# Lab3 RISC-V64进程调度模拟

## 1. 实验目的

在lab2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

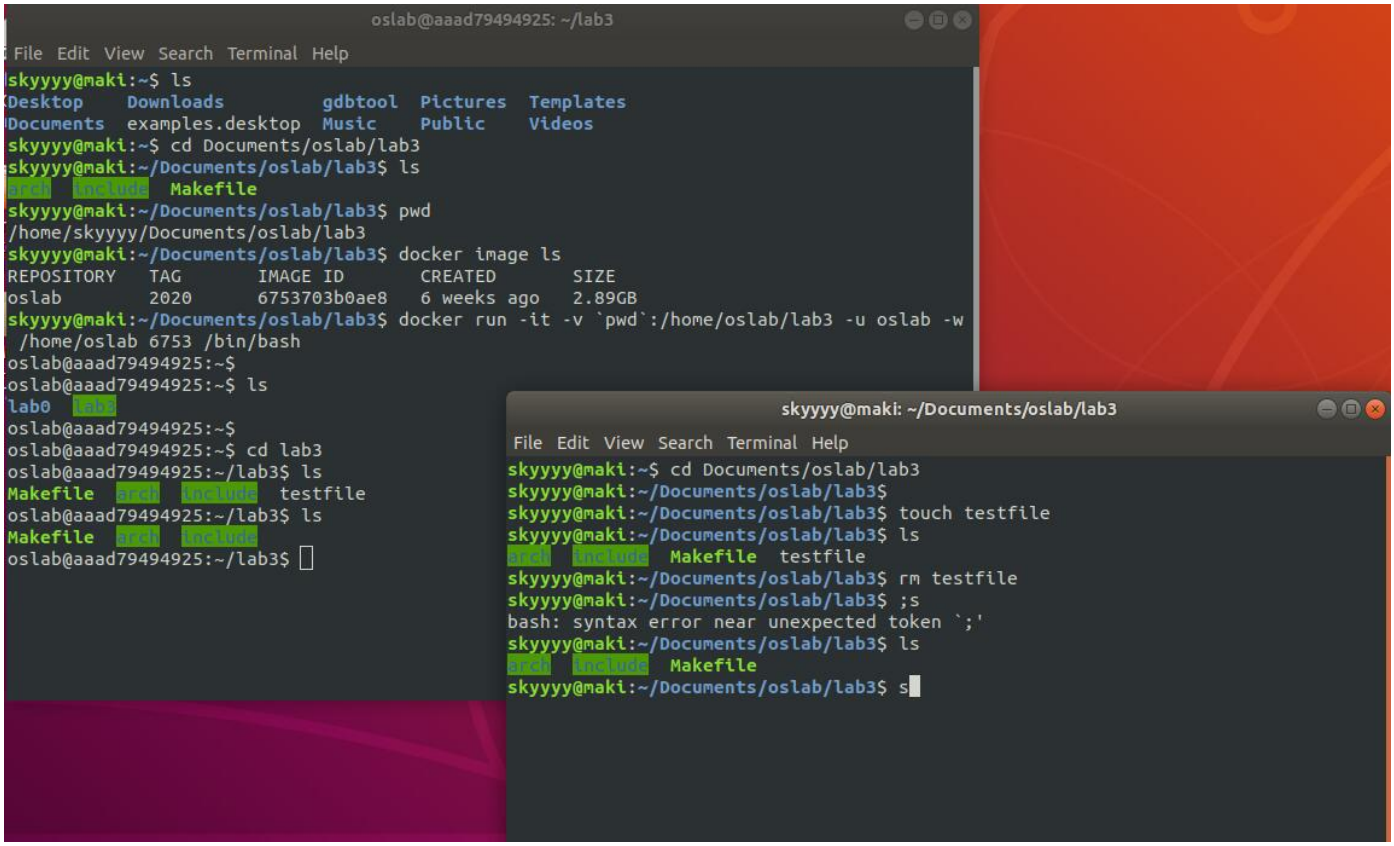
本次实验为单人完成。

## 3. 实验步骤

实验代码位于 5. Code

### 3.1 环境搭建

#### 3.1.1 建立映射




#### 3.1.2 组织文件结构

```
lab3
├── arch
│   └── riscv
│       ├── kernel
```

```
│       │   ├── clock.c
│       │   ├── entry.S
│       │   ├── head.S
│       │   ├── init.c
│       │   ├── main.c
│       │   ├── Makefile
│       │   ├── print.c
│       │   ├── sbi.c
│       │   ├── sched.c
│       │   ├── test.c
│       │   ├── trap.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── defs.h
│   ├── riscv.h
│   ├── sched.h
│   ├── stddef.h
│   ├── stdio.h
│   └── test.h
└── Makefile
```

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。 程序执行时，首先将`task[1-4]`的剩余运行时间设为0，通过时钟中断处理函数引起第一次调度，`task0`使用一个时间片运行`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。

- `entry.s`:
  - `__init_epc`: 将`sepc`寄存器置为`test()`函数。
  - `trap_s`: 首先保存当前的寄存器，随后调用`handle_s`函数，最后恢复寄存器。
  - `__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`task0`、`task[1-4]`进行初始化，设置`current`变量指示当前执行的进程。
  - `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。如果当前进程均已执行完毕，为`task0`新增一个时间片以重新分配优先级和运行时间。
