Lab 2: RV64 进程调度模拟

1 实验目的

结合课堂所学习的相关内容，在上一实验实现时钟中断的基础上进一步实现简单的进程调度。

2 实验内容及要求

- 理解进程调度与进程切换过程
- 利用时钟中断模拟进程调度实验，实现优先级抢占式算法和短作业优先非抢占式算法

请各小组独立完成实验，任何抄袭行为都将使本次实验判为0分。

请跟随实验步骤完成实验并根据文档中的要求记录实验过程，最后删除文档末尾的附录部分，并命名为"**学号1*姓名1*学号2_姓名2_lab2.pdf**"，你的代码请打包并命名为"**学号1*姓名1*学号2_姓名2_lab2**"，文件上传至学在浙大平台。

本实验以双人组队的方式进行，**仅需一人提交实验，**默认平均分配两人的得分（若原始打分为X，则可分配分数为2X，平均分配后每人得到X分）。如果有特殊情况请单独向助教反应，出示两人对于分数的钉钉聊天记录截图。单人完成实验者的得分为原始打分。

姓名	学号	分工	分配分数
徐铭	3210102037	priority算法及其他所有	50%
王思雨	3210106039	sjf算法及其他所有	50%

3 实验步骤

3.1 环境搭建

本实验提供的代码框架结构如图，你可以点击 lab2.zip 进行下载。首先，请下载相关代码，并移动至你所建立的本地映射文件夹中（即 `lab0` 中创建的 `os_experiment` 文件夹）。

```
.
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       └── kernel
│           ├── Makefile
│           ├── clock.c
│           ├── entry.S
│           ├── head.S
│           ├── init.c
│           ├── main.c
```

```
|           ├── print.c
|           ├── sbi.c
|           ├── sched.c
|           ├── test.c
|           ├── trap.c
|           └── vmlinux.lds
└── include
    ├── clock.h
    ├── defs.h
    ├── init.h
    ├── riscv.h
    ├── sbi.h
    ├── sched.h
    ├── stddef.h
    ├── stdio.h
    └── test.h
```

首先请阅读【附录A.进程】确保对实验相关知识有基本的了解。

本实验意在模拟操作系统中的进程调度，实验中将定义结构体模拟进程的资源，利用时钟中断模拟 CPU 时间片以触发调度算法。 本实验中我们定义五个进程，分别是进程 `task[0-4]`。程序执行时，首先将 `task[0-4]` 的剩余运行时间设为 0，然后运行 `init_test_case()` 函数，根据 `counter_priority` 数组为进程分配相应的时间片与优先级，随后进行进程调度，根据运行结果观察调度算法实现正确性。

代码的总体框架工程文件均已给出，需要修改的部分均已进行了标注，主要包括：

- `entry.s`：
  - `__init_epc`：将 `sepc` 寄存器置为 `test()` 函数。
  - `__swtich_to`：需要保存当前进程的执行上下文，再将下一个执行进程的上下文载入到相关寄存器中。
- `sched.c`：
  - `task_init()`：对 `task[0-4]` 进行初始化。
  - `do_timer()`： 在时钟中断处理中被调用，首先会将当前进程的剩余运行时间减少一个单位，之后根据调度算法来确定是继续运行还是调度其他进程来执行。
  - `schedule()`：根据调度算法，考虑所有可运行的进程的优先级和剩余运行时间，按照一定规则选出下一个执行的进程。如果与当前进程不一致，则需要进行进程切换操作。
- `init.c`：
  - `intr_enable()`：打开中断开关，注意与 lab1 的区别，详见3.2
  - `intr_disable()`：关闭中断开关

3.1.3 总体执行流程

1. 打开时钟中断开关，设置中断函数处理地址，设置第一次时钟中断时间，初始化任务的内存空间和结构体。
2. main 函数执行 call_first_process 函数，根据调度算法选择第一个应用程序，进入U态用户应用程序
3. 时钟中断发生，进入 trap_s 中断处理函数
4. trap_s 中断处理函数保护寄存器，然后进入 handler_s 处理函数
5. handler_s 判断是否是时钟中断，是的话进入 do_timer
6. do_timer 函数中，任务时间片减减
7. 如果时间片变成0，执行 schedule，选择出来下一个要执行的任务
8. switch_to 函数，更新 current 变量，然后调用 __switch_to 函数，做实际切换操作。

