PRJ 2

PRJ 2
- 杂记
  - 1. 疑难杂症：终端不回显
- TASK 1
  - 1. 思路分析
    - 1.1 PCB的初始化
    - 1.2 内核进程切换的实现
  - 2. Debug过程
    - 2.1 初始化current_running 指针
    - 2.2. 结点出队时未删干净
    - 2.3. 内存分配策略理解有误
    - 2.4. 关于栈空间的一些迷思
    - 2.5. tp==current_running?
- TASK 2
  - 1. 思路分析
    - 1.1. 队列API的完善
  - 1.2 调度器对进程的挂起&解挂
    - 1.2. 实现互斥锁
  - 2. Debug过程
    - 2.1 忘记判断对应资源是否已配锁
    - 2.2 队列的next问题
    - 2.3 解锁后BQ的处理
- TASK 3
  - 1. 思路分析
    - 1.1 宏开关控制
    - 2.2 系统调用的flow
    - 2.3 sp和ra到底存?
  - 2. Debug过程
    - 2.1 STVEC和stvec?
    - 2.2 内联汇编的痛苦挣扎
      - 2.2.1 基本语法的学习
      - 2.2.2 任性的编译器
      - 2.2.3 PRJ1的遗留问题
      - 2.2.4 内联汇编debug技巧
      - 2.2.5 内联汇编返回值的处理
    - 2.3 还有多久才能走进用户态……
    - 2.4 tp指针的初始化
    - 2.5 interrupt_helper之后…
- TASK 4
  - 1. 思路分析
    - 1.1 set_timer的功能
    - 1.2 唤醒进程的时机?
  - 2. Debug过程
- TASK 5
  - 1. 思路分析
  - 2. 杂记
    - 2.1 无效准备
    - 2.2 测试用例的设计

杂记

1. 疑难杂症：终端不回显

发现有时候core dumped后linux终端直接无法回显，但是其它分屏都可正常运行，且重启后恢复正常。

查找资料得知这可能与linux的伪终端机制有关

（具体触发原因我也没找到详细的解释），可通过`stty`指令修改终端驱动程序的配置。

```
stty -echo # 关闭回显
stty echo # 打开回显
```

> Linux 的伪终端的基本原理 及其在远程登录（SSH，telnet等）中的应用 - zzdyyy - 博客园 (cnblogs.com)
>
> 浅析Linux中stty命令的作用、常用用法及案例使用 - 古兰精 - 博客园 (cnblogs.com)

TASK 1

1. 思路分析

1.1 PCB的初始化

- 初始化pid0_pcb：即使其栈指针已经给出，也需要为之栈指针alloc：

```
const ptr_t pid0_stack = INIT_KERNEL_STACK + PAGE_SIZE;
pcb_t pid0_pcb = {
    .pid = 0,
    .kernel_sp = (ptr_t)pid0_stack,
    .user_sp = (ptr_t)pid0_stack
};
```

```
ptr_t allocKernelPage(int numPage)
{
    // align PAGE_SIZE
    ptr_t ret = ROUND(kernMemCurr, PAGE_SIZE);
    kernMemCurr = ret + numPage * PAGE_SIZE;
    ret = ROUND(kernMemCurr, PAGE_SIZE);
    return ret;
}
```

```
#define PAGE_SIZE 4096 // 4K
#define INIT_KERNEL_STACK 0x50500000
#define INIT_USER_STACK 0x52500000
#define FREEMEM_KERNEL (INIT_KERNEL_STACK+PAGE_SIZE)
```

  由图可知，若未为之alloc，讲导致第一个非内核程序（此处还不能称为用户程序，其和内核程序未明确分开）和kernel取到的内核栈指针都是`0x50501000`

- 初始化非内核程序的pcb
  - step1：从seq中读取程序（暂不能称为用户程序！）
  - step2：分配pid，从1开始（0是内核程序的pid）
  - step3：初始化pcb栈
    - switch_to的ra？ → 程序入口
    - switch_to的sp？ → 内核栈！！！
  - step4：加入ready队列，等待调度

1.2 内核进程切换的实现

> 做完例外后复盘：这是内核主动做出调度，故只需保存重要的寄存器，而例外是东窗事发，暂不知哪些才是有用信息，故需保存所有寄存器。该过程切换的是**内核栈**！

执行的flow：

- do_scheduler

    - 从RQ中选择一个task（FCFS）
    - 把当前task放到队尾
    - 修改current_running
    - 调用switch_to
- switch_to
    - 保存当前重要寄存器
    - 切换tp
    - 把main中准备的新任务的switch_to相关的信息取出

2. Debug过程

2.1 初始化current_running 指针

估计写得快睡着了，开始一通瞎写：

```
    /* TODO: [p2-task1] remember to initialize 'current_running' */
    *current_running = pid0_pcb;
```

程序运行时core dumped，单步跟踪：

```
Breakpoint 1, init_pcb (seq_start_loc=27678, seq_end_loc=27714) at ./init/main.c:138
138         *current_running = pid0_pcb;
(gdb) s
Remote connection closed
(gdb)
```

发现运行到此处时就会崩盘。定睛一看才发觉自己写写了个寂寞：应该是pid0_pcb取地址赋值给current_running，而不是把前者解引用.....