  - `switch_to`: 进程切换，更新`current`变量，调用`__switch_to`函数切换进程的执行上下文。

### 3.1.3 实验相关背景

#### a. 进程

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，称之为程序（Program）；进程（Process）则是正在运行并使用计算机资源的程序。

**b. 线程**

线程的概念将“正在运行”的动态特性从进程中剥离出来，是一个借助 CPU 和栈的执行流。进程主要功能是作为资源的分配单位，管理页表、文件、网络等资源。而一个进程的多个线程则共享这些资源，专注于执行，从而作为执行的调度单位。

本次实验假设操作系统一个进程只有一个线程，简化对进程的调度。当操作系统引入线程后，**进程是资源分配的最小单位，线程是调度的最小单位**。

**c. 进程的表示**

本次实验采取一种基础的实现，每个进程包括：

- **进程ID**：用于唯一确认一个进程。
- **运行时间片**：为每个进程分配的运行时间。
- **优先级**：在调度时配合调度算法，选出下一个执行的进程。
- **运行栈**：每个进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**：当进程不在执行状态时，需要保存其上下文（即状态寄存器的值），使得能够将其恢复，继续运行。

**d. 本次实验进程的调度与切换**

- 在每次**时钟中断处理**时，操作系统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。
- 在**进程调度**时，操作系统会对所有可运行的进程进行判断，按照一定规则选出下一个执行的进程。如果没有符合条件的进程，则会对所有进程的优先级和剩余运行时间相关属性进行更新，再重新选择。最终将选择得到的进程与当前进程切换。
- 在**进程切换**的过程中，首先需要保存当前进程的执行上下文，再将将要执行进程的上下文载入到相关寄存器中，至此完成了进程的调度与切换。

# 3.2 sched.c进程调度功能实现

## 3.2.1 调度算法切换

本实验需要实现两种调度算法，可以通过宏定义及编译选项`gcc -D`进行控制。

- 通过修改`lab3/Makefile`中的`CFLAG = ${CF} ${INCLUDE} -DSJF / -DPRIORITY`以控制实验所采用的调度算法。
  - -DSJF 短作业优先式
  - -DPRIORITY 优先级抢占式

```
#TASK_MM = -DSJF
TASK_MM = -DPRIORITY

CFLAG = ${CF} ${INCLUDE} ${TASK_MM}
```