9. __switch_to 函数把当前进程的一些寄存器保存到进程的结构体中，然后读取要切换到的进程的寄存器。
10. 通过 ra 寄存器跳转到目标进程的执行位置处，开始正常执行下一个任务。

3.2 init.c中打开中断开关（20%）

3.2.1 理解用户应用程序与内核的运行状态与中断控制

在 lab1 中，我们提到，qemu 完成初始化后默认处于 S 态，然后在 S 态运行内核。lab1 中我们始终处于 S 态（进入 OpenSBI 时处于 M 态），为了实现时钟中断，我们在S态把 `sstatus.sie` 设置为1，开启 S 态的中断。现在我们要实现用户应用程序（通过函数模拟），而为了系统的安全，用户应用程序需要运行在U态，来限制用户应用程序的权限。因此在 lab2 中，我们不再只运行在 S 态，而是在 S 态与 U 态进行切换。

在通常的系统中，进入 S 态后，我们会屏蔽掉 S 态的中断，避免再次中断干扰内核的运行，中断结束后返回 U 态时重新开启 S 态中断。这一功能已经由 RISC-V 硬件实现，请阅读 RISC-V特权手册的4.1.1节 中有关 SPP、SIE 与 SPIE 的内容，理解 RISC-V 是如何在不同状态下切换的，在不同状态下切换时中断开关是如何控制的。

我们需要修改 lab1 中中断使能的函数，利用 RISC-V 的机制，使得时钟中断在进入 U 态时被使能。

请完成 `init.c` 中的 `intr_enable()` 和 `intr_disable()`，**并在下方的代码框中补充完整你的代码：（5%）**

```
void intr_enable(void) {
  // TODO: 设置 sstatus[spie] = 1
  write_csr( sstatus, read_csr(sstatus)|0x0000000000000020);
}

void intr_disable(void) {
  // TODO: 设置 sstatus[spie] = 0
  write_csr( sstatus, read_csr(sstatus)&0xFFFFFFFFFFFFFFDF);
}
```

请在完成本次实验后，回答以下问题：（15%）

1. RISC-V 是如何实现 S 态中断的屏蔽与使能的?

   通过S态的CSR寄存器sstatus，sstatus寄存器中的SIE位和SPIE位分别表示当前状态的中断使能和进入当前状态的前一个状态的中断使能，当程序进入S态时，执行`sstatus[SPIE] = status[SIE]`，结束中断时再赋值回去。

   （提示：RISC-V 是如何控制 sstatus.SIE 与 sstatus.SPIE 的，它们分别代表什么含义）

2. 请验证在 lab1 中触发时钟中断时，cpu 处于 S 态，而 lab2 中触发时钟中断时，cpu处于 U 态，请给出sstatus的截图并作出解释：

   lab1：

```
Reading symbols from /home/oslab/os_experiment/lab1/vmlinux...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) b handler_s
Breakpoint 1 at 0x802004dc: file trap.c, line 10.
(gdb) c
Continuing.

Breakpoint 1, handler_s (cause=9223372036854775813) at trap.c:10
10          if (cause >> 63) {
(gdb) i register sstatus
sstatus         0x8000000000006120       -9223372036854750944
(gdb)
```

在lab1中，CPU没有U态，因此触发时钟中断是在S态，随后CPU才调用OpenSBI的函数进入M态进行CSR寄存器的读写和输出；

lab2：

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) b handler_s
Breakpoint 1 at 0x80201428: file trap.c, line 8.
(gdb) continue
Continuing.

Breakpoint 1, handler_s (cause=5, epc=2149658624) at trap.c:8
8           if (cause >> 63 == 1) {
(gdb) i registers sstatus
sstatus         0x8000000000006020       -9223372036854751200
```

在lab2中，我们初始化结束后进入了S态，在`call_first_process()`函数中进入U态，随后才开始时钟中断，因此中断触发时CPU处在U态

（提示：触发中断后，会进入中断处理流程，此时通过 gdb 查看 sstatus 寄存器，可以知道进入中断前 cpu处于什么态。）

3. lab2 中，我们是什么时候第一次进入U态的？请给出第一次进入U态的代码位置或截图：

