P2的整体逻辑

曾有同学问过，P1这个实验和写一个普通的C程序感觉区别不是很大。“操作系统”的概念体现在哪里？实际上P1就是一个在裸板上执行的程序，更多的是教大家在没有操作系统的环境下，应当怎样编写一个程序。对于操作系统本身的概念涉及不多。

与P1不同，P2是一个虽然简单，但是具备了最最基础的功能的内核。操作系统是负责管理硬件并提供系统服务的。因此，在P2中，内核实际上扮演了一个“提供系统服务”的角色。而我们的测试程序将使用内核提供的系统服务来完成各种任务。

测试程序更接近于我们平时在操作系统上执行的C程序，它会使用C库中的一些函数（以及一些系统服务）。为了更接近真实系统，我们做了一个超小型的C库(tiny_libc)。测试程序会使用libc中的函数执行一些任务。libc中的函数有的是简单的字符串操作等不依赖OS的功能，有的则依赖于OS提供的服务。依赖于OS的服务最终会调用到`sys/syscall.h`中的函数，其定义位于`tiny_libc/syscall.c`中。这些函数负责真正触发内核的相关功能，请求相应的系统服务。任务一和任务二中，可以直接调用内核的相关函数执行系统服务。在后面的任务中，建立了完整的例外处理机制后，需要通过触发软中断陷入内核态请求系统服务。

以printf为例，在任务一中，用户程序调用printf。printf是被定义在`tiny_libc/printf.c`中的。之后，可以调用`tiny_libc/syscall.c`中的`sys_write`函数。再由`sys_write`函数调用内核中的`port_write`服务直接写串口，从而实现输出。也可以在`printf.c`中，直接调用内核提供的`printk`函数来使用内核提供的功能。在后面任务三有了例外处理等保护机制后，触发内核服务的方式需要由对`port_write`或`printk`内核函数的直接调用的方式，改成用`ecall`陷入内核态，再通过内核的例外处理机制执行相应的内核功能的方式。

FAQ

Part1:

- 代码中的ready_queue等队列如何初始化？如何操作？
- 在任务一中，测试程序中的sys_yield()、sys_move_cursor()等是否要改成do_scheduler()、vt100_move_cursor()等内核函数?
- switch_to()和SAVE_CONTEXT/RESTORE_CONTEXT是什么关系？
- printk函数崩溃是什么原因？
- enable_interrupt/disable_interrupt实现存在错误？
- enable_preempt/disable_preempt是做什么用的？
- pid0是做什么用的？
- 如何查找一个函数（或变量名等符号）的位置？
- 实验2的目录结构是什么样的？每个文件大概是做什么用的？
- GP寄存器如何初始化？
- cursor是做什么用的？如何初始化？
- 执行qemu以后输出大量truly illegal insn怎么办？
- kernel.c在哪里？为什么找不到这个文件？
- test/test.h中为什么声明了两个num_lock_tasks?
- 锁要如何设计？有什么需要注意的？
- printf无法打印？
- qemu显示hardware error错误？
- trap.S的setup_exception需要修改？

Part2

- 中断的机制流程是什么样的？

- sleep应该如何实现?
- 调用ecall后未能跳入到stvec所在地址? QEMU显示n=0x16?
- 如何进入U态?
- sbi_set_timer的作用是?
- kernel/syscall/syscall.c下面的syscall全局变量的用途是什么?
- 系统调用对应的例外是什么?
- time_base的含义是什么?
- csr.h中的SCAUSE_IRQ_FLAG会报警告?
- gdb单步跟踪ecall/sret好像什么都没发生?
- 任务三的时候几个测试任务好像没法切换?
- irq.h中的TIMER_INTERVAL宏是做什么用的?
- 进行任务切换需要保存几次上下文? 都需要保存哪些寄存器?
- SIE寄存器和SSTATUS寄存器的SIE位的区别是什么? 为什么要有这么两个功能相近的设置，只保留一个可不可以?
- screen_write/screen_reflush等应该怎么用?
- SAVE_CONTEXT中的_restore_kernel_tpsp等代码有什么作用?
- stval是做什么用的?

代码中的ready_queue等队列如何初始化? 如何操作?

代码中所有对于队列的操作都需要同学们根据自己的实际情况自行实现。start code中和队列相关的代码都需要大家根据自己的具体实现方式自行调整。

早期版本的实验中，我们仿照Linux的链表实现提供了一个list.h。但是从往年的实际情况看，Linux中链表的设计更侧重于追求通用性，用到了一些和C语言内存布局方面相关的技巧，导致部分同学理解起来比较困难。我们的内核比较简单，需要的队列的类型很少，即使不做很通用的设计，也可以正常完成（虽然有可能有部分代码上的冗余）。因而，今年不再提供链表的具体实现，请大家根据自己的需求自行设计。如果需要，框架中涉及到链表类型的函数声明等都可以自行修改。

简单起见建议大家就采用自己在数据结构教科书上学到的标准的链表实现即可。**学有余力**可以稍微参考一下Linux或其他操作系统中链表的实现方式。

Linux的链表的设计更多的是追求的代码可重用。传统的教科书里定义链表时，都是把数据和指针域放在一个结构体里，然后针对这个结构体写各种插入删除等代码的。但如果有多种不同数据类型都要串成链表，代码就不好复用。例如:

```
// 一般的链表,同样的东西需要反复定义
// 特别是链表的增加/删除等操作的函数也需要跟着重新定义
struct IListNode {
    int val;
    struct IListNode* next;
};
void IListAdd(struct IListNode* head, struct IListNode *new_node); // ...
struct DListNode {
    double val;
    struct DListNode* next;
};
void DListAdd(struct DListNode* head, struct DListNode *new_node); // ...
struct SListNode {
    struct Something val;
    struct SListNode* next;
};
void SListAdd(struct SListNode* head, struct SListNode *new_node); // ...
```