- 在`sched.c`中使用`#ifdef`，`#endif`语句控制进程调度的代码实现。

### 3.2.2 实现task_init()

本次实验使用了多个进程，需要对无力内存区域进行划分。此次实验中手动做内存分配，把物理地址空间划分成为多个帧（frame）。从`0x80210000`地址开始连续地给此次实验的 4 个 Task [1-4] 以及内核栈Task[0]做内存分配。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。内存空间分配地址如下图。

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

可将Task [1-4] 看作是从 0 号进程 (Task [0] ) fork 出的子进程，后续实验的实现将主要考虑如何对这四个子进程进行调度。为方便起见，将Task [1-4] 进程均设为`dead_loop()`（见`test.c`）。

在`task_init()`函数中对实验中的进程进行初始化设置：

- 初始化`current`与`task[0]`
  - 设置`current`指向`Task0 Space`的基地址（0x80210000）。
- 设置`task[0]` 为 `current`。
  - 初始化`task[0]`中的成员变量：`state` = TASK_RUNNING；`counter` = 1；`priority` = 5；`blocked` = 0；`pid` = 0。

    **Lab3中task只需要TASK_RUNNING一种状态**
  - 设置`task[0]`的`thread`中的`sp`指针为 Task0 Space基地址 + 4KB的偏移。
- 参照`task[0]`的设置，对`task[1-4]`的成员变量完成初始化设置
  - `counter` = 0
  - `priority` = 5
- 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】。

### 3.2.3 短作业优先非抢占式算法实现

- `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`(不包括`FIRST_TASK`，即task[0])，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。
  - 如果所有运行状态下的进程剩余运行时间都为0，给`task 0`分配一个时间片，由其通过`init_test_case()`函数为其余进程分配剩余运行时间与优先级，重新开始调度。

  代码

### 3.2.4 优先级抢占式算法实现

- `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`(不包括`FIRST_TASK`)，调度规则如下：
    - 高优先级的进程，优先被运行。
    - 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，给`task 0`分配一个时间片，由其通过`init_test_case()`函数为其余进程分配剩余运行时间与优先级，重新开始调度。

代码

### 3.2.5 实现 `switch_to(struct task_struct* next)`

判断下一个执行的进程`next`与当前的进程`current`是否为同一个进程，如果是同一个进程，则无需做任何处理。如果不一致，则更新`current`变量，并调用`__switch_to(struct task_struct* prev,struct task_struct* next)`(`entry.S`)。

代码

## 3.3 实现entry.s

代码

### 3.3.1 __init_epc

运行中的进程遇到时钟中断后，需要将当前运行线程的上下文环境保存在栈上(`entry.S`中的`trap_s`函数)，当线程再次被调度时，再将上下文从栈上恢复。但是对于新创建的、初次被进行调度的任务，其栈上还没有可以被`trap_s`函数用来恢复上下文的寄存器信息，所以需要为其第一次调度提供一个特殊的返回函数，将其寄存器信息填写到栈上。

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，在`sret`时返回`test()`函数作为该任务的代码段，此时由于还未对时钟中断进行处理，会重新回到`trap_s`，新任务的寄存器信息便被填到了栈上。

**Q1：使用gdb调试观察，说明第一个测试用例中，`task1`切换到`task2`，`task2`初次被调度时的每个函数的运行内容及函数间的跳转过程。**

- task1执行中，时钟中断触发。
- trap_s: 保存上下文并调用中断函数`handler_s(scause)`。
- handler_s: 时钟中断的实现，并且每10ticks（0.1s）调用一次`do_timer()`。
- do_timer: 对task1的剩余运行时间counter减1个单位，在SJF调度算法下若task1对counter==0，则调用`schedule()`进行进程调度。
- schedule: 在当前所以`TASK_RUNNING`状态的task中寻找下一个执行（task2），调用`swicth_to`实现。
- switch_to->__switch_to: 调用`__switch_to`保存task1的线程，加载task2点线程。
- __init_epc: 将sepc设置为test函数的地址，sret指令使得pc被填充为sepc的值，即进入test函数。
- trap_s: 时钟中断未处理，再次进入trap_s保存上下文。