修改后再次debug跟踪，发现ra的值为0，导致切换进程出错：

```
(gdb)
switch_to () at ./arch/riscv/kernel/entry.S:78
78          ld ra, SWITCH_TO_RA(sp)
(gdb) p $fp
$1 = (void *) 0x50205d80 <pcb>
(gdb) s
80          ld s0, SWITCH_TO_S0(sp)
(gdb) p $ra
$2 = (void (*)()) 0x0
```

2.2. 结点出队时未删干净

```
#define INIT_KERNEL_STACK 0x50500000
#define INIT_USER_STACK 0x52500000
#define FREEMEM_KERNEL (INIT_KERNEL_STACK+PAGE_SIZE)
```

看到了小飞机，但是不能左右飞行.....而且print1，print2输出不对头？

```
> [INIT] SCREEN initialization succeeded.
> [TASK] This task is to test scheduler. (0)

                                                                  _
                                                                -=\`\
                                                            |\ ____\_\__
                                                          -=\c`""""""""" "`)
                                                             `~~~~~/ /~~`
                                                               -==/ /    [U-BOOT
] ERROR: truly_illegal_insn                                      '-'
exception code: 2 , Illegal instruction , epc 0 , ra 0
### ERROR ### Please RESET the board ###
```

gdb跟踪：发现在执行第一个task后跳过了第二个task：

```
(gdb)
get_pcb (node=0x50205e80 <pcb+128>)
    at ./kernel/sched/sched.c:83
83          for(int i=0;i<NUM_MAX_TASK;i++){
(gdb)
84              if(node == &pcb[i].list)
(gdb)
83          for(int i=0;i<NUM_MAX_TASK;i++){
(gdb)
84              if(node == &pcb[i].list)
(gdb)
83          for(int i=0;i<NUM_MAX_TASK;i++){
(gdb) p i
$6 = 1
(gdb) s
84              if(node == &pcb[i].list)
(gdb)
85                  return &pcb[i];
(gdb) p i
$7 = 2
(gdb)
```

发现封装函数后忘记把原本的摘除结点操作删除：

```
        add_node_rq(current_running);
    }
    list_node_t* tmp = seek_ready_node();

    current_running = get_pcb(tmp);
    current_running->status = TASK_RUNNING;

    ready_queue.next = ready_queue.next->next;

    // TODO: [p2-task1] switch_to current_running
    switch_to(prior_running, current_running);
```

同时为了避免摘除的结点使用指针对prev和next再做修改，将之相应域置为NULL：

```
list_node_t* seek_ready_node(){
    list_node_t *p = ready_queue.next;
    // delete p from queue
    ready_queue.next = p->next;
    p->next->prev = &ready_queue;
    p->next = NULL;
    p->prev = NULL;
    return p;
}
```

2.3. 内存分配策略理解有误

> 这是没仔细读mm.c文件的血淋淋教训.....

修改之后发现调用第二个任务时ra会变为0，情况如图：

```
> [INIT] SCREEN initialization succeeded.  [U-BOOT] ERROR: truly_i
llegal_insnis task is to test scheduler. (0)
exception code: 2 , Illegal instruction , epc 0 , ra 0
### ERROR ### Please RESET the board ###
```

gdb查看init_pcb_stack时的情况：

```
(gdb) s
81          pt_switchto->regs[0] = entry_point;     // ra
(gdb) x/10x pcb->kernel_sp
0x50500e70:     0x52010054      0x00000000      0x50500e70      0x000
00000
0x50500e80:     0x00000000      0x00000000      0x00000000      0x000
00000
0x50500e90:     0x00000000      0x00000000
(gdb) x/10x pcb[1].kernel_sp
0x0:    0x00000000      0x00000000      0x00000000      0x00000000
0x10:   0x00000000      0x00000000      0x00000000      0x00000000
0x20:   0x00000000      0x00000000
(gdb) s
82          pt_switchto->regs[1] = pcb->kernel_sp;  // sp
(gdb) x/10x pcb[1].kernel_sp
0x0:    0x00000000      0x00000000      0x00000000      0x00000000
0x10:   0x00000000      0x00000000      0x00000000      0x00000000
0x20:   0x00000000      0x00000000
(gdb) x/10x pcb->kernel_sp
0x50500e70:     0x520401cc      0x00000000      0x50500e70      0x000
00000
0x50500e80:     0x00000000      0x00000000      0x00000000      0x000
00000
```

发现写第二个pcb的时候覆盖了第一个，同时注意到二者传入的kernel stack似乎是一致的：

```
Breakpoint 1, init_pcb_stack (kernel_stack=1347424256, user_stack=1380974592,
    entry_point=1375797332, pcb=0x50205e38 <pcb>) at ./init/main.c:71
71              (regs_context_t *)(kernel_stack - sizeof(regs_context_t));

(gdb) c
Continuing.

Breakpoint 1, init_pcb_stack (kernel_stack=1347424256, user_stack=1380974592,
    entry_point=1375994316, pcb=0x50205e70 <pcb+56>) at ./init/main.c:71
71              (regs_context_t *)(kernel_stack - sizeof(regs_context_t));

(gdb) x/10x pcb->kernel_sp
0x50500e70:     0x52010054      0x00000000      0x50500e70      0x00000000
0x50500e80:     0x00000000      0x00000000      0x00000000      0x00000000
0x50500e90:     0x00000000      0x00000000
(gdb) x/10x pcb[1].kernel_sp
0x0:    0x00000000      0x00000000      0x00000000      0x00000000
0x10:   0x00000000      0x00000000      0x00000000      0x00000000
0x20:   0x00000000      0x00000000
```

意识到自己可能未正确理解栈分配函数。仔细看其实现，参数numPage表示其具体要分配几页，而我先前草草扫了一眼，误以为是根据`base+numPage*SIZE`的策略进行内存分配，直接把当前的task个数当作参数🐻.....