对于这种问题，在一些新的编程语言中可以用范型或者面向对象的类的继承等方式解决。例如C++就可以写`list<int>`、`list<double>`等。但C语言本身没提供这类的语法。所以Linux中，把链表的指针域和相关的操作抽了出来，任何结构体想串成链表只需要嵌入相应指针域就行。

```
// Linux 里的指针域及其操作的定义
struct list_head {
    struct list_head *next, *prev;
};
static inline void list_add(struct list_head *new, struct list_head *head);
static inline void list_add_tail(struct list_head *new, struct list_head *head);
static inline void list_del_init(struct list_head *entry);
// 使用的时候把指针域嵌入到数据中
struct val {
    int val;
    struct list_head node;
};
```

感兴趣的同学请自行查阅参考资料中的相关链接。

在任务一中，测试程序中的sys_yield()、sys_move_cursor()等是否要改成do_scheduler()、vt100_move_cursor()等内核函数?

不要将测试程序中的函数直接改成相应的内核函数！

在任务一和任务二中，sys_yield()、sys_move_cursor()等函数确实就是直接调用do_scheduler()、vt100_move_cursor()等内核函数实现的。但是我们仍然希望在形式上，测试程序中是在调用tiny_libc库中的sys_yield()、sys_move_cursor()等函数。这种设计是为了强调一个概念：do_scheduler()、vt100_move_cursor()等内核函数都是内核中的系统服务函数，用户程序不直接调用内核函数（否则就感觉测试程序好像是系统的一部分似的）。虽然我们的内核和测试程序由于种种限制编译在了一起，但本质上，内核是内核，测试程序就和大家平时写的普通的程序差不多，它俩逻辑上是分开的。测试程序是通过系统调用请求内核服务的。

任务一和任务二中由于还没有例外处理机制，在实现系统调用时只能通过普通的函数调用实现。在后面有了例外处理机制后，再将sys_yield()等函数改成用ecall指令实现的方式，实现完整的带保护的系统调用方式。这一过程对于测试程序是透明的，测试程序只知道自己是通过sys_yield()等tiny_libc中的函数去请求系统服务，具体请求系统服务的实现方式它无需关心。这样一层封装可以更明确地强调，从用户程序的角度看，用户程序和内核是分开的。这种设计是为了让大家对于操作系统内核的概念有一个更直观的感觉。

switch_to()和SAVE_CONTEXT/RESTORE_CONTEXT是什么关系?

`switch_to`函数是在内核中主动切换进程时调用的。`SAVE_CONTEXT`和`RESTORE_CONTEXT`是后面例外处理的时候保存例外现场使用的。这两种情况有明显的区别。

`switch_to`是主动切换的，软件明确的知道在调用这个函数后就就进入进程切换了。且因为是主动调用switch_to，所以遵循函数调用的相关约定。所有需要由调用者保存的寄存器已经在函数调用前都保存在栈上了。`switch_to`中只需要保存所有该由被调用者保存的寄存器即可。

`SAVE_CONTEXT`和`RESTORE_CONTEXT`是后面例外处理的时候保存例外现场使用的。例外往往都是突然发生的。例如外部的时钟触发了中断，或者某条指令发生了异常等，用户程序往往不可预知例外是在何处发生的。因此，在例外发生时，需要假定目前所有的通用寄存器都在被使用（因为是突然发生的，可能在任意一条指令处触发，所以没法确定此刻哪些寄存器是有用的，哪些是没用的，和函数调用时的情况

不同）。这也就是为什么`SAVE_CONTEXT`和`RESTORE_CONTEXT`涉及所有的寄存器的原因。

printk函数崩溃是什么原因?

printk函数会试图把当前的光标保存到current_running，如果current_running为NULL的话会出错。这里根据你自己的设计可以选择删除那段保存光标的代码或者让current_running始终非空。

```
/* libs/printk.c: vprintk() */
disable_preempt();
port_write(buff);
for (int i = 0; i < ret; ++i) {
    if (buff[i] == '\n') {
        current_running->cursor_y++;
    } else if (buff[i] == '\r') {
        current_running->cursor_x = 1;
    } else {
        current_running->cursor_x++;
    }
}
enable_preempt();
```

enable_interrupt/disable_interrupt实现存在错误?

这里确实存在问题，start code中的代码是：

```
ENTRY(enable_interrupt)
  li t0, SR_SIE
  csrs CSR_SSTATUS, t0
  jr ra
ENDPROC(enable_interrupt)

ENTRY(disable_interrupt)
  li t0, SR_SIE
  csrs CSR_SSTATUS, t0
  jr ra
ENDPROC(disable_interrupt)
```

应该改成：

```
ENTRY(enable_interrupt)
  li t0, SR_SIE
  csrs CSR_SSTATUS, t0
  jr ra
ENDPROC(enable_interrupt)

ENTRY(disable_interrupt)
  li t0, SR_SIE
  csrc CSR_SSTATUS, t0
  jr ra
ENDPROC(disable_interrupt)
```

enable_preempt/disable_preempt是做什么用的?