- handler_s: 直接return。
- trap_s: 恢复上下文。
- task2执行。

### 3.3.2 在trap_s中断处理中添加保存epc的指令

当异常发生时，`epc`寄存器会保存当前的异常返回地址，该地址是与当前task相关的。由于task会在异常处理中发生调度，所以我们需要在中断处理时保存当前的`epc`，当中断处理完成之后再载入`epc`。本实验发生在S模式，因此使用的是`sepc`寄存器，实现同`lab2`。

### 3.3.3 __switch_to(struct task_struct* prev, struct task_struct* next)

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。

- 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。

**Q2：为什么要设置前三行汇编指令?**

```
li    a4,  40
add   a3, a0, a4
add   a4, a1, a4
```

进程数据结构为：

```
struct task_struct {
     long state;    // 进程状态 Lab3中进程初始化时置为TASK_RUNNING
     long counter;  // 运行剩余时间
     long priority; // 运行优先级 1最高 5最低
     long blocked;
     long pid;      // 进程标识符
    // Above Size Cost: 40 bytes

    struct thread_struct thread; // 该进程状态段
};
```

由于前5个成员都为long类型，所以该结构体成员在内存中连续存储。5个long类型成员大小为40字节，因此40为成员`struct thread_struct thread`地址相对于结构体指针的偏移量。前三行指令方便直接访问修改`thread_struct`中的成员。

## 3.5 编译及测试

### 1. 运行截图

展示默认设置的第一个测试用例。

**a. 短作业优先非抢占式算法（SJF）**

```
MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
ZJU OS LAB 3      Student:3190102362 王奕天
task init...
[PID = 1] Process Create Successfully!
[PID = 2] Process Create Successfully!
[PID = 3] Process Create Successfully!
[PID = 4] Process Create Successfully!
initialize test case....

******************************************************
Task0 is running to allocate new priority and counter.
[*PID = 0] Context Calculation: counter = 1,priority = 5
[ 0 -> 1 ] Switch from task 0[0000000080211000] to task 1[0000000080211000], prio: 4, counter: 1
[*PID = 1] Context Calculation: counter = 1,priority = 4
[ 1 -> 2 ] Switch from task 1[0000000080212000] to task 2[0000000080212000], prio: 5, counter: 2
[*PID = 2] Context Calculation: counter = 2,priority = 5
[*PID = 2] Context Calculation: counter = 1,priority = 5
[ 2 -> 3 ] Switch from task 2[0000000080213000] to task 3[0000000080213000], prio: 2, counter: 3
[*PID = 3] Context Calculation: counter = 3,priority = 2
[*PID = 3] Context Calculation: counter = 2,priority = 2
[*PID = 3] Context Calculation: counter = 1,priority = 2
[ 3 -> 4 ] Switch from task 3[0000000080214000] to task 4[0000000080214000], prio: 1, counter: 4
[*PID = 4] Context Calculation: counter = 4,priority = 1
[*PID = 4] Context Calculation: counter = 3,priority = 1
[*PID = 4] Context Calculation: counter = 2,priority = 1
[*PID = 4] Context Calculation: counter = 1,priority = 1
Current Setting is done, use task0 to initialize a new setting.
[ 4 -> 0 ] Switch from task 4[0000000080215000] to task 0[0000000080210000], prio: 5, counter: 1

******************************************************
Task0 is running to allocate new priority and counter.
[*PID = 0] Context Calculation: counter = 1,priority = 5
```