```c
void call_first_process() {
  // set current to 0x0 and call schedule()
  printf("call first process\n");
  current = (struct task_struct*)(Kernel_Page + LAB_TEST_NUM * PAGE_SIZE);
  current->pid = -1;
  current->counter = 0;
  current->priority = 0;

  schedule();
}
```

在这个函数中，执行完`schedule()`函数中的`switch_to(task[next])`函数，我们就进入了U态

```c
153     show_schedule(next);
154     switch_to(task[next]);
```

（提示：sstatus 记录了进入中断前 cpu 处于什么状态，执行 sret 会回到该状态）

3.3 sched.c 进程调度功能实现（40%）

3.3.1 调度算法切换

本实验需要实现两种调度算法，可以使用 `make priority && make run` 和 `make sjf && make run` 运行不同的调度算法。`make` 默认会编译 `sjf` 算法。

注意如果要切换不同的调度算法，需要先执行 `make clean`。

> 调度算法的切换实际上是通过宏定义及编译选项 `gcc -D` 进行，在 `sched.c` 中使用 `#ifdef`，`#endif` 语句实现版本控制。

3.3.2 实现 task_init()（10%）

只有有了栈，才可以进行函数调用，因为在函数之间切换时需要进行"现场保存"，即保存寄存器信息，方便在结束函数调用时恢复原来的 cpu 状态。

> 编译器在编译高级语言程序时，会自动加上保存现场的指令，因此通常不需要我们对栈进行操作。但是如果我们使用汇编代码（不是内联汇编，而是纯汇编代码，即 .s 文件），那么编译器就不会对我们的代码进行调整，所以如果我们使用汇编代码，就需要考虑寄存器保存。

为了避免进程和进程之间互相影响，每个进程都要有属于自己的栈。内核在创建一个进程时，会创建两个栈，分别是供进程使用的用户栈和供内核使用的内核栈。

在 lab1 中，我们分配了一块内存作为栈，但并没有对用户栈和内核栈进行分离，用户态和内核态使用同一个栈。尽管这样可以正常运行，但是用户应用程序可以通过栈来获取到内核的信息甚至影响内核的运行，这对系统的安全来说是极大的威胁，因此在操作系统中，我们一般把用户栈和内核栈进行分离，这部分将在 `lab4 用户模式` 中实现。

在 lab1 中，我们实现了`trap_s`，在进程进入内核之后，将现场信息保存在栈中。在进程切换时，同样需要保存现场信息，所以我们在每个进程的栈中创建一个结构 `task_struct`，用于保存进程的元信息和进程切换时的寄存器信息。

在本实验中，我们提前给 Task[0-4] 分配好栈空间，并且仍不对内核栈和用户栈作区分。我们这样对物理内存区域进行划分（如下图）：此次实验中我们手动做内存分配，把物理地址空间划分成多个帧 (frame)。即，从 `0x80210000` 地址开始，连续地给此次实验的 5 个 Task [0-4] 做内存分配，我们以 `4KB` 为粒度，按照每个 Task 一帧的形式进行分配，并将 `task_struct` 存放在该帧的低地址部分， 将栈指针 `sp` 指向该帧的高地址。 **（请同学按照下图的内存空间分配地址，不要随意修改，否则有可能影响到最终的实验结果）**

请思考：为什么可以用压栈的方式保存中断上下文，而不用压栈的方式保存 `task_struct`？

> 提示：task 数组是一个存储 task_struct 结构体指针的数组，在一般的 C 语言程序编写中会使用 malloc （操作系统或标准库提供） 来分配空间进行存储，但是我们现在还没有实现动态内存分配，因此只能提前申请内存并硬编码每个 task 的栈空间。实现动态内存管理之后，我们就可以动态地管理栈了。

答：因为中断上下文中保存的都是一些通用寄存器的值，其地址是固定的，且本身不占用程序运行的栈空间；而`task_struct`是每次程序运行都会动态分配的内存，每次地址都不一样，本身已经使用了程序运行的栈空间，

```
-------------------------------------------------------------------------
|             |   Task0   |   Task1   |   Task2   |   Task3   |   Task4   |
|   Kernel    |           |           |           |           |           |
|             |   Space   |   Space   |   Space   |   Space   |   Space   |
-------------------------------------------------------------------------
^             ^           ^           ^           ^           ^
0x80200000    0x80210000  0x80211000  0x80212000  0x80213000  0x80214000
```

为方便起见，我们将 Task [0-4] 进程均设为 `dead_loop()`（见 `test.c`），也就是说每个 Task 执行的代码是一样的，这段代码的汇编被放到上图中 Kernel 里面了。

在 `task_init()` 函数中对实验中的进程进行初始化设置：

- 初始化 `task[0-4]`
- 在初始化时，我们需要将 `thread` 中的 `ra` 指向一个初始化函数 `__init_sepc` (entry.S)，在该函数中我们将 `test` 函数地址赋值给 `sepc` 寄存器，详见【3.3.1】。

请在下方代码框中补充完整你的代码：

