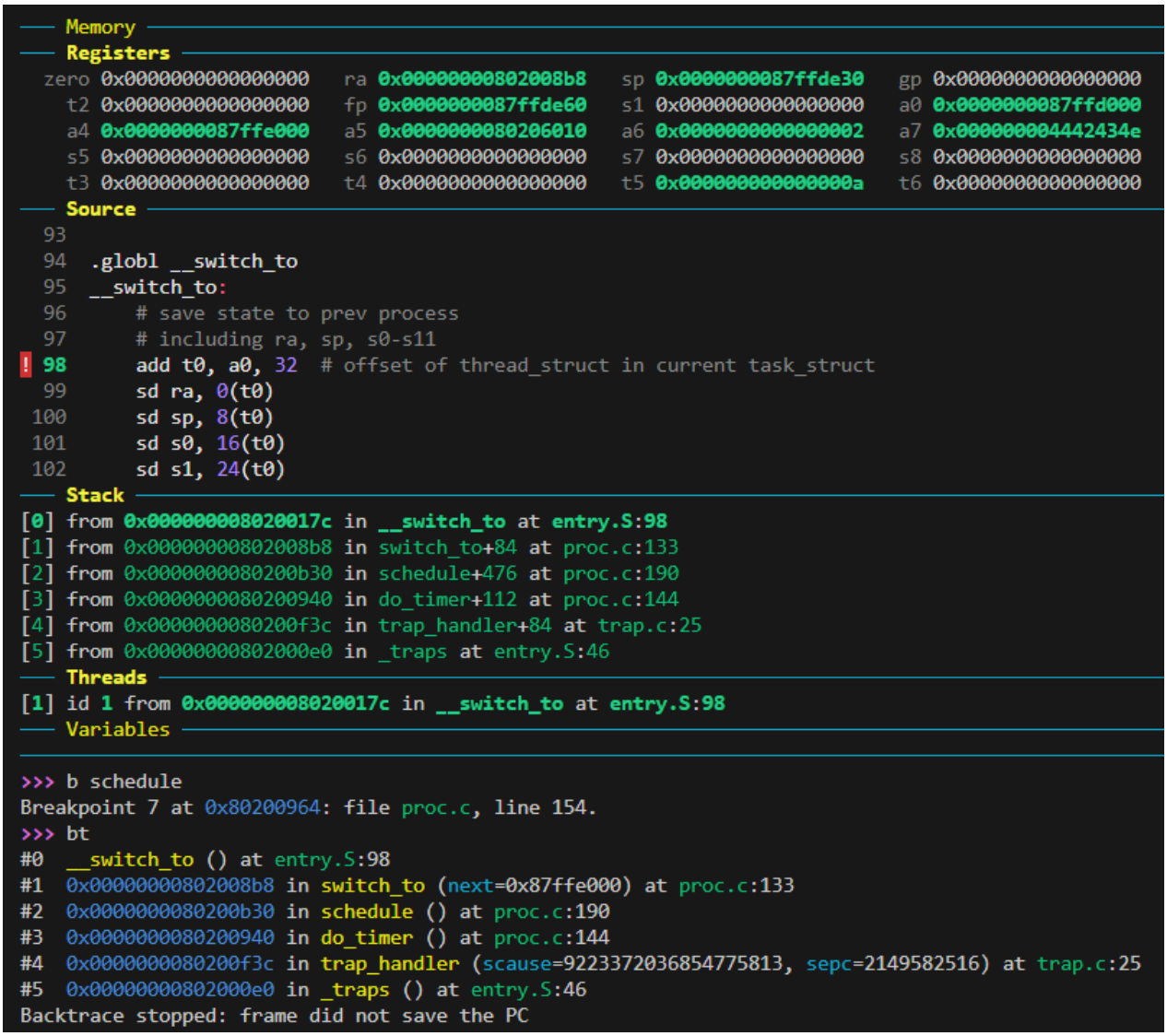
1. **在 RV64 中一共有 32 个通用寄存器，为什么 `__switch_to` 中只保存了 14 个？**
   因为`__switch_to`实际上需要保存恢复的不是当前线程的所有寄存器状态，而是该线程在进入时钟中断已经将所有寄存器保存在栈上后，继续调用时钟中断处理函数直至调用`__switch_to`的`switch_to`函数的状态，因为`switch_to`没有参数返回值没有使用临时寄存器，因此只需要保存从时钟中断`__traps`保存后发生变化的`ra`,`sp`以及`s0-s11`