**b. 优先级抢占式算法（PRIORITY）**

```
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
ZJU OS LAB 3      Student:3190102362 王奕天
task init...
[PID = 1] Process Create Successfully!
[PID = 2] Process Create Successfully!
[PID = 3] Process Create Successfully!
[PID = 4] Process Create Successfully!
initialize test case....

******************************************************
Task0 is running to allocate new priority and counter.
[*PID = 0] Context Calculation: counter = 1,priority = 5
[ 0 -> 4 ] Switch from task 0[0000000080211000] to task 4[0000000080214000], prio: 1, counter: 4
[*PID = 4] Context Calculation: counter = 4,priority = 1
[*PID = 4] Context Calculation: counter = 3,priority = 1
[*PID = 4] Context Calculation: counter = 2,priority = 1
[*PID = 4] Context Calculation: counter = 1,priority = 1
[ 4 -> 3 ] Switch from task 4[0000000080215000] to task 3[0000000080213000], prio: 2, counter: 3
[*PID = 3] Context Calculation: counter = 3,priority = 2
[*PID = 3] Context Calculation: counter = 2,priority = 2
[*PID = 3] Context Calculation: counter = 1,priority = 2
[ 3 -> 1 ] Switch from task 3[0000000080214000] to task 1[0000000080211000], prio: 4, counter: 1
[*PID = 1] Context Calculation: counter = 1,priority = 4
[ 1 -> 2 ] Switch from task 1[0000000080212000] to task 2[0000000080212000], prio: 5, counter: 2
[*PID = 2] Context Calculation: counter = 2,priority = 5
[*PID = 2] Context Calculation: counter = 1,priority = 5
Current Setting is done, use task0 to initialize a new setting.
[ 2 -> 0 ] Switch from task 2[0000000080213000] to task 0[0000000080210000], prio: 5, counter: 1

******************************************************
Task0 is running to allocate new priority and counter.
[*PID = 0] Context Calculation: counter = 1,priority = 5
```

### 2. 测试用例

#### a. 短作业优先非抢占式算法

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。

| 测试目标 | 测试用例(counter_priority) | 预期结果 | 实际结果 |
| --- | --- | --- | --- |
| 随机1 | {1,4},{2,5},{3,2},{4,1} | 1>2>3>4 | 1>2>3>4 |
| 随机2 | {5,1},{3,3},{4,1},{2,1} | 4>2>3>1 | 4>2>3>1 |
| 相同剩余运行时间，以遍历顺序执行 | {1,1},{1,1},{1,1},{1,1} | 4>3>2>1 | 4>3>2>1 |
| 存在剩余运行时间为0 | {4,1},{0,1},{2,1},{3,1} | 3>4>1 | 3>4>1 |
| 剩余运行时间全为0 | {0,1},{0,1},{0,1},{0,1} | NULL | NULL |

#### b. 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},{2,5},{3,2},{4,1} | 4>3>1>2 | 4>3>1>2 |
| 随机2 | {5,1},{3,3},{4,5},{2,2} | 1>4>2>3 | 1>4>2>3 |
| 相同优先级与剩余运行时间，以遍历顺序执行 | {1,1},{1,1},{1,1},{1,1} | 4>3>2>1 | 4>3>2>1 |
| 存在相同优先级，按照SJF执行 | {5,1},{3,3},{4,1},{2,1} | 4>3>1>2 | 4>3>1>2 |
| 存在剩余运行时间为0 | {5,2},{3,2},{0,1},{2,1} | 4>2>1 | 4>2>1 |
| 剩余运行时间全为0 | {0,1},{0,2},{0,3},{0,1} | NULL | NULL |

## 4. 讨论与心得

本次实验比较困难的一个点是`switch_to`的实现，需要对结构体有比较扎实的理解；由于文件较多，对程序运行流程的整体把握显得更为重要；此外，对gdb对使用还需要进一步熟悉。其余根据实验指导按部就班实现并不困难，许多要点已经给出源码或者在头文件中给出提示。

## 5. Code

## lab3/arch/riscv

### main.c

```c
#include "test.h"
extern test();
extern init();

int main(){
  puts("ZJU OS LAB 3      Student:3190102362 王奕天\n");
    init();
    test();
    while(1){
    }
    return 0;
}
```