```
void task_init(void) {
  puts("task init...\n");

  for(int i = 0; i < LAB_TEST_NUM; ++i) {
    // TODO
    // initialize task[i]
    // get the task_struct based on Kernel_Page and i
    // set state = TASK_RUNNING, counter = 0, priority = 5,
    // blocked = 0, pid = i, thread.sp, thread.ra

    printf("[PID = %d] Process Create Successfully!\n", task[i]->pid);
  }
  task_init_done = 1;
}
```

3.3.3 短作业优先非抢占式算法实现（15%）

- `do_timer()`
  - 将当前所运行进程的剩余运行时间减少一个单位（`counter--`）
  - 如果当前进程剩余运行时间已经用完，则进行调度，选择新的进程来运行，否则继续执行当前进程。
- `schedule()`
  - 遍历进程指针数组 `task`，从 `LAST_TASK` 至 `FIRST_TASK`，在所有运行状态(TASK_RUNNING)下的进程剩余运行时间最小的进程作为下一个执行的进程。若剩余运行时间相同，则按照遍历的顺序优先选择。**【注意， 测试代码只赋值了 5 个任务，但 `LAST_TASK` 是 64，遍历的时候注意先判断指针是否为空】**
  - 如果所有运行状态下的进程剩余运行时间都为0，则通过 `init_test_case()` 函数重新为进程分配运行时间与优先级，然后再次调度。

请在下方代码框中补充完整你的代码：

```
#ifdef SJF
```

```
// simulate the cpu timeslice, which measn a short time frame that gets assigned
// to process for CPU execution
void do_timer(void) {
  if (!task_init_done) return;

  printf("[*PID = %d] Context Calculation: counter = %d,priority = %d\n",
         current->pid, current->counter, current->priority);

  // current process's counter -1, judge whether to schedule or go on.
  // TODO
  current->counter--;
  if(current->counter<=0) schedule();
  //else do_timer();
}


// Select the next task to run. If all tasks are done(counter=0), reinitialize
all tasks.
void schedule(void) {
  unsigned char next;
  // TODO
  next=NR_TASKS;
  struct task_struct* it=LAST_TASK;
  for(int index=NR_TASKS-1;index>=0;index--){
    it=task[index];
    if(it== 0) continue;
    if(it->state==TASK_RUNNING&&it->counter>0){
      if(next==NR_TASKS) next=index;
      if(it->counter<task[next]->counter) next=index;
    }
  }
  if(next==NR_TASKS){
    init_test_case();
    call_first_process();//本来写的是schedule
  }
  else{
    show_schedule(task[next]->pid);

    switch_to(task[next]);
  }

}

#endif
```

3.3.4 优先级抢占式算法实现（15%）

- `do_timer()`
  - 将当前所运行进程的剩余运行时间减少一个单位（`counter--`）
  - 每次 `do_timer()` 都进行一次抢占式优先级调度。
- `schedule()`
  - 遍历进程指针数组 `task`，从 `LAST_TASK` 至 `FIRST_TASK`，调度规则如下：
    - 高优先级的进程，优先被运行（值越小越优先）。
    - 若优先级相同，则选择剩余运行时间少的进程（若剩余运行时间也相同，则按照遍历的顺序优先选择）。

- 如果所有运行状态下的进程剩余运行时间都为 0，则通过 `init_test_case()` 函数重新为进程分配运行时间与优先级，然后再次调度。

请在下方代码框中补充完整你的代码：

```
#ifdef PRIORITY

// simulate the cpu timeslice, which measn a short time frame that gets assigned
// to process for CPU execution
void do_timer(void) {
  if (!task_init_done) return;

  printf("[*PID = %d] Context Calculation: counter = %d,priority = %d\n",
         current->pid, current->counter, current->priority);

  // current process's counter -1, judge whether to schedule or go on.
  // TODO
  current->counter--;
  schedule();
}