修改后可运行三个指定程序，但无法再调度回原本执行了一半的程序：

```
> [TASK] This task is to test scheduler. (0)
> [TASK] This task is to test scheduler. (0)
```



```
exception code: 2 , Illegal instruction , epc 5020252e , ra 5020172a
### ERROR ### Please RESET the board ###
```

2.4. 关于栈空间的一些迷思

main里的switch_to内容中的stack是哪个栈？switch_to的时候在使用谁的栈？

起初我查资料的时候看到了这样一段话：

> 进程由于中断而陷入到内核态，进程进入内核态之后，首先把用户态的堆栈地址保存在内核态堆栈中，然后设置堆栈寄存器地址为内核栈地址，这样就从用户栈转换成内核栈。
>
> 当进程从内核态转换到用户态时，将堆栈寄存器的地址再重新设置成用户态的堆栈地址（即终端前进程在用户态执行的位置），这一过程也成为现场恢复

于是我大义凛然地把用户栈存进了switch_to内容中的stack，这在task1还能勉强跑过，后续再加上上下文切换后就寄透了。

后续仔细阅读讲义，可知**目前的程序和内核程序未作明显区分，也即二者还是运行在同一个特权级别的**，故当前应该使用的都是内核栈。

2.5. tp==current_running?

起初我以为更新了tp后，会有奇妙的机制帮我把current_running也一并改了，故在`do_scheduler`中未对之进行更新。

后续单步跟踪时发现：

```
(gdb)
81        addi fp, a1, 0
(gdb)
82        ld t0, PCB_KERNEL_SP(a1)  # t0 is the kernel sp of n
ext task
(gdb) p current_running
$9 = (pcb_t * volatile) 0x502037e8 <pid0_pcb>
(gdb) p $fp
$10 = (void *) 0x50205e58 <pcb+56>
(gdb)
```

在entry中指针已经更新，而current_running其始终指向pid0_pcb，睁大我的狗眼仔细看了看讲义：

然这个过程中难免会有补丁报错的情况，请同学们根据报错的位置再去手动修改对应的文件。下面的 Project2 任务 1 的实验步骤：

1. 完成 sched.h 中 PCB 结构体设计，以及 main.c 中 init_pcb 的 PCB 初始化方法。
2. 实现 entry.S 中的 switch_to 汇编函数，使其可以将当前运行进程的执行现场保存在 current_running 指向的 PCB 中，以及将 current_running 指向的 PCB 中的执行现场进行恢复。注意，请在 switch_to 时保证，current_running 同时也被保存在了 tp 寄存器中。代码的其他部分假定了 tp 和 current_running 是等价的。
3. 实现 sched.c 中 do_scheduler 方法，使其可以完成任务的调度切换。

我是怎么把它写成fp指针的？？？？修改为tp指针：

```
(gdb)
81          addi tp, a1, 0
(gdb)
82          ld t0, PCB_KERNEL_SP(a1)  # t0 is the kernel sp of next task
(gdb) p current_running
$5 = (pcb_t * volatile) 0x502037e8 <pid0_pcb>
(gdb) p $tp
$6 = (void *) 0x50205e58 <pcb+56>
(gdb)
```

再仔细看看讲义：

然这个过程中难免会有补丁报错的情况，请同学们根据报错的位置再去手动修改对应的文件。下面的 Project2 任务 1 的实验步骤：

1. 完成 sched.h 中 PCB 结构体设计，以及 main.c 中 init_pcb 的 PCB 初始化方法。
2. 实现 entry.S 中的 switch_to 汇编函数，使其可以将当前运行进程的执行现场保存在 current_running 指向的 PCB 中，以及将 current_running 指向的 PCB 中的执行现场进行恢复。注意，请在 switch_to 时保证，current_running 同时也被保存在了 tp 寄存器中。代码的其他部分假定了 tp 和 current_running 是等价的。
3. 实现 sched.c 中 do_scheduler 方法，使其可以完成任务的调度切换。

该不会是说两个都要改吧……

同时又发现一个不太聪明的bug：

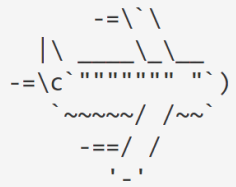
```
    if(current_running->pid != 0){
        // add to the ready queue
        current_running->status = TASK_READY;
        add_node_rq(current_running);
    }
```

居然直接把pcb结构体加进了queue里……

改一改就成功啦！

```
输出    终端    端口    问题

> [TASK] This task is to test scheduler. (9)
> [TASK] This task is to test scheduler. (9)
```