在一些内核代码里能看到`enable_preempt`和`disable_preempt`这一对函数，这对函数原本的功能是允许发生抢占。但我们的实验中为了简化，不需要支持内核抢占。因此，所有的这对函数都可以去除。因为在内核态下是不会发生时钟中断导致抢占的。

建议可以把这两个函数直接改成：

```
ENTRY(enable_preempt)
  jr ra
ENDPROC(enable_preempt)

ENTRY(disable_preempt)
  jr ra
ENDPROC(disable_preempt)
```

相当于不执行任何功能，或者直接删掉就完了。以前有极少数同学会出现不删这两个函数就有奇怪的错误，删掉就能正常工作的情况。

pid0是做什么用的?

在start code中给了一个`pid0_stack`、`pid0_pcb`等的定义。这些定义可用可不用，与每个人自己的设计有关。代码框架中的pid0相关的变量代表的是最开始的内核。最开始刚启动的时候跑的是内核的代码。之后要切换到第一个任务中。但`switch_to`假定是从一个任务切换到另一个任务。所以对于内核代码切换到第一个任务的这一过程，不完全符合`switch_to`的假设。

`switch_to`的主要假设是`a0`和`a1`两个寄存器传入了两个pcb。pcb中记录了上次进程切换时kernel stack的位置。对于内核而言，它没有相应的pcb。为了符合这个假设，所以我们做了一个pid0的pcb。并且最开始将current_running设置为pid0_pcb。这样就可以比较自然地从代表内核的pid0_pcb切换到第一个任务的pcb。后面内核这个进程可以用作一个idle进程。当没有任何任务可以调度的时候就跑内核的死循环代码（循环调用`do_scheduler`）。

也可以对switch_to做一些小改动来完成这个。`switch_to`需要传入pcb的原因单纯是需要保存/恢复kernel_stack的位置。所以其实也可以选择把`switch_to`改成传两个指针进去。一个用于保存上一个任务的kernel_stack，一个则是指向了下一个任务的kernel_stack的值。这样，对于内核切换第一个任务的过程，调用switch_to的时候用于保存上一个任务的kernel_stack的指针随便指向一个合法的位置就行。当然，也有更为朴素的方式，就是直接判断是否是内核切第一个任务，并特殊处理即可。

如何查找一个函数（或变量名等符号）的位置?

最简单的方式是使用grep命令查找。方式是先切换到OS源代码的根目录下，然后在命令行输入grep命令进行查找：

```
# name部分替换为要搜索的符号。
grep -r “name”
```

通过这种方式可以快速找到符号所在的文件。例如，如果想找`list_node_t`的所有出现的位置，可以：

```
$ grep -r "list_node_t"
include/os/sched.h:    list_node_t list;
include/os/sched.h:void do_block(list_node_t *, list_head *queue);
include/os/sched.h:void do_unblock(list_node_t *);
include/os/list.h:} list_node_t;
include/os/list.h:typedef list_node_t list_head;
kernel/sched/sched.c:void do_block(list_node_t *pcb_node, list_head *queue)
kernel/sched/sched.c:void do_unblock(list_node_t *pcb_node)
```

实验2的目录结构是什么样的？每个文件大概是做什么用的？

Project2的目录结构及各文件作用如下图所示：

```
$ tree Project2-SimpleKernel
Project2-SimpleKernel
├── arch
│   └── riscv
│       ├── boot
│       │   └── bootblock.S # 引导块源代码，需要使用p1的支持加载大核的版本的代码
│       ├── include
│       │   ├── asm # 汇编文件中使用的头文件
│       │   │   ├── regs.h # 定义了一些寄存器在栈/结构体中的偏移
│       │   │   ├── sbiasm.h # 汇编调用SBI的宏的定义
│       │   │   └── sbidef.h # SBI调用号宏定义
│       │   ├── asm.h # 用于帮助定义汇编代码中的函数/过程的宏
│       │   ├── atomic.h # 原子指令的封装
│       │   ├── common.h # 串口输出的相关函数/常量的定义
│       │   ├── csr.h # CSR特权寄存器相关定义
│       │   └── sbi.h # SBI相关功能
│       ├── kernel
│       │   ├── entry.S # 上下文切换/例外处理及中断相关的功能
│       │   ├── head.S # 内核的汇编入口函数，建立C语言执行环境等相关功能
│       │   └── trap.S # 初始化例外处理相关寄存器
│       └── sbi
│           └── common.c # 串口输出的实现
├── drivers
│   ├── screen.c # 屏幕缓冲区相关函数的定义
│   └── screen.h # 屏幕缓冲区相关函数的实现
├── include
│   ├── assert.h # 断言功能的实现
│   ├── os
│   │   ├── irq.h # 例外处理相关函数和全局变量声明
│   │   ├── list.h # 链表相关函数声明
│   │   ├── lock.h # 锁相关函数和全局变量声明
│   │   ├── mm.h # 内存分配相关函数和全局变量声明
│   │   ├── sched.h # 任务管理/调度相关函数和全局变量声明
│   │   ├── string.h # 字符串处理相关函数和全局变量声明
│   │   ├── sync.h # 同步相关函数和全局变量声明
│   │   ├── syscall.h # 系统调用相关函数和全局变量声明
│   │   ├── syscall_number.h # 系统调用号宏定义
│   │   └── time.h # 时间相关函数声明
│   ├── stdarg.h # 变长参数相关的宏定义
│   ├── stdio.h # 内核中使用的stdio.h
│   ├── sys
│   │   └── syscall.h # （用户使用的）系统调用相关声明
│   └── type.h # 内核中使用的各种类型的定义
├── init
│   └── main.c # 内核C语言入口和各类初始化函数定义
├── kernel
│   ├── irq
│   │   └── irq.c # 例外处理相关函数和全局变量的定义
│   ├── locking
│   │   └── lock.c # 锁相关函数和全局变量的定义
│   ├── mm
```

```
|   |   └── mm.c # 内存分配相关函数和全局变量定义
|   ├── sched
|   |   ├── sched.c # 任务管理/调度相关函数和全局变量的定义
|   |   └── time.c # 时间相关函数的定义
|   └── syscall
|       └── syscall.c # 内核中系统调用相关处理函数的定义
├── libs
|   ├── printk.c # 内核printk实现
|   └── string.c # 内核中的字符串维护相关函数的实现
├── Makefile # Makefile文件
├── riscv.lds # linker script文件
├── test # 测试程序
|   ├── test.c # 测试程序入口/类型的定义
|   ├── test.h
|   └── test_project2 # 各个测试程序的相关声明和实现
|       ├── test2.h
|       ├── test_lock.c
|       ├── test_scheduler.c
|       ├── test_sleep.c
|       └── test_timer.c
├── tiny_libc # 迷你C库，给用户程序(测试程序)用的
|   ├── include
|   |   ├── assert.h
|   |   ├── mthread.h
|   |   ├── stdarg.h
|   |   ├── stdatomic.h
|   |   ├── stdbool.h
|   |   ├── stdint.h
|   |   ├── stdio.h
|   |   ├── string.h
|   |   └── time.h
|   ├── mthread.c # 锁相关的实现
|   ├── printf.c # printf相关的实现
|   ├── string.c
|   ├── syscall.c # 调用系统服务的封装
|   ├── syscall.S
|   └── time.c
└── tools
    └── createimage.c
```