## init.c

时钟中断相关初始化，并且调用`task_init()` (sched.c) 进行进程初始化

```c
#include "riscv.h"

extern void clock_init(void);

void intr_enable(void) {
    //设置sstatus[sie]=1,打开s模式的中断开关
    //your code
    __asm__ volatile (
    "li t0, 0x0000000000000002\n"
    "csrs sstatus, t0\n"
    :
    :
  );
}

void intr_disable(void) {
    //设置sstatus[sie]=0,关闭s模式的中断开关
    //your code
    __asm__ volatile (
    "li t0, 0xfffffffffffffffd\n"
    "csrc sstatus, t0\n"
    :
    :
  );
 }

void idt_init(void) {
    extern void trap_s(void);
    //向stvec寄存器中写入中断处理后跳转函数的地址
    //your code
    unsigned long long int addr = &trap_s;
```

```
    __asm__ volatile (
    "mv t0, %[addr]\n"
    "csrw stvec, t0\n"
    :
    : [addr] "r" (addr)
  );

}

void init(void) {
    idt_init();
    intr_enable();
    clock_init();
    task_init();
}
```

## sbi.c

```
#include "defs.h"
#include "riscv.h"

uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t arg2) {
    uint64_t ret_val;
    __asm__ volatile (
          //Your code
    "mv a7, %[sbi_type]\n"
  "mv a0, %[arg0]\n"
  "mv a1, %[arg1]\n"
  "mv a2, %[arg2]\n"
  "ecall\n"
  "mv %[ret_val], a0\n"
  : [ret_val] "=r" (ret_val)
  : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2)
  : "memory"

   );
    return ret_val;
}


void trigger_time_interrupt(unsigned long long stime_value) {
    sbi_call(0, stime_value, 0, 0);
}
```

### trap.c

```
#include "defs.h"

extern main(),puts(),put_num(),ticks,do_timer();

extern void clock_set_next_event(void);

void handler_s(uint64_t cause, uint64_t epc )
{
  // interrupt
  if ( cause & 0x1000000000000000 == 0x1000000000000000 ) {
    // supervisor timer interrupt
    uint64_t cause1 = cause & 0x7fffffffffffffff;
    if ( cause1 == 5 ) {
      //run do_timer() per 10 ticks, which means 0.1s.
            if(ticks%10==0){
              do_timer();
            }
            clock_set_next_event();
    }
  }
  return;
}
```

## sched.c

- `void task_init(void)`: 进行手动内存分配与进程初始化
- `void do_timer(void)`: 减少当前进程运行时间并控制是否进行调度
- `void schedule(void)`: SJF/PRIORITY调度算法实现
- `void switch_to(struct task_struct* next)`: 调用`__switch_to()`(entry.s)实现调度

```
#include "stdio.h"
#include "sched.h"

#define Kernel_Page             0x80210000
#define LOW_MEMORY              0x80211000
#define PAGE_SIZE               4096UL

struct task_struct* current;
struct task_struct* task[NR_TASKS];


extern task_test_done;
extern void __init_epc(void);
```

```c
//If next==current,do nothing; else update current and call __switch_to.
void switch_to(struct task_struct* next) {
    /*your code*/
    if (current->pid == next->pid) return;
    else{
      struct task_struct* n = current;
      current = next;
      __switch_to(n, next);
    }
}


int task_init_done = 0;
//initialize tasks, set member variables
void task_init(void) {
    puts("task init...\n");

    //initialize task[0]
    current = (struct task_struct*)Kernel_Page;
    current->state = TASK_RUNNING;
    current->counter = 1;
    current->priority = 5;
    current->blocked = 0;
    current->pid = 0;
    task[0] = current;
    task[0]->thread.sp = (unsigned long long) task[0] + TASK_SIZE;
    asm("la t0, __init_epc");
    asm("sd t0, %0"::"m"(task[0]->thread.ra));

    //set other 4 tasks
    for (int i = 1; i <= LAB_TEST_NUM; ++i) {

        /*your code*/
        task[i] = (struct task_struct*)(Kernel_Page + i*TASK_SIZE);
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = 5;
        task[i]->blocked = 0;
        task[i]->pid = i;
        task[i]->thread.sp = (unsigned long long) task[i] + TASK_SIZE;

        asm("la t0, __init_epc");
        asm("sd t0, %0"::"m"(task[i]->thread.ra));

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