// Select the task with highest priority and lowest counter to run. If all tasks
are done(counter=0), reinitialize all tasks.
void schedule(void) {
  unsigned char next = 0;
  // // TODO
  printf("schedule begin\n");
  int zero_counter_num = 0, max_priority = 65535, min_counter = 65535;
  for( int i = 0; i < NR_TASKS; i++ )
    if( !task[i] || task[i]->state != TASK_RUNNING || task[i]->counter <= 0 )
      zero_counter_num++;
  if( zero_counter_num == NR_TASKS ) init_test_case();

  for( int i = 0; i < NR_TASKS; i++ ) {
    if( !task[i] || task[i]->state != TASK_RUNNING || task[i]->counter <= 0 )
      continue;
    else {
      if( task[i]->priority < max_priority ) {
        max_priority = task[i]->priority;
        min_counter = task[i]->counter;
        next = i;
      }else if( task[i]->priority == max_priority && task[i]->counter <
min_counter ) {
        max_priority = task[i]->priority;
        min_counter = task[i]->counter;
        next = i;
      }
    }
  }

  if( current->pid != task[next]->pid ) {
    printf(
        "[ %d -> %d ] Switch from task %d[%lx] to task %d[%lx], prio: %d, "
        "counter: %d\n",
        current->pid, task[next]->pid, current->pid,
        (unsigned long)current->thread.sp, task[next]->pid,
```

```
        (unsigned long)task[next], task[next]->priority, task[next]->counter);
  }

  show_schedule(next);
  switch_to(task[next]);
}

#endif
```

3.4 实现 entry.s（20%）

3.4.1 实现 __switch_to 函数（10%）

实现切换进程的过程，`a0` 参数为 `struct task_struct* prev`，即切换前进程的地址。 `a1` 参数为 `struct task_struct* next` ，即切换后进程的地址。

> 如果你不知道为什么 `a0` 寄存器和 `a1` 寄存器存储的是这两个参数的话，请参考 RISC-V中文手册 第 3.2 节。

该函数需要做以下事情。

- 保存当前的 `ra`， `sp`，`s0~s11` 寄存器到当前进程的 `thread_struct` 结构体中。
- 将下一个进程的 `thread_struct` 的数据载入到 `ra`，`sp`，`s0~s11` 中。

保存 `sp` 和 `s0~s11` 是因为这些是 `callee saved registers`，其他的寄存器会被编译器自动保存到栈上，但是 `ra` 是 `caller saved register` 为什么也需要手动保存呢？这是因为我们需要在 `__switch_to` 中完成控制流的切换，不同的进程可能会在不同的位置调用 `__switch_to` （尽管现在只能通过 `switch_to()` 一种方式），所以要手动完成 `ra` 的切换。

请在下方代码框中补充完整你的代码：

```
# entry.S
__switch_to:
    li    a4,  40
    add   a3, a0, a4
    add   a4, a1, a4
    # TODO: Save context into prev->thread
    sd ra, 0(a3)
    sd sp, 8(a3)
    sd s0, 16(a3)
    sd s1, 24(a3)
    sd s2, 32(a3)
    sd s3, 40(a3)
    sd s4, 48(a3)
    sd s5, 56(a3)
    sd s6, 64(a3)
    sd s7, 72(a3)
    sd s8, 80(a3)
    sd s9, 88(a3)
    sd s10, 96(a3)
    sd s11, 104(a3)
```

```
    # TODO: Restore context from next->thread
    ld ra, 0(a4)
    ld sp, 8(a4)
    ld s0, 16(a4)
    ld s1, 24(a4)
    ld s2, 32(a4)
    ld s3, 40(a4)
    ld s4, 48(a4)
    ld s5, 56(a4)
    ld s6, 64(a4)
    ld s7, 72(a4)
    ld s8, 80(a4)
    ld s9, 88(a4)
    ld s10, 96(a4)
    ld s11, 104(a4)

    # return to ra22
    ret
```

为什么要设置前三行汇编指令?
答：因为每个task的栈空间首先存了`task_struct`，每个`task_struct`占了40个字节，因此这里先用三行汇编指令对要保存状态的两个进程的起始地址+40，算出各自寄存器保存的开始地址，方便后续汇编的书写；

3.4.2 __init_epc（10%）

为什么本实验在初始化时需要为 Task[0-4] 的 ra 寄存器指定为`__init_sepc`？

对于正常的进程切换流程（假设从进程A切换到进程B），控制流如下：