GP寄存器如何初始化?

编译器在索引一些全局变量的时候，会用GP寄存器做相对寻址。因此，需要将该寄存器初始化为编译器计算出的global pointer的地址。我们的start code中的head.S已经给出了初始化代码。目前任务一的switch_to中，test程序是和内核编译在一起的，所以，test程序和kernel用的gp是同一个。因此，在head.S中初始化好gp后，在P2的任务一二中可以不用再初始化。只要保存恢复寄存器的时候别把gp的值弄错了就行。

此外，同P1一样，也可以采用阻止编译器生成和gp相关的指令的方法解决所有GP相关的问题。方法是将riscv.lds中global pointer相关的语句删除：

```
__global_pointer$ = MIN(__SDATA_BEGIN__ + 0x800,
                        MAX(__DATA_BEGIN__ + 0x800, __BSS_END__ - 0x800));
```

删除后，编译器就不会再生成使用gp跳转的指令，也不会再有`__global_pointer$`这个符号。

cursor是做什么用的？如何初始化？

cursor就相当于是每个进程在屏幕上的光标位置。简单理解可以把屏幕想象成一台打印机，打印字符就像打印机在顺序地往后打印东西。如果多个进程轮流都把自己的字符往上打，那么打印出来的东西肯定是混乱的（每个进程都输出了一部分，交错在一起）。所以需要管理记录每个进程打印的位置，并且在进程切换的时候把打印的光标移动回它原来的位置。

我们的测试程序开头都调用了移动光标的函数，使得每个程序从不同的行开始打印。这样就不会有遮蔽的情况发生了。cursor初始化时可以都初始化成1。因为测试程序一开始会调用光标移动函数指定位置，所以理论上不初始化也不会出错，就是这样设计系统比较丑。按理说还是初始化一下比较好。

执行qemu以后输出大量truly_illegal_insn怎么办？

qemu输出`truly_illegal_insn`是因为执行到了非法指令。大概率是处理器的PC跑到了奇怪的位置，处理器读取到的指令是非法指令。在实验二中，可能导致非法指令的情况非常多，只能自行用gdb一点一点跟踪。

比较常见的可能是：

1. 初始化任务的时候初始化入口地址有问题（可以用gdb打印变量确认）
2. 任务切换的时候，没能跳到正确的地址（可以把断点设置在`switch_to`观察保存恢复现场的过程，以及最后跳转到了哪里）

kernel.c在哪里？为什么找不到这个文件？

这里是任务书的笔误，应该是main.c。

test/test.h中为什么声明了两个num_lock_tasks?

这个是start code中的一个笔误，后面一个应该是`num_lock2_tasks`，和test.c中的定义保持一致。不过因为这俩变量是一个值，所以不管它问题也不是很大。

锁要如何设计？有什么需要注意的？

锁在设计的时候要注意区分用户程序和内核。用户不应当能够直接得到内核的关键信息。用户程序通过C库/系统调用来请求内核的服务。对于我们的实验而言，可以简单地理解为，test目录下的程序，通过tiny_libc目录下的函数，请求内核服务。

在设计给用户使用的锁的接口的时候存在一个问题，要保证用户无法直接拿到内核的核心信息。例如，内核中实际管理的锁的结构一般是这样的：

```
/* 内核中一种可能的锁的实现方案 */
typedef struct mutex_lock
{
    volatile lock_status_t status;
    list_head block_queue; // 等待该锁的线程的阻塞队列
} mutex_lock_t;
void do_mutex_lock_init(mutex_lock_t *lock);
void do_mutex_lock_acquire(mutex_lock_t *lock);
void do_mutex_lock_release(mutex_lock_t *lock);
```

但这里定义的这个`mutex_lock_t`只能在内核内部使用，因为其包含了`block_queue`这一敏感信息。用户程序如果拿到了这个信息，就能顺着查找到其他进程的pcb结构甚至保存的上下文的位置等等。因此，不能直接将这一组接口简单封装直接给用户使用。