TASK 2

1. 思路分析

1.1. 队列API的完善

- 将add_node_rq（把元素加入RQ）改为更普适的add_node_to_q，需给出队列指针以及待入队元素；
- 加了删除元素的函数，由于是环形队列，可直接定位到元素的前驱后继，故传参时只需要传入元素本身；
- 在task1中额外维护了一个rq_tail的尾指针，task2意识到利用环形队列特性可快速定位队尾指针。

1.2 调度器对进程的挂起&解挂

挂起的过程：

- 根据pcb_node定位到pcb
- 修改PCB状态
- 加入block queue

解除挂起的过程：

解除挂起等价于双向循环队列删结点

- 根据pcb_node定位到pcb
- 将其前驱后继互指
- 将其前驱后继修改为NULL

1.2. 实现互斥锁

全局初始化：讲所有🔒的状态都置为unlock，并对每个锁的等待队列初始化；

互斥锁初始化(锁分配策略)：

- 维护一个全局变量`lock_num`，意为当前被使用的锁个数；

- 用户传入一个key（对应一种资源），判断该资源是否已经分配锁：
  - 未分配，直接分配当前`lock_num`指向的锁；
  - 已分配，无需额外分配
- 返回锁的句柄（lock_num）

申请互斥锁：

判断申请的锁是否lock：

- 是，加入WQ
- 否，分配锁，将锁置为LOCK

释放锁：

检查当前WQ：

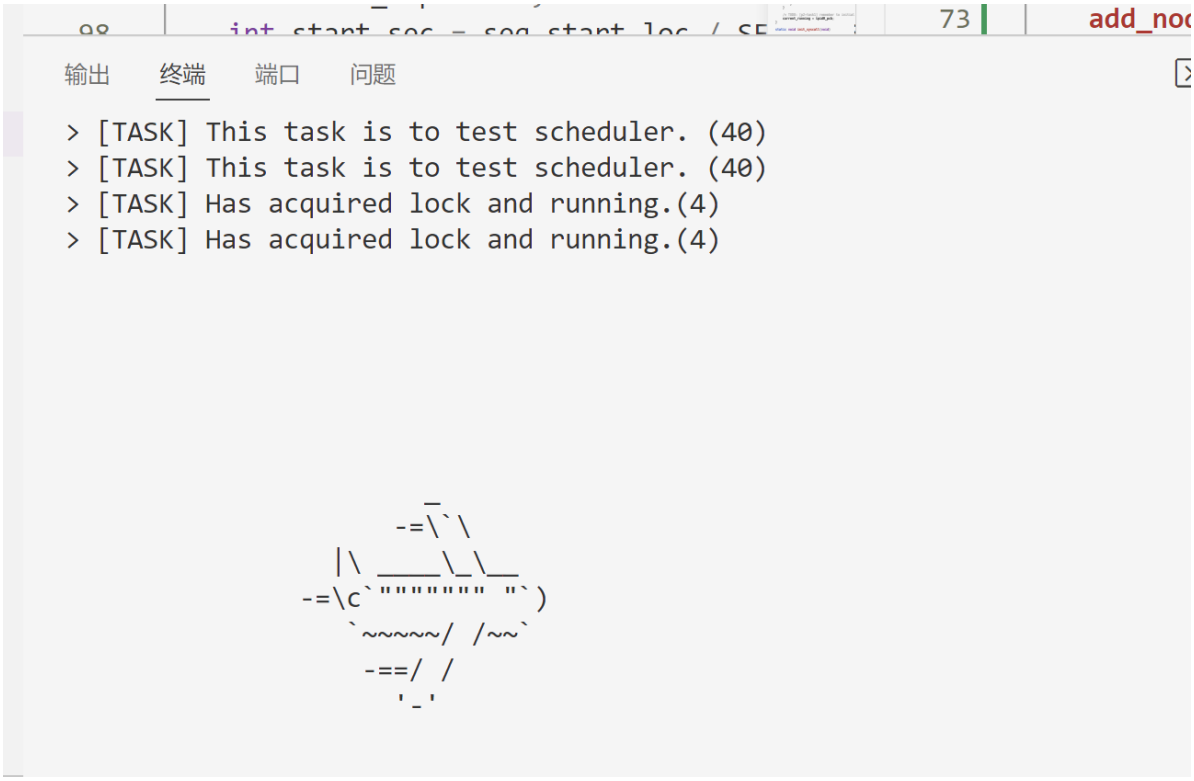
- 非空，取一个返回RQ，锁不释放
- 空，锁UNLOCK

> **上课后老师建议把所有进程皆放回RQ，由调度算法决定**，但由于给出的测试用例并未循环申请锁，若全部放回队列将导致其皆打印出"得到锁"（实际上锁仍为unlock状态），故最终还是采用"一次放一个"的策略。

2. Debug过程

2.1 忘记判断对应资源是否已配锁

发现两个进程同时获得了锁.....