```
#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;
    if (task_test_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.
    /*your code*/
    current->counter--;
    if (current->counter <= 0) schedule();
}

//Select the next task to run. If all tasks are done(counter=0), set task0's counter to
1 and it would
//assign new test case.
void schedule(void) {
    unsigned char next;
    /*your code*/
    _Bool allzero = 1;
    next = LAB_TEST_NUM;
    for(int i = LAB_TEST_NUM; i > 0; i--){
        if(task[i]->state == TASK_RUNNING){
            if(task[i]->counter > 0) allzero = 0;
            if((task[next]->counter <= 0 && task[i]->counter > 0) || (task[i]->counter
> 0 && task[i]->counter < task[next]->counter))
                next = i;
        }
    }

    if(allzero){
        printf("Current Setting is down, use task0 to initialize a new setting.\n");
        task[0]->counter = 1;
        next = 0;
    }

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

```
}

#endif

#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;
    if (task_test_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.
    /*your code*/
    current->counter --;
    schedule();
}

//Select the next task to run. If all tasks are done(counter=0), set task0's counter to
1 and it would
//assign new test case.
void schedule(void) {
    unsigned char next;
    /*your code*/
    _Bool allzero = 1;
    next = LAB_TEST_NUM;
    for(int i = LAB_TEST_NUM; i > 0; i--){
        if(task[i]->state == TASK_RUNNING){
            if(task[i]->counter > 0) allzero = 0;
            if(task[i]->priority < task[next]->priority && task[i]->counter > 0) next =
i;
            else if((task[next]->counter <= 0 && task[i]->counter > 0)|| (task[i]-
>priority == task[next]->priority && task[i]->counter < task[next]->counter && task[i]-
>counter > 0))
                next = i;
        }
    }

    if(allzero){
        printf("Current Setting is down, use task0 to initialize a new setting.\n");
        task[0]->counter = 1;
        next = 0;
    }

    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);
    }

    switch_to(task[next]);

}

#endif
```