这里建议大家可以参考UNIX的信号量的相关接口的设计方式，给用户提供这样一组接口：

```c
/* mutex_get的参数key是一个特殊的键值，给到内核后，内核会返回一个id。
 * 这个id代表一个内核管理的锁对象。同一个key值会返回代表同一个锁对象的id。
 * 后续的操作都用代表锁对象的id来实现。
 */
int mutex_get(int key);
int mutex_lock(int handle);
int mutex_unlock(int handle);
/* 如果想少定义接口函数，也可以合并一下后两个函数 */
int mutex_get(int key);
int mutex_op(int handle, int op); // op = 0就执行lock， op = 1就执行unlock
```

封装成用户使用的锁结构的时候可以用变量的地址做key：

```c
typedef int mthread_mutex_t;
int mthrea_mutex_init(void *handle)
{
    int *id = (int*)handle;
    *id = mutex_get((int)handle); // 使用handle的地址作为KEY
}
int mthread_mutex_lock(void* handle) {return mutex_lock(*(int*)handle);}
```

当然，也有其他可行的设计，这里都只是简单示意一下。核心的思路就是内核返回内核锁对象的id给用户，而不是直接把锁对象的地址给用户。从而避免用户通过地址直接访问到内核中的数据结构。

printf无法打印?

start code中的printf是个半成品，按照任务书的要求，是需要封装一下printk或者port_write来实现打印的。

```c
/* tiny_libc/printf.c */
int vprintf(const char *fmt, va_list _va)
{
    va_list va;
    va_copy(va, _va);

    int ret;
    char buff[256];

    ret = mini_vsnprintf(buff, 256, fmt, va);

    buff[ret] = '\0';

    //call kernel print function or ecall
    //sys_write(buff);

    return ret;
}
```

可以看到`sys_write`被注释掉了。这里可以调用`tiny_libc/syscall.c`中的`sys_write`函数。再由`sys_write`函数调用内核中的`port_write`服务直接写串口，从而实现输出。也可以在`printf.c`中，直接调用内核提供的`printk`函数来使用内核提供的功能。在后面任务三有了例外处理等保护机制后，触发内核服务的方式需要由对`port_write`或`printk`内核函数的直接调用的方式，改成用`ecall`陷入内核态，再通过内核的例外处理机制执行相应的内核功能的方式。

qemu显示hardware error错误?

特征：QEMU显示“qemu: hardware error: sifive_test_write: write addr=.....”。

这个一般是访问到了0x0开头的4K空间的地址内可能会出现这种报错。RISC-V板卡一般这个位置是一个Debug模块。然后因为乱写debug模块对应的地址导致debug模块发生硬件错误。addr后面表示的是你尝试写入的位置。遇到这个问题多半是因为指针指飞了，访问了野指针/空指针之类的错误。需要跟着gdb慢慢调。一种可能的调试思路是想办法找到到底是哪行C代码触发了这个错误。为了找到对应的C代码，首先要去找发生错误的汇编代码。QEMU报错时会给出的pc和mepc地址。pc就是当前正在执行的指令的地址。mepc是上一次发生例外时的指令的地址。理论上pc地址是直接导致错误的地址。但pc有可能会是bbl/uboot这种machine态的代码。一般而言，如果pc是bbl/uboot中的代码，那么有可能是你调用sbi的时候参数有什么问题。mepc存的是调用sbi的指令的地址。

综上，我们可以优先看看pc这个地址对应的指令是啥，如果找不到，再试试mepc这个地址对应的指令。通过指令的地址找到相应指令，再对应回C代码里面，找到发生问题的C代码。

我们start code给的Makefile会自动生成一个kernel.txt，里面是kernel的反汇编代码：

```
asm:
	${OBJDUMP} -d main > kernel.txt
```

打开以后大致如下：

```
Disassembly of section .text:

ffffffc050300000 <_start>:
ffffffc050300000:	10401073                csrw    sie,zero
ffffffc050300004:	14401073                csrw    sip,zero
ffffffc050300008:	6299                    lui     t0,0x6
ffffffc05030000a:	1002b073                csrc    sstatus,t0
ffffffc05030000e:	00023697                auipc   a3,0x23
```

根据看到的pc/mepc查找一下相应的指令在哪个函数里面，再对着对应的C代码对照着看看C代码出现在哪个函数中。这里还有个额外的技巧，我们编译的时候如果开启了调试信息(gcc的`-g`或`-ggdb`选项)，那么其实objdump是有能力显示汇编对应的源代码的（不一定完全准确，因为编译优化等原因分属于不同C语句的汇编指令会混在一起，所以有时候显示汇编对应的源代码是乱的）。命令为：

```
# 想在命令行下直接看的话可以接个less
riscv64-unknown-linux-gnu-objdump --source -d main | less
# 或者也可以像makefile里那样导入到文件中
riscv64-unknown-linux-gnu-objdump --source -d main > kernel.txt
```

命令的效果为：

```
static void init_pcb()
{
ffffffc0503002e2:	1101                    addi    sp,sp,-32
ffffffc0503002e4:	ec06                    sd      ra,24(sp)
ffffffc0503002e6:	e822                    sd      s0,16(sp)
ffffffc0503002e8:	1000                    addi    s0,sp,32
    for (int i = 0; i < NUM_MAX_TASK; ++i) {
ffffffc0503002ea:	fe042623                sw      zero,-20(s0)
ffffffc0503002ee:	a01d                    j       ffffffc050300314
<init_pcb+0x32>
        pcb[i].status = TASK_EXITED;
```

```
ffffffc0503002f0:       0010e717                auipc   a4,0x10e
ffffffc0503002f4:       35870713                addi    a4,a4,856 #
ffffffc05040e648 <pcb>
ffffffc0503002f8:       fec42683                lw      a3,-20(s0)
ffffffc0503002fc:       06800793                li      a5,104
ffffffc050300300:       02f687b3                mul     a5,a3,a5
ffffffc050300304:       97ba                    add     a5,a5,a4
ffffffc050300306:       4711                    li      a4,4
ffffffc050300308:       cfb8                    sw      a4,88(a5)
```