意识到自己似乎没处理key值相等的case，但凡有初始化mutex锁的需求都分配了一个锁：

```c
int do_mutex_lock_init(int key)
{
    /* TODO: [p2-task2] initialize mutex lock */
    mlocks[lock_used_num].key = key;
    return lock_used_num++;
}
```

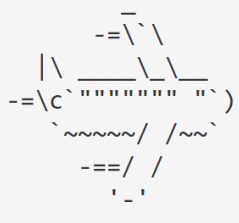
应该在分配前检查对应的key是否已经分配了lock：

```c
int do_mutex_lock_init(int key)
{
    /* TODO: [p2-task2] initialize mutex lock */
    // check whether this key has already got a lock
    for(int i=0;i<lock_used_num;i++){
        if(mlocks[i].key == key)
            return i;
    }
    mlocks[lock_used_num].key = key;
    return lock_used_num++;
}
```

此时可以看到lock1已经分配到了锁：

```
输出  终端  端口  问题

> [TASK] This task is to test scheduler. (1)
> [TASK] This task is to test scheduler. (1)
> [TASK] Has acquired lock and running.(0)
> [TASK] Applying for a lock.
```



2.2 队列的next问题

但是发现一旦跑完无法正常切换至lock2进程继续执行，gdb跟踪其释放锁的过程：

发现居然在对ready_queue的头结点在做unblock.....

```
74
(gdb)
do_mutex_lock_release (mlock_idx=0) at ./kernel/locking/lock.c:81
81          for(p=head->next; p!=head; p=p->next)
(gdb) s
82              do_unblock(p);
(gdb)
do_unblock (pcb_node=0x50203b20 <ready_queue>) at ./kernel/sched/sched.c:70
70          pcb_t * tmp = get_pcb_from_node(pcb_node);
(gdb) p p
No symbol "p" in current context.
(gdb)
```

仔细查看自己写的释放锁的函数：

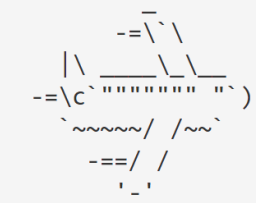
```
void do_mutex_lock_release(int mlock_idx)
{
    /* TODO: [p2-task2] release mutex lock */
    // 释放锁
    spin_lock_release(&mlocks[mlock_idx].lock);
    // 该锁对应的block队列中的所有进程置为ready
    list_node_t* p, *head;
    head = &mlocks[mlock_idx].block_queue;
    for(p=head->next; p!=head; p=p->next)
        do_unblock(p);
}
```

在执行完do_unblock后p本身已经被插入ready_queue，故其作为队尾，下一个结点已经变为ready_queue，故应该在此前就保留队列中下一个元素。

```
for(p=head->next; p!=head;){
        tmp = p->next;
        do_unblock(p);
        p = tmp;
    }
}
```

yes人生赢家：

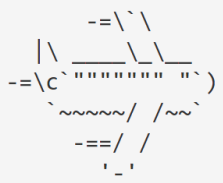
```
> [TASK] This task is to test scheduler. (8)
> [TASK] This task is to test scheduler. (8)
> [TASK] Has acquired lock and running.(0)
> [TASK] Has acquired lock and running.(1)
```



2.3 解锁后BQ的处理

好景不长，跑了一会就出事了：

```
> [TASK] This task is to test scheduler. (35)
> [TASK] This task is to test scheduler. (35)
> [TASK] Has acquired lock and running.(1)
> [TASK] Has acquired lock and running.(2)
```



发现两个进程又同时获得了锁。发现lock1释放锁后没有lock2申请锁的过程，自动转入了lock1再度申请锁的过程：

查看申请锁程序的代码：

```
        kernel_yield();

        kernel_mutex_acquire(mutex_id);

        for (int i = 0; i < 5; i++)
        {
            kernel_move_cursor(0, print_location);
            kernel_print("> [TASK] Has acquired lock and running.(%d)\n", i, 0);
            kernel_yield();
        }

        kernel_move_cursor(0, print_location);
        kernel_print("%s", (long)blank, 0);

        kernel_move_cursor(0, print_location);
        kernel_print("> [TASK] Has acquired lock and exited.\n", 0, 0);

        kernel_mutex_release(mutex_id);

        kernel_yield();
    }
```

在一次`kernel_mutex_acquire`中其未获得锁，便被置于阻塞队列，此后被调入ready队列后其不会再度申请获得锁，而是直接往下执行。同时仔细阅读讲义，要求其进入就绪队列后，“并获得锁”，即但凡队列不空，就不该把当前的锁unlock，同时一次不应该把所有的阻塞队列里的进程都放回ready队列，而是一次只放回一个：

```c
void do_mutex_lock_release(int mlock_idx)
{
    /* TODO: [p2-task2] release mutex lock */
    list_node_t* p, *head;
    head = &mlocks[mlock_idx].block_queue;
    p = head->next;
    // 阻塞队列为空，释放锁
    if(p==head)
        spin_lock_release(&mlocks[mlock_idx].lock);
    else
        do_unblock(p);
}
```

TASK 3

1. 思路分析

1.1 宏开关控制

此处引入了态的概念，我们需要转换API，但频繁修改注释真的很麻烦（尤其是做到了后面需要验收前面的时候再切换的时候）……

先是一个不太聪明的控制方法：

```c
#include <stdio.h>
#include <unistd.h>  // NOTE: use this header after implementing syscall!
#include <kernel.h>

/**
 * NOTE: kernel APIs is used for p2-task1 and p2-task2. You need to change
 * to syscall APIs after implementing syscall in p2-task3!
 */
#ifndef __UNISTD_H__
    int main(void)
```