2. **阅读并理解 `arch/riscv/kernel/mm.c` 代码，尝试说明 `mm_init` 函数都做了什么，以及在 `kalloc` 和 `kfree` 的时候内存是如何被管理的。**
   1. `mm_init`
      `mm_init`调用`kfreerange`函数，将物理内存地址从`_ekernel`到`PHY_END`的空间，按页大小遍历打包成页（起始地址需要`Roundup`到下一个页起始地址保证页对齐），并将每一页内容清空，插入空闲页链表中。
   2. `kalloc`
      `kalloc`在空闲页链表中取出第一个空闲页并清空，返回页地址，如果空闲页链表为空则返回0.
   3. `kfree`
      `kfree`用于释放一个 4 KiB 的物理页，首先判断传入地址是否是页对齐，如果传入的地址不是按 PGSIZE 对齐则将其`Rounddown`。然后用0清空这一页，再将该清空后的空闲页插入空闲页链表的头部。

3. **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 `__switch_to` 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）。**

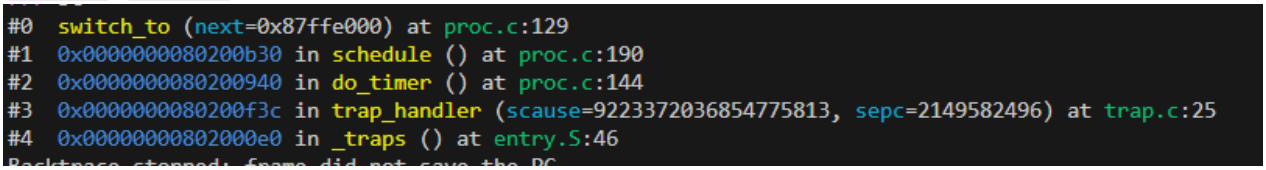
- 使用GDB在`__switch_to`处打上断点，跟踪到Test中第一次线程切换时，使用`bt`打印函数栈帧，发现此时`ra`处在`proc.c`的`switch_to`函数中，所以第一次调用之后保存/恢复的函数返回点是`schedule`中调用的`switch_to`函数，之后再逐层函数栈返回至`_traps`，由`sret`返回`dummy`