这样更容易找到机器指令对应的原始的C语句。如果无法显示源码，可能是源码路径的问题，可以查看一下源码路径然后通过`--prefix`来指定路径的前缀。查看编译出来的调试信息的路径的方法是：

```
$ riscv64-unknown-linux-gnu-readelf --debug-dump=info main
# 以下是在我的机器上给出的输出，能明显地看到编译器产生的文件路径
Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0x2a (32-bit)
   Version:       2
   Abbrev Offset: 0x0
   Pointer Size:  8
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_stmt_list   : 0x0
    <10>   DW_AT_low_pc      : 0xffffffc05030004c
    <18>   DW_AT_high_pc     : 0xffffffc05030006a
    <20>   DW_AT_name        : (indirect string, offset: 0x0):
./arch/riscv/kernel/trap.S
    <24>   DW_AT_comp_dir    : (indirect string, offset: 0x1b): /home/wlm/文
档/ict/os_course/OS研讨课Guiding book/os_exp2
    <28>   DW_AT_producer    : (indirect string, offset: 0x5a): GNU AS 2.32
    <2c>   DW_AT_language    : 32769    (MIPS assembler)
.......
# 看到路径后，如果发现文件位置变了，可以重新编译，也可以用prefix强行指定路径的前缀：
$ riscv64-unknown-linux-gnu-objdump --source --prefix=./ -d main
```

trap.S的setup_exception需要修改?

start code中的`setup_exception`是需要大家在任务三中完成的。但是在任务一二中，start code试图调用这个未实现的函数，可能会造成一些麻烦。建议大家可以在做任务一和任务二时在函数结尾出加一句`jr ra`：

```
ENTRY(setup_exception)
  /* TODO:
   * save exception_handler_entry into STVEC
   * enable global exceptions */

  jr ra
ENDPROC(setup_exception)
```

中断的机制流程是什么样的?

以时钟中断为例，外部时钟产生时钟中断后：

1. sip寄存器中的stip位会被硬件自动设置，代表有一个未被处理的时钟中断。

2. 处理器看到sip被设置后，会检查sie寄存器中的stie位。如果stie位为允许状态，且sstatus中的sie位为允许状态，则产生中断。
3. 中断产生后，处理器切换到内核态，将sstatus寄存器的spie位设置为sstatus寄存器的sie的值，然后自动将sie位置0。
4. 处理器自动跳转到stvec寄存器中存放的地址的位置继续执行。（stvec中存放的地址即所谓的例外处理函数入口地址）
5. 例外处理函数负责保存中断现场，并进入时钟中断的处理函数
6. 时钟中断处理函数调用sbi_set_timer（该函数会将新的tick写入到mtimecmp寄存器，并清空sip的stip位）
7. 例外处理函数恢复现场并调用sret
8. 处理器自动根据sstatus的spp位切换回用户态/内核态，将sstatus寄存器的sie的值设置为sstatus寄存器的spie的值，spie置1，并跳转到sepc继续执行。

sleep应该如何实现？

和sleep相关的函数均需要自行实现，框架中没有提供任何相应的代码。

sleep在任务三和任务四的实现思路会略有差别。sleep的比较常规的实现思路是弄一个sleep的队列，把所有要sleep的进程按照唤醒时间的顺序列在队列里。每隔一段时间检查一下队列头，看看是否需要唤醒。需要唤醒的时候就把可以唤醒的进程添加回就绪队列中。

关键的点在于什么时候去检查sleep队列？最直观的思路是设好时钟，每次时钟中断的时候检查一下sleep队列。但任务三的时候还没有时钟中断。因此无法实现在时钟中断的时候检查队列。这里大家可以采用另一个思路：在每次调度的时候检查一下sleep队列。这就相当于每次准备调度前，把可以调度的进程从sleep队列转移回就绪队列。

调用ecall后未能跳入到stvec所在地址？QEMU显示n=0x16？

有些同学会遇到ecall后未能跳到stvec设置的地址的问题。部分同学还会遇到QEMU显示n=0x16这种情况。这种情况多半是因为调用ecall时没有在用户态。

有些同学细细读代码的时候可能发现了，SBI相关的接口最后也是调用的ecall：

```
#define SBI_CALL(which, arg0, arg1, arg2)                    \
    ({                                                       \
        register uintptr_t a0 asm("a0") = (uintptr_t)(arg0);  \
        register uintptr_t a1 asm("a1") = (uintptr_t)(arg1);  \
        register uintptr_t a2 asm("a2") = (uintptr_t)(arg2);  \
        register uintptr_t a7 asm("a7") = (uintptr_t)(which); \
        asm volatile("ecall"                                 \
                     : "+r"(a0)                              \
                     : "r"(a1), "r"(a2), "r"(a7)             \
                     : "memory");                            \
        a0;                                                  \
    })
```

那么系统是怎么区分哪些是我们自己做的系统调用，哪些是SBI呢？答案是如果当前是S态也就是内核态，那么ecall就会进入到m态，也就是bbl/u-boot中去，执行的就是SBI。如果是u态也就是用户态，那么ecall就会进入到s态，也就是执行的是我们自己的系统调用。

QEMU显示n=0x16的原因是s态执行了ecall进入了u-boot中，u-boot把你的系统调用当作SBI调用处理了。但SBI的调用号没有0x16，它不知道这是个什么调用，所以就打印了个`n=0x16`。