用`unistd.h`里的宏控制通断，每次注释掉头文件即可。后续意识到可以直接额外定义一个宏放在`kernel.h`里，每次只要控制一个注释。意识到后已经懒得改了，但后续修改sys_yield之后就学聪明了，把`NO_SYS_YIELD`直接定义在kernel里：

```c
            sys_move_cursor(0, print_location);
            printf("> [TASK] This task is to test scheduler. (%d)", i);
        #ifndef NO_SYS_YIELD
            sys_yield();
        #endif
        }
    }
```

2.2 系统调用的flow

- 用户调用api
- invoke syscall用ecall跳转到etvec给出的地址（exception_handler_entry）
  - 设置etvec是在init_exception时调用了set_up_exception
- exception_handler_entry调用interrupt_helper
- interrupt_helper决定是系统调用还是中断，并调用相应的跳转表
  - 对于系统调用，会使用handle_syscall
- handle_syscall根据exec table调用

2.3 sp和ra到底存?

	regs_context	switchto_context
sp	用户栈	内核栈
ra	初始化：啥都行，只要把sepc置为函数入口	初始化：ret_from_exception

在进程从内核态进入用户态时需要经过`sret`，同时在刚刚启动进程时需做必要的寄存器初始化工作，而这需要在`ret_from_exception`完成，故在kernel switch to用户进程时先让其进入`ret_from_exception`，switch to的过程运行在内核态，相应的需要使用内核栈。

而例外发生前使用的是用户栈，故regs_context中存储的是用户栈，每次从例外返回后又将用户栈恢复。而初始若要进入用户程序，需要将sepc置为函数入口，因为调用`sret`后硬件会自动将pc值置为`sepc`中存储的值。

2. Debug过程

2.1 STVEC和stvec?

```
la t0, exception_handler_entry
csrw STVEC, t0
```

```
rnel/locking/lock.c ./kernel/sched/time.c ./kernel/sched/sched.c ./kernel/syscall/syscall.c
ntk.c ./libs/string.c
./arch/riscv/kernel/trap.S: Assembler messages:
./arch/riscv/kernel/trap.S:8: Error: unknown CSR `STVEC'
./init/main.c:153:5: warning: second argument of 'main' should be 'char **' [-Wmain]
```

后续改为小写就能过了，这居然是个大小写敏感的汇编寄存器!

2.2 内联汇编的痛苦挣扎

2.2.1 基本语法的学习

> 一些汇编基础知识：
>
> riscv下的GCC内联汇编_Yinwhe的博客-CSDN博客
>
> 一些RISCV系统调用的规范：
>
> system-calls - PK/Linux 上的 RISC-V ecall 系统调用约定 - IT工具网 (coder.work)

```
    asm volatile(
        "nop\n"
        "mv %%a0, %0\n"
        "mv %%a1, %1\n"
        "mv %%a2, %2\n"
        "mv %%a3, %3\n"
        "mv %%a4, %4\n"
        "mv %%a7, %5\n"
        "ecall\n"
        :
        :"r"(arg0),"r"(arg1),"r"(arg2),"r"(arg3),"r"(arg4),"r"(sysno)
    );
```

输出：

```
tiny_libc/syscall.c: Assembler messages:
tiny_libc/syscall.c:11: Error: illegal operands `li %x10,a5'
tiny_libc/syscall.c:12: Error: illegal operands `li %a1,a4'
tiny_libc/syscall.c:13: Error: illegal operands `li %a2,a3'
tiny_libc/syscall.c:14: Error: illegal operands `li %a3,a2'
tiny_libc/syscall.c:15: Error: illegal operands `li %a4,a1'
tiny_libc/syscall.c:16: Error: illegal operands `li %a7,a0'
make: *** [Makefile:164: build/syscall.o] Error 1
```

这和intel和mips的内联汇编似乎有所不同，在内联汇编中使用寄存器似乎无需在寄存器前使用两个`%`？

保留一个百分号后输出：

```
tiny_libc/syscall.c: In function 'invoke_syscall':
tiny_libc/syscall.c:10:5: error: invalid 'asm': operand number out of range
   10 |     asm volatile(
      |     ^~~
make: *** [Makefile:164: build/syscall.o] Error 1
```

不得行.....但是只能查到intel的内联汇编报出改错的原因（就是未加两个%）

> gcc - 错误 : invalid 'asm' : operand number missing after %-letter when using inline assembly with GCC - IT工具网 (coder.work)

后来阅读示例代码发现对于RISCV的汇编，引用寄存器时无需做出任何标记：

```
asm volatile(
                "sub a7, a7, a7;\
                mv s0, %1;\
                mv s1, %2;\
                add a2, s1, %3;\
                mv  a5, s1;\
....
                bne a2, a5, loop"
                :"=r" (sum) // %0
                :"r"(pA),   // %1
                "r"(pB),    // %2
                "r"(colCnt) // %3
                : "s0", "s1"
    );
```