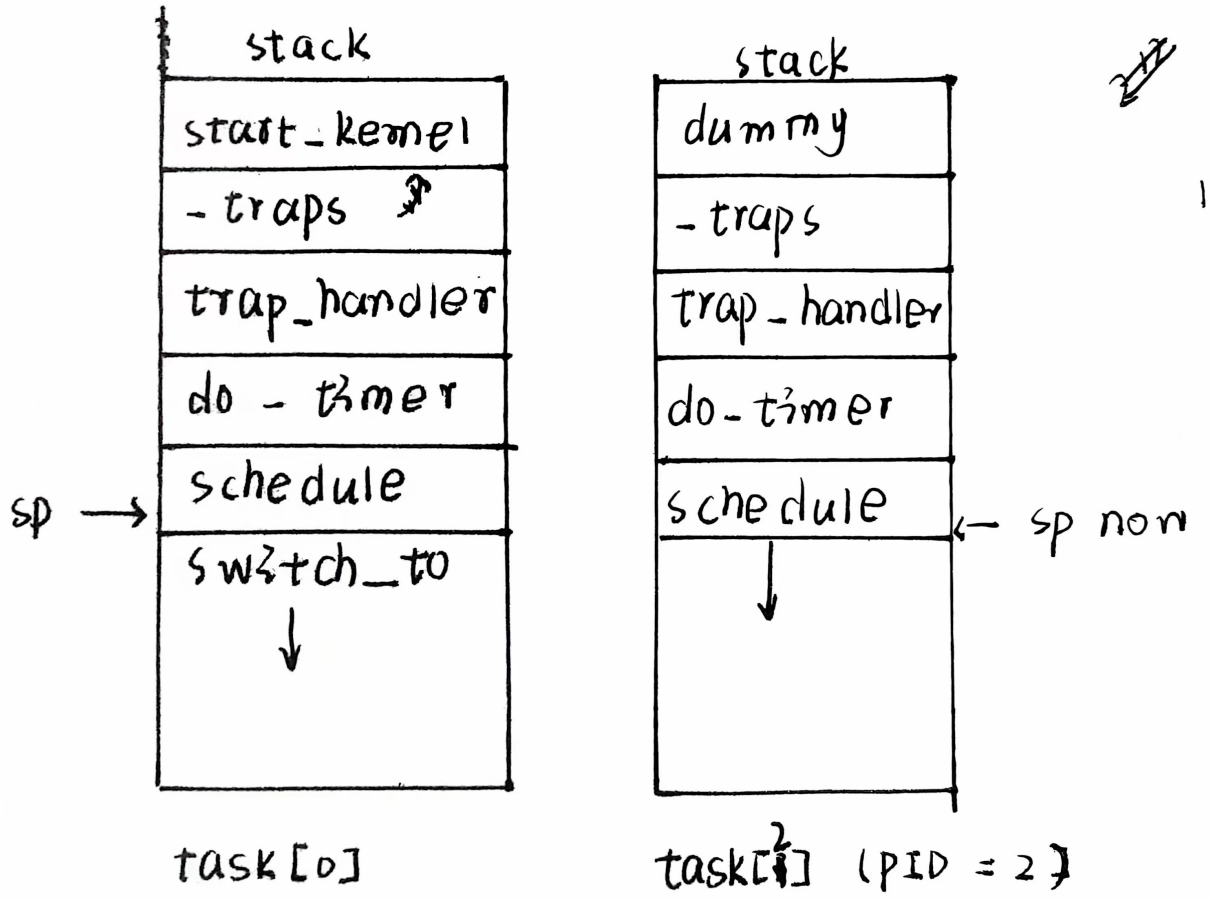




4. **请尝试分析并画图说明 kernel 运行到输出第二次 `switch to [PID ...]` 的时候内存中存在的全部函数帧栈布局。**

   - 可通过 gdb 调试使用 `backtrace` 等指令辅助分析，注意分析第一次时钟中断触发后的 `pc` 和 `sp` 的变化。
     在第一次以及第二次打印`switch to [PID..`时，使用`backtrace`命令打印函数栈帧，均有`task[0]`与`task[2]`(`PID = 2`的线程)栈帧如下，再加上进入中断前的函数，则有全部函数栈帧布局如图二





三、问题与心得

1. 在完成`do_timer`后，在`trap_handler`中调用它，如果将`do_timer`放置在时钟中断处理`clock_set_next_event`前，会导致线程在第一轮调度时所有线程第一次运行`dummy`前就进行一次`counter`自减，线程实际运行时间片比预期少1

```c
void trap_handler(uint64_t scause, uint64_t sepc) {
    // 检查中断类型
    if (scause & SCAUSE_INTERRUPT_MASK) {
        // 这是一个中断
        uint64_t interrupt_id = scause & 0x7FFFFFFF; // 获取中断 ID

        // 检查是否是时钟中断
        if (interrupt_id == SUPERVISOR_TIMER_INTERRUPT_ID) {
            // 打印调试信息
            printk("[S] Supervisor Mode Timer Interrupt\n");

            //线程运行时间自减
            do_timer();

            // 处理时钟中断，设置下一个时钟中断
            clock_set_next_event();
```

```
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 1 current_counter = 9
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 2 current_counter = 8
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 3 current_counter = 7
```

而在第二轮调度重新分配时间片后恢复正常

```
[S] Supervisor Mode Timer Interrupt

SET [PID = 4 PRIORITY = 1 COUNTER = 1]
SET [PID = 3 PRIORITY = 4 COUNTER = 4]
SET [PID = 2 PRIORITY = 10 COUNTER = 10]
SET [PID = 1 PRIORITY = 7 COUNTER = 7]

switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 10 current_counter = 10
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 11 current_counter = 9
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 12 current_counter = 8
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 13 current_counter = 7
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 14 current_counter = 6
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 15 current_counter = 5
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 16 current_counter = 4
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 17 current_counter = 3
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 18 current_counter = 2
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 19 current_counter = 1
[S] Supervisor Mode Timer Interrupt
```

2. 在`dummy`中存在如下代码

```
    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.
```

该代码会使`counter`在等于1时在`dummy`中自减至0，如果不注释掉，则需要在`do_timer`的`counter`自减前多加一条`counter==0`的判断（或者小于0同样进行调度），否则`counter`会变为负值且不能正常进行线程调度

```
void do_timer(void) {
    // 1. 如果当前线程是 idle 线程或当前线程时间片耗尽则直接进行调度
    // 2. 否则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 == task[0]) schedule();
    else {
        //if (current->counter == 0) schedule();
        current->counter -= 1;
        if (current->counter == 0) schedule();
    }
}
```

```
[PID = 2] is running. auto_inc_local_var = 9 current_counter = 2
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 10 current_counter = 0
[S] Supervisor Mode Timer Interrupt
[PID = 2] is running. auto_inc_local_var = 11 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 12 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 13 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 14 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 15 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 16 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 17 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 18 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 19 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 20 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 21 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 22 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 23 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 24 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 25 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 26 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 27 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 28 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 29 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 30 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 31 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 32 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 33 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 34 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 35 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 36 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 37 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 38 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 39 current_counter = -1
[PID = 2] is running. auto_inc_local_var = 40 current_counter = -1
Test failed!
    Expected: 22222222221111111333342222222222111111113
    Got:      22222222222222222222222222222222222222(
```