如何进入U态？

想要切换特权级，只能通过例外处理相关机制进行。用sret执行例外返回时，如果sstatus的spp设置正确，那么就会自动返回到User态。

因此，想让测试程序执行在U态的方式是：在进入程序入口时，将程序入口写入到sepc寄存器中，再通过sret指令进入程序执行入口。这样测试程序就执行在User态了。

sbi_set_timer的作用是?

sbi_set_timer()的作用是让处理器在指定的tick数时中断。该函数的参数会被写入mtimecmp中，当处理器执行的tick数超过该值时会触发中断。建议每秒触发时钟中断数为200次左右，大家可以根据timebase来算出每次中断的tick数。换句话说，想让`sbi_set_timer`隔X个tick后触发中断需要写作类似于`sbi_set_timer(current_tick() + X)`这个样子，也就是当前时刻加上希望多少个tick后触发中断。获取当前tick的函数被定义在了time.c中。

kernel/syscall/syscall.c下面的syscall全局变量的用途是什么?

根据理论课上的知识，系统调用的流程是用户将系统调用号放到寄存器中（在RISC-V中是a7），之后触发系统调用（ecall）。硬件切换到内核态交由操作系统处理（RISC-V会切换特权级到S态并跳转到stvec记录的地址处）。操作系统根据系统调用号来调用对应的内核处理函数。这里的syscall数组正是用于实现该功能的。syscall数组是一个指针数组，数组下标即为系统调用号，数组的元素极为系统调用号对应的处理函数。这样，根据系统调用号就可以通过syscall函数指针数组直接索引到对应的处理函数并调用了。在C的语法中，可以直接将一个函数指针以类似函数的形式调用。

系统调用对应的例外是什么?

系统调用对应的scause的值是8，也就是手册上写的Environment Call From U-mode。请特别注意，这是一个异常(exception)，而不是中断(interrupt)。因为是由指令直接触发的，而不是来自外部的。scause中有一个看上去很像系统调用的中断“Software Interrupt”，这个中断是RISC-V的核间中断(IPI, Inter Processor Interrupt)，是多核处理器由一个核心发往另一个核心的中断。因为IPI是来自于另一个核心的，所以对于当前核心来说IPI是来自外部的，因此是一个中断，而不是异常。最后，再次强调，请大家注意区分哪个是系统调用的scause值。

time_base的含义是什么?

该变量存储了处理器1秒的tick数，也就是处理器的主频。因为QEMU和板子的频率差很多，所以我们在u-boot(QEMU上用的)和bbl(板子用的)中都提供了一个我们自己实现的sbi函数sbi_read_fdt，可以读出当前环境下的主频。

建议每秒中断200-1000次，中断次数高有利于暴露某些隐藏的问题。根据往年的教学经验，有些同学的代码在中断次数少的情况下就能跑出现象，每秒处理的中断一多，有一些进程保存恢复或者一些很隐蔽的问题就会暴露出来。甚至设置几种不同的时钟中断次数会出不同的错误。不过如果中断次数设置得过高，可能会导致大量的时间全部用在了处理中断上面，用户程序会明显执行得比较慢。

csr.h中的SCAUSE_IRQ_FLAG会报警告?

task3的框架代码csr.h第35行，#define SCAUSE_IRQ_FLAG (1 << 63)应改为#define SCAUSE_IRQ_FLAG (1UL << 63)，否则使用的时候会报一个type width的warning。当然，这里更好的建议是大家自己在别的文件里面定义一个宏而不是用这里这个。因为csr.h同时会被汇编文件和C文件使用，而汇编器不太认识1UL这种写法（这种写法的含义是告诉C编译器常量1的类型是unsigned long的）。这也是为什么最初没有给大家加这个UL的原因。当然，Linux中用宏来实现了一些特殊机制让csr.h可以同时兼容C和汇编文件。感兴趣的同学也可以尝试用宏定义和预处理来解决这一问题。

另外，还有一种更简单直接的解决方式是将宏定义为0x80000000。这样虽然可读性略微差一点点，但是能够比较简单地解决这个问题。

gdb单步跟踪ecall/sret好像什么都没发生？

ecall/sret都是无法单步跟踪的。只能把断点设置在例外入口或sepc处。可以设置为临时断点（临时断点的gdb命令是tb，普通断点是b）

任务三的时候几个测试任务好像没法切换？

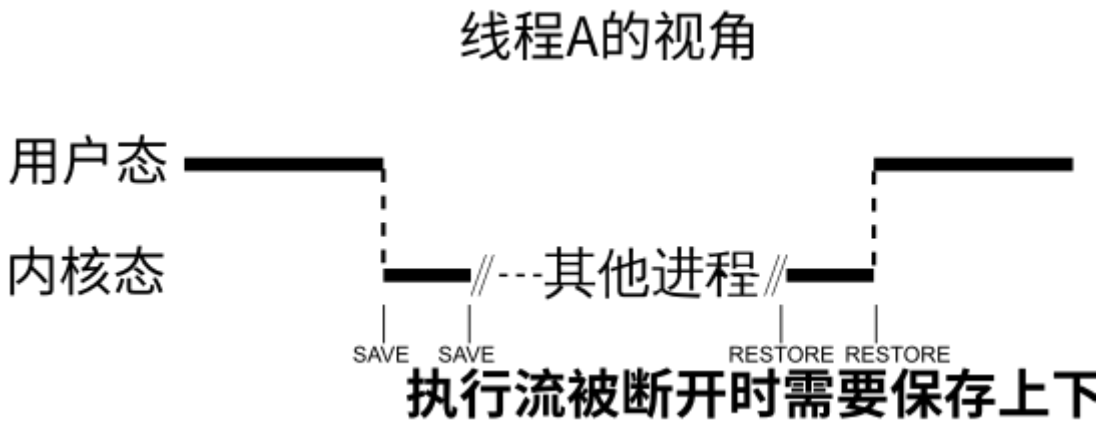
任务三用的测试程序中没有写`sys_yield`，大家做的时候可以自行添加上。测试任务四的时候再把所有测试程序中的`sys_yield`都注释掉就行。