```
executing A ->
    Time Interrupt -> 触发中断
        trap_s -> 将中断上下文压入A的栈中
            hander_s -> 进行进程调度
                ... ->
                    switch_to（in A） -> 调用__switch_to
                            __switch_to ->  将A的进程上下文保存，将B的进程上下文加载到寄存器，此时发生了A的栈到B的栈的切换
                    switch_to（in B） <- 此时已经进入B的控制流
                ... <-
            hander_s <-
        trap_s <- 从B的栈中弹出中断上下文，sret回到B的程序代码中
executing B <-
```

可以看出能够进行进程切换，重要的一点是在栈上具有每个进程的控制流和上下文信息，这样在我们切换栈之后，就可以恢复原来的控制流。然而在第一次启动某个进程时，该进程的栈上没有任何信息，因此不能通过仅仅切换栈来启动某个进程，还需要手动设置`sepc`进入该进程的代码。

具体的方法是，将其任务结构体中的`ra`寄存器填写为`__init_epc`的地址，这样，在`__switch_to`函数返回的时候，就会直接跳转到`__init_epc`函数中。而该函数在`sret`时返回到`test()`函数，也就是进程实际的工作函数中。

将栈的内容（部分）展示出来如下：

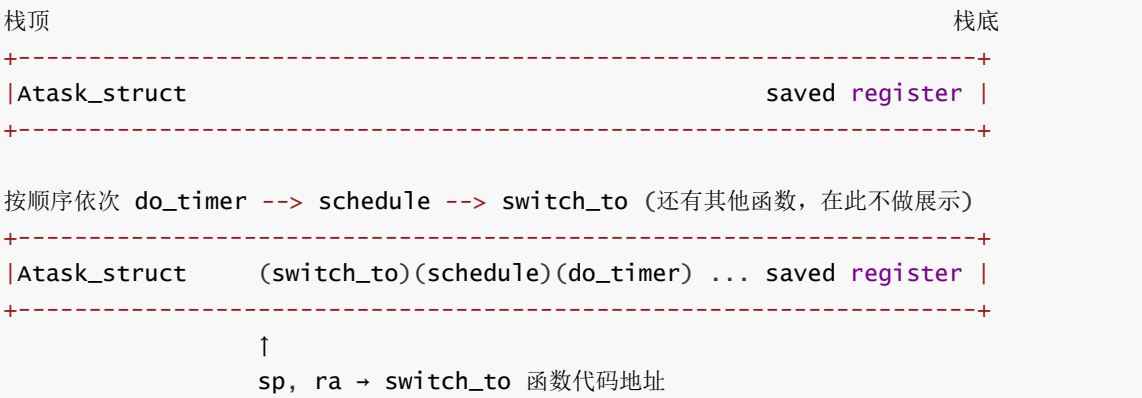
```
================================================================================
=====
```

==========A已经启动，B第一次启动==========

==========

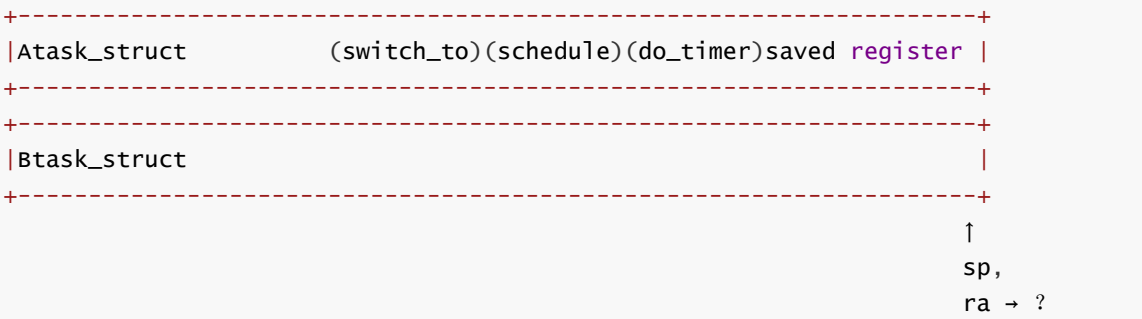
A进程执行中，发生时钟中断

在trap_s中将中断上下文压栈



按顺序依次 do_timer --> schedule --> switch_to（还有其他函数，在此不做展示）

现在执行 __switch_to，保存了 A 进程的进程上下文，加载了 B 进程的进程上下文



__switch_to 通过 ret 跳转到 B 进程的 ra 处，此时的 ra 的值是我们初始化的值

1. 现在处于中断处理的 S 模式，开始执行代码的时候需要用 sret 指令返回 sepc 开始执行。
2. 第一次进入 B 进程，B 进程栈上什么也没有

因此需要一个特殊的启动函数 init_epc ，他需要完成两件事情

1. sepc 设置成任务真正要开始执行的代码地址
2. 直接利用 sret 开始执行

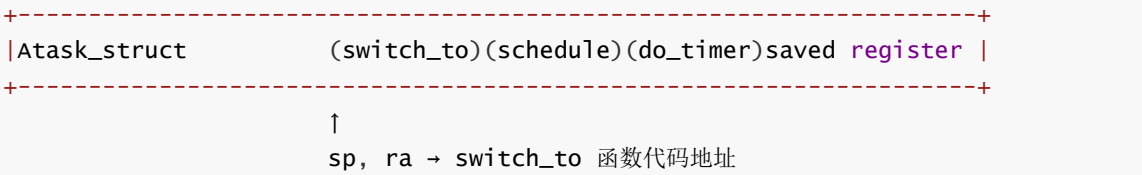
==========

==========一个正常的切换流程==========

==========

A进程执行中，发生时钟中断

按顺序依次 do_timer --> schedule --> switch_to 函数