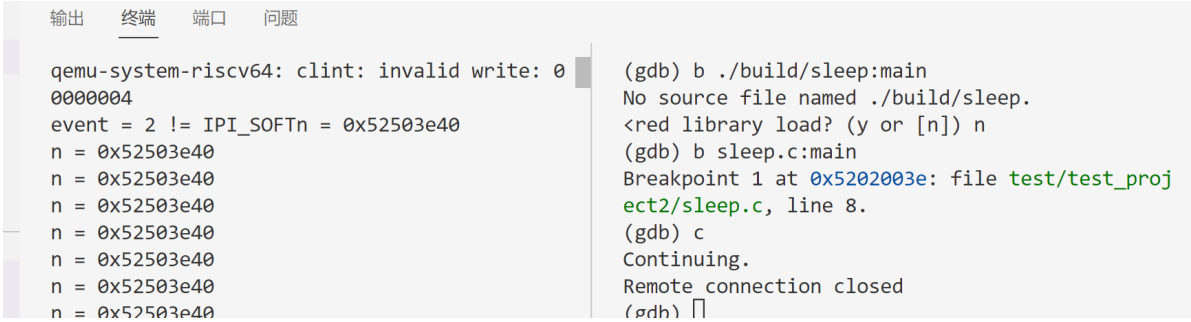
> 原文：risc-v GCC内嵌汇编 - sureZ_ok - 博客园 (cnblogs.com)

2.2.2 任性的编译器

开启1mol报错：

```
n = 0x52503e40
qemu-system-riscv64: clint: invalid write: 00000004
event = 2 != IPI_SOFTn = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
n = 0x52503e40
qemu-system-riscv64: clint: invalid write: 00000004
```

gdb断点打在sleep测试的main函数入口：



发现在调用main之前就已经报错，查看sleep的反汇编顺便发现自己写的invoke_syscall有问题：

```
5202010c:   fe843503            ld  a0,-24(s0)
52020110:   0001                    nop
52020112:   853e                    mv  a0,a5
52020114:   85ba                    mv  a1,a4
52020116:   8636                    mv  a2,a3
52020118:   86b2                    mv  a3,a2
5202011a:   872e                    mv  a4,a1
5202011c:   88aa                    mv  a7,a0
5202011e:   00000073            ecall
```

这种写法将导致a0寄存器的值被a5覆盖，并在mv a7, a0时错把a5的值传递给a7，应在最初就将a5中的值传给a7，修改后发现报错变了：

```
n = 0x17
qemu-system-riscv64: clint: invalid write: 0
0000004
event = 4 != IPI_SOFTn = 0x16
n = 0x14
n = 0x17
n = 0x16
n = 0x14
n = 0x17
n = 0x16
n = 0x14
n = 0x17
n = 0x16
n = 0x14
n = 0x17
n = 0x16
n = 0x14
n = 0x17
```

还是不太理解这个报的是啥错捏……

同时注意到传参似乎是倒着来的，也就是sysno分配的是a5寄存器？（编译器真的好任性啊）

```
    asm volatile(
        "nop\n"
        "mv a7, %0\n"
        "mv a0, %1\n"
        "mv a1, %2\n"
        "mv a2, %3\n"
        "mv a3, %4\n"
        "mv a4, %5\n"
        "ecall\n"
        :
        :"r"(sysno),"r"(arg0),"r"(arg1),"r"(arg2),"r"(arg3),"r"(arg4)
    );
```

```
    52020110:   0001                    nop
    52020112:   88be                    mv  a7,a5
    52020114:   853a                    mv  a0,a4
    52020116:   85b6                    mv  a1,a3
    52020118:   8632                    mv  a2,a2
    5202011a:   86ae                    mv  a3,a1
    5202011c:   872a                    mv  a4,a0
    5202011e:   00000073            ecall
```

2.2.3 PRJ1的遗留问题

发现此时ra给出的地址不是4倍数对齐的：

```
100       addi t1, t1, SWITCH_TO_SIZE
(gdb)
101       sd t1, PCB_KERNEL_SP(a1)  # revise the kerne
l sp, write back to pcb
(gdb) p $ra
$4 = (void (*)()) 0x52020152 <sys_move_cursor+28>
(gdb) n
103       jr ra
(gdb)
Remote connection closed
```

原因是我最初写的loader中每个程序的任务不是严格放在给定的程序起始位置的，而是采取了就读入位置进行移位的方法（这种方法给后续 debug也带来很大的阻碍，因为程序的实际指令和反汇编代码并非完全对应的，add-symbol-file的时候会无法跟踪到真正的指令流）。修改loader和task_info_t结构体后确保指令和反汇编代码对齐：

```
uint64_t load_task_img(char *taskname){
    int i;
    int entry_addr;
    int start_sec;
    int blocknums;
    for(i=0;i<TASK_MAXNUM;i++){
        if(strcmp(taskname, tasks[i].taskname)==0){
            entry_addr = TASK_MEM_BASE + TASK_SIZE * i;
            start_sec = tasks[i].start_addr / SECTOR_SIZE;
 // 起始扇区：向下取整
            blocknums = NBYTES2SEC(tasks[i].task_size + tasks[i].start_addr) -
tasks[i].start_addr / SECTOR_SIZE;
            bios_sdread(entry_addr, blocknums, start_sec);
            memcpy(entry_addr, entry_addr + (tasks[i].start_addr -
start_sec*512), tasks[i].task_size);
            return entry_addr;  // 返回程序存储的起始位置
        }
    }
    // 匹配失败，提醒重新输入
    char *output_str = "Fail to find the task! Please try again!";
    for(i=0; i<strlen(output_str); i++){
        bios_putchar(output_str[i]);
    }
    bios_putchar('\n');
    return 0;
}
```