irq.h中的TIMER_INTERVAL宏是做什么用的？

irq.h中的TIMER_INTERVAL宏是个历史遗留问题，不出意外的话应该用不到。

进行任务切换需要保存几次上下文？都需要保存哪些寄存器？

从内核中进行进程切换需要保存一次上下文，从用户态由系统调用或者中断进入内核时也需要保存一次上下文。



由于中断/异常进入到内核态时，需要假定用户的所有寄存器都是有用的，所以需要保存全部的通用寄存器和几个关键的csr寄存器（sstatus、sepc等）。在内核态，从一个进程切到另一个的时候，由于是内核主动调用一个函数切走的，所以按照C语言的约定，函数的被调用方只需要保证保留寄存器不发生变化就可以。所以在这一次进程切换中，只需要保存保留寄存器（其余的寄存器如果有用的话，会被编译器自动压在栈上）。

这里有一点比较容易想不到就是为什么不能switch to回来以后直接sret，而是还得恢复内核态的上下文然后再从ret_from_exception处返回。在一些操作系统的设计中，恢复上下文永远是最后一步。这种情况下可以只有一个上下文。但在我们的系统中，switch_to或者说do_scheduler可能一些系统调用中被调用，没法保证例外返回是最后一步。比如在获取锁的时候，被阻塞时会调用do_scheduler，但等到进程被切换回来继续执行以后，还需要继续执行后面的内核代码。

SIE寄存器和SSTATUS寄存器的SIE位的区别是什么？为什么要有这么两个功能相近的设置，只保留一个可不可以？

SIE寄存器是软件用来开启/关闭例外的，由软件控制，主要控制外部中断的开关。SSTATUS寄存器的SIE位是软硬件均可控制的。

硬件自动开关中断的功能是借由SSTATUS寄存器的SIE位实现的。例外发生时，硬件会将SPIE位设置为SIE位的值，同时关闭SIE位。当执行sret例外返回时，硬件会将SIE位的值设置为SPIE位的值。由于需要保证内核态下，例外保持屏蔽状态，所以一般软件只操作SPIE位，对于SIE位的值除非必要否则不主动控制。

screen_write/screen_reflush等应该怎么用?

在任务一、二中我们是调用`port_write`或`vt100_move_cursor`等函数直接输出到串口/控制光标。在后面的实验中，我们改为采用screen缓冲区来实现输出。在`screen.c`中我们提供了一系列维护屏幕缓冲区的函数，它的原理是先调用`screen_write`输出到屏幕缓冲区，然后每隔一段时间调用`screen_reflush`刷新屏幕。刷新屏幕的开销较大，建议每次处理时钟中断的时候刷新一次。

SAVE_CONTEXT中的_restore_kernel_tpsp等代码有什么作用?

SAVE_CONTEXT的代码建议整体按照自己的理解重写，start code给的部分可以直接删掉。原本start code的写法是为了支持内核态中断的情况，之前是为了允许内核态的测试任务也允许时钟中断所以才这么写的。但我们现在的实验要求不需要大家跑内核态的测试程序。任务三四五及后面所有实验的测试程序都是用户态的程序。所以没必要设计得这么复杂。且根据前一届同学的经验，这段代码弄不清楚的话很容易出各种奇怪的错误，所以建议整个SAVE_CONTEXT都按照自己的理解重写。

实现SAVE_CONTEXT的时候需要注意一个细节的问题，我们的32个通用寄存器都是需要保存的。在保存之前我们不能破坏现场。但是`sd`指令需要用一个寄存器来寻址。因此，总是需要把一个寄存器的值冲掉，加载成我们的pcb的地址。RISC-V为这种情况提供了一个特殊的寄存器：sscratch。可以临时用CSR指令把寄存器原来的值存放在sscratch中，然后再用这个已经被暂存的寄存器来寻址。

建议的一种实现方式是：在内核中，tp寄存器存放current_running的值。在例外返回前，将tp的值写入到sscratch寄存器。然后进入例外的时候，用`csrrw tp, CSR_SSCRATCH, tp`指令将当前tp的值存入sscratch，将原先sscratch中的值取出到tp（相当于将用户程序的tp的值和sscratch中的值交换了一下）。这样当前tp的值就又是current_running的值了，而用户态程序原本的tp被临时存放进了sscratch。接下来就可以用tp来找到pcb结构体中的`user_sp`和`kernel_sp`等域用于切换用户态/内核态栈指针。后面就可以顺理成章地将余下的寄存器全都保存下来了。

stval是做什么用的?

stval寄存器存放的是例外带有的额外信息。目前在RISC-V上，只有和访存相关的例外会使用该寄存器。如果发生任何访存相关的例外，stval中会存放发生例外的内存地址。在实验二中，暂时不会用到该寄存器。

参考资料

1. selfimpr1991, “ 【linux 内核数据结构】最为经典的链表 list.” https://blog.csdn.net/wenqian1991/article/details/44515713, 2015. [Online; accessed 23-September-2021].
2. full_of_bull, “Linus:利用二级指针删除单向链表.” https://coolshell.cn/articles/8990.html, 2013. [Online; accessed 23-September-2021].