```
现在执行 __switch_to，保存了 A 进程的 callee saved，加载了 B 进程的 callee saved
+-------------------------------------------------------------------+
|Atask_struct          (switch_to)(schedule)(do_timer)saved register |
+-------------------------------------------------------------------+
+-------------------------------------------------------------------+
|Btask_struct          (switch_to)(schedule)(do_timer)saved register |
+-------------------------------------------------------------------+
                       ↑
                       sp，ra → switch_to 函数代码地址

1. __switch_to 通过 ret 跳转到 B 进程的 ra 处
2. 继续执行，结束 switch_to 函数
3. 继续执行，结束 schedule 函数
4. 继续执行，结束 do_timer 函数
5. 继续执行 trap_s 的剩余部分

trap_s 剩余部分代码，恢复 B 进程的 saved register
+-------------------------------------------------------------------+
|Btask_struct                                        saved register |
+-------------------------------------------------------------------+
                                                     ↑
                                                     sp

最后，trap_s 使用 sepc 跳转回中断发生前代码执行的位置。
（因此 trap_s 需要保护 sepc 寄存器，即将 sepc 寄存器也保存在栈上一份）
```

请在下方代码框中补充完整你的代码：

```
.globl __init_sepc
__init_sepc:
    # TODO
    la  t0, test
    csrw sepc, t0