## entry.s

- `trap_s`: 调用中断处理函数`handler_s(scause)`以及中断前后的保存与加载
- `__switch_to`: 实现调度
- `__init_epc`: 为初次调度的进程提供返回函数，将寄存器信息压栈

```
.section .text.entry

.extern test
.global trap_s
.extern handler_s
.equ reg_size, 0x8
.align 2

trap_s:
  #save caller-saved registers and spec
  addi sp, sp, -256
    sd x1, 0(sp)
    sd x2, 8(sp)
    sd x3, 16(sp)
    sd x4, 24(sp)
    sd x5, 32(sp)
    sd x6, 40(sp)
    sd x7, 48(sp)
    sd x8, 56(sp)
    sd x9, 64(sp)
    sd x10, 72(sp)
    sd x11, 80(sp)
    sd x12, 88(sp)
    sd x13, 96(sp)
    sd x14, 104(sp)
    sd x15, 112(sp)
    sd x16, 120(sp)
  sd x17, 128(sp)
  sd x18, 136(sp)
  sd x19, 144(sp)
  sd x20, 152(sp)
  sd x21, 160(sp)
```

```
    sd x22, 168(sp)
    sd x23, 176(sp)
    sd x24, 184(sp)
    sd x25, 192(sp)
    sd x26, 200(sp)
    sd x27, 208(sp)
    sd x28, 216(sp)
    sd x29, 224(sp)
    sd x30, 232(sp)
    sd x31, 240(sp)
      csrr t0, sepc
      sd t0, 248(sp)

    # call handler_s(scause)
    csrr a0, scause
      call handler_s

    # load sepc and callee-saved registers
    ld t0, 248(sp)
      csrw sepc, t0
      ld x1, 0(sp)
      ld x2, 8(sp)
      ld x3, 16(sp)
      ld x4, 24(sp)
      ld x5, 32(sp)
      ld x6, 40(sp)
      ld x7, 48(sp)
      ld x8, 56(sp)
      ld x9, 64(sp)
      ld x10, 72(sp)
      ld x11, 80(sp)
      ld x12, 88(sp)
      ld x13, 96(sp)
      ld x14, 104(sp)
      ld x15, 112(sp)
      ld x16, 120(sp)
    ld x17, 128(sp)
    ld x18, 136(sp)
    ld x19, 144(sp)
    ld x20, 152(sp)
    ld x21, 160(sp)
    ld x22, 168(sp)
    ld x23, 176(sp)
    ld x24, 184(sp)
    ld x25, 192(sp)
    ld x26, 200(sp)
    ld x27, 208(sp)
    ld x28, 216(sp)
    ld x29, 224(sp)
```

```
  ld x30, 232(sp)
  ld x31, 240(sp)
    addi sp, sp, 256

  sret


.globl __switch_to
__switch_to:
  li    a4,  40
  add   a3, a0, a4
  add   a4, a1, a4
  #Save context into prev->thread
  #your code
  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)

  #Restore context from next->thread
  #your code
  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 ra
  ret
```

```
.globl __init_epc
__init_epc:
  la  t0, test
  csrw sepc, t0
  sret
```

# lab3/include

## sched.h

```
#ifndef _SCHED_H
#define _SCHED_H

#define TASK_SIZE   (4096)
#define THREAD_OFFSET  (5 * 0x08)

#ifndef __ASSEMBLER__

/* task的最大数量 */
#define NR_TASKS    64

#define FIRST_TASK  (task[0])
#define LAST_TASK   (task[NR_TASKS-1])

/* 定义task的状态，Lab3中task只需要一种状态。*/
#define TASK_RUNNING                0
// #define TASK_INTERRUPTIBLE       1
// #define TASK_UNINTERRUPTIBLE     2
// #define TASK_ZOMBIE              3
// #define TASK_STOPPED             4

#define PREEMPT_ENABLE  0
#define PREEMPT_DISABLE 1

/* Lab3中进程的数量以及每个进程初始的时间片 */
#define LAB_TEST_NUM        4
#define LAB_TEST_COUNTER    5

/* 当前进程 */
extern struct task_struct *current;

/* 进程指针数组 */
extern struct task_struct * task[NR_TASKS];

/* 进程状态段数据结构 */
struct thread_struct {
    unsigned long long ra;
    unsigned long long sp;
```

```
    unsigned long long s0;
    unsigned long long s1;
    unsigned long long s2;
    unsigned long long s3;
    unsigned long long s4;
    unsigned long long s5;
    unsigned long long s6;
    unsigned long long s7;
    unsigned long long s8;
    unsigned long long s9;
    unsigned long long s10;
    unsigned long long s11;
};

/* 进程数据结构 */
struct task_struct {
     long state;    // 进程状态 Lab3中进程初始化时置为TASK_RUNNING
     long counter;  // 运行剩余时间
     long priority; // 运行优先级 1最高 5最低
     long blocked;
     long pid;      // 进程标识符
    // Above Size Cost: 40 bytes

    struct thread_struct thread; // 该进程状态段
};

/* 进程初始化 创建四个dead_loop进程 */
void task_init(void);

/* 在时钟中断处理中被调用 */
void do_timer(void);

/* 调度程序 */
void schedule(void);

/* 切换当前任务current到下一个任务next */
void switch_to(struct task_struct* next);

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

/* 死循环 */
void dead_loop(void);

#endif

#endif
```