    sret
```

3.5 编译及测试（20%）

运行`make priority && make run`和`make sjf && make run`进行编译测试，请对`main.c`做修改，确保输出本组成员的学号与姓名。

注意如果要切换不同的调度算法，需要先执行`make clean`。

3.5.1 运行结果（20%）

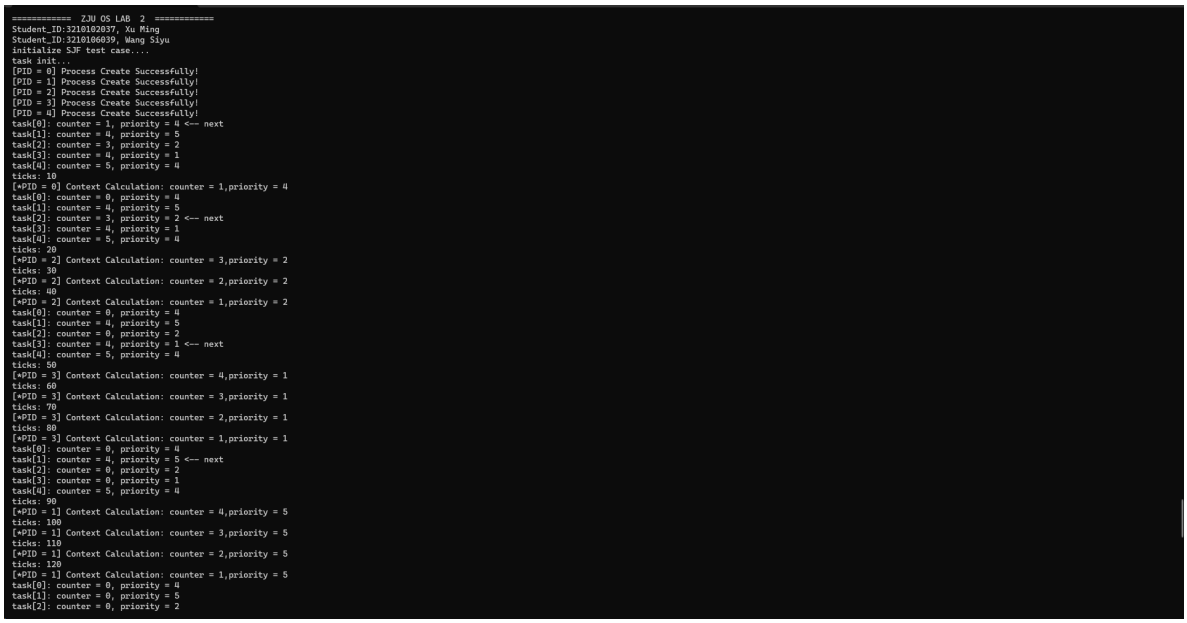
在`test.c`中的`counter_priority`控制着进程的优先级以及时间片，共有三组测例，请将第一组测例的运行结果截图放在报告中，截图必须截到本组成员的学号与姓名。并将全部测例的运行结果填到下面的表中。

注意：实验验收时将修改`test.c`中的`counter_priority`数组改变进程的优先级及剩余运行时间，确保代码在不同运行情况下的正确性。同学们可以自行设置测试用例进行实验以确保代码在边界情况的准确性，请将测试用例按照格式补充在表格中。

请在此附上你的代码运行结果截图。

短作业优先非抢占式算法

答:



优先级抢占式算法

答:

短作业优先非抢占式算法

测试目标	测试用例{counter,priority}	预期结果	实际结果
随机1	{1, 4}, {4, 5}, {3, 2}, {4, 1}, {5, 4}	1, 3, 4, 2, 5	1, 3, 4, 2, 5
随机2	{3, 1}, {2, 1}, {4, 1}, {1, 1}, {1, 2}	5, 4, 2, 1, 3	5, 4, 2, 1, 3
相同剩余运行时间，以遍历顺序执行	{1, 1}, {1, 1}, {1, 1}, {1, 1}, {1, 1}	5, 4, 3, 2, 1	5, 4, 3, 2, 1

优先级抢占式算法

测试情况	测试用例{counter,priority}	预期结果	实际结果
随机1	{1, 4}, {4, 5}, {3, 2}, {4, 1}, {5, 4}	4, 3, 1, 5, 2	4, 3, 1, 5, 2
随机2	{3, 1}, {2, 1}, {4, 1}, {1, 1}, {1, 2}	4, 2, 1, 3, 5	4, 2, 1, 3, 5
相同剩余运行时间，以遍历顺序执行	{1, 1}, {1, 1}, {1, 1}, {1, 1}, {1, 1}	5, 4, 3, 2, 1	5, 4, 3, 2, 1

4 讨论和心得

本次实验总体来说还是比较简单的，我遇到的主要问题有两个，首先是一开始没搞清楚这里的调度算法与时钟中断之间的关系，以及具体要干什么，实验会写的十分糊涂。后来，重新会看lab1的实验指导和lab2的总体执行流程，我明白了进程调度就是在时钟中断中进行的。通过main函数中的`call_first_process()`，我们其实已经进入了一个进程的执行过程，只不过这个进程执行的是死循环。然后在循环过程中，发生了时钟中断，再通过我们的调度函数去判断是否需要切换进程。

第二个问题是，调度函数有一些边界情况比较难想清楚，所以我通过设置一些边界条件，比如`{0, 1}, {0, 1}, {0, 1}, {0, 1}, {0, 1}`作为首组tasks的测试情况，成功找出了我调度函数中的一些考虑不周全的地方，例如，应该先判断当前tasks是否所有counter均为0，以此来判断是否需要重新分配测试样例，再去找出下一个要切换的进程，否则，由于`do_timer()`函数是先对当前进程`counter--`的，对前面说的这种测试样例，会产生负数counter，则会产生bug；其次，不应该走捷径，用当前进程作为标准去选择下一个进程，而应该从全局考虑。