发现汇编结束后指令就跳转到了奇怪的地方。

仔细查看此时的内联汇编代码：

```
    asm volatile(
        "nop\n"
        "mv a7, a0\n"
        "mv a0, a1\n"
        "mv a1, a2\n"
        "mv a2, a3\n"
        "mv a3, a4\n"
        "mv a4, a5\n"
        "ecall\n"
        "ret\n"
    );
```

意识到ecall之后ra寄存器的值已经发生改变，应该将之记录在栈里。同时想起来在main函数里忘记对sepc初始化，应该设置为exception_handler_entry

中断or系统调用时栈的切换：

> 当进程由于中断或系统调用从用户态转换为内核态时，进程所使用的栈也要从用户栈切换到内核栈。系统调用实质就是通过指令产生中断（软中断）。进程由于中断而陷入到内核态，进程进入内核态之后，首先把用户态的堆栈地址保存在内核态堆栈中，然后设置堆栈寄存器地址为内核栈地址，这样就从用户栈转换成内核栈。
>
> 当进程从内核态转换到用户态时，将堆栈寄存器的地址再重新设置成用户态的堆栈地址（即终端前进程在用户态执行的位置），这一过程也成为现场恢复

2.2.4 内联汇编debug技巧

调用ecall后地址错误：

```
Continuing.

Breakpoint 1, invoke_syscall (sysno=22,
    arg0=0, arg1=4, arg2=0, arg3=0,
    arg4=0) at tiny_libc/syscall.c:10
10          asm volatile(
(gdb) p $estvc
$1 = void
(gdb) p $stvec
$2 = 1344278864
(gdb) p/x $stvec
$3 = 0x50201150
(gdb) x/10i $stvec
   0x50201150 <exception_handler_entry>:        sd   sp,8(tp) # 0x8
   0x50201154 <exception_handler_entry+4>:      ld   sp,0(tp) # 0x0
   0x50201158 <exception_handler_entry+8>:      addi sp,sp,-288
   0x5020115a <exception_handler_entry+10>:     sd   zero,0(sp)
   0x5020115c <exception_handler_entry+12>:     sd   ra,8(sp)
   0x5020115e <exception_handler_entry+14>:     sd   sp,16(sp)
   0x50201160 <exception_handler_entry+16>:     sd   gp,24(sp)
   0x50201162 <exception_handler_entry+18>:     sd   tp,32(sp)
   0x50201164 <exception_handler_entry+20><continue wi
```

查看寄存器内确实写入了入口地址，但断点打在入口exception_handler_entry，发现ecall并没有跳转到该处：

```
   0x50201150 <exception_handler_entry>:        sd   sp,8(tp) # 0x8
   0x50201154 <exception_handler_entry+4>:      ld   sp,0(tp) # 0x0
   0x50201158 <exception_handler_entry+8>:      addi sp,sp,-288
   0x5020115a <exception_handler_entry+10>:     sd   zero,0(sp)
   0x5020115c <exception_handler_entry+12>:     sd   ra,8(sp)
   0x5020115e <exception_handler_entry+14>:     sd   sp,16(sp)
   0x50201160 <exception_handler_entry+16>:     sd   gp,24(sp)
   0x50201162 <exception_handler_entry+18>:     sd   tp,32(sp)
   0x50201164 <exception_handler_entry+20><continue wi
<ut paging--b exception_handler_entry
:       sd      t0,40(sp)
   0x50201166 <exception_handler_entry+22>:     sd      t1,48(sp)
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000520200f8 in invoke_syscall
                                                   at tiny_libc/syscall.c:10
        breakpoint already hit 1 time
(gdb) b exception_handler_entry
Breakpoint 2 at 0x50201150: file ./arch/riscv/kernel/entry.S, line 223.
(gdb) c
Continuing.
Remote connection closed
(gdb)
```

```
n = 0x16
[U-BOOT] ERROR: truly_ill
egal_insn
exception code: 2 , Illeg
al instruction , epc 4 ,
ra 4
### ERROR ### Please RESE
T the board ###
QEMU: Terminated
stu@stu:~/linmengying20/P
roject2_SimpleKernel$
```

同时后续pc变成奇怪的值，也不知道发生了啥……但是内联汇编单步调试会被掉过？在内联汇编里多加几个label：

```
"mv a3, a4\n"
"mv a4, a5\n"
"debug1:\n"
"ecall\n"
"debug2:\n"
"mv %0, a0\n"
:"=r"(ret_value)
```

后续打断点的时候可以hit到label上。

2.2.5 内联汇编返回值的处理

起初我在内联汇编中直接写了`ret`，意识到在内联汇编中就返回将导致栈指针错误：

```
0000000052020106 <debug1>:
    52020106:   00000073            ecall

000000005202010a <debug2>:
    5202010a:   8082                ret
    5202010c:   4781                li  a5,0
    5202010e:   853e                mv  a0,a5
    52020110:   7462                ld  s0,56(sp)
    52020112:   6121                addi    sp,sp,64
    52020114:   8082                ret
```

一种思路是将后续的反汇编代码转移至内联汇编的ret前，这种做法过于马后炮分析，不采用；但若简单地return 0又将导致返回值有误，故可额外开一个参数在内联汇编中接受返回值，并在return中将之返回。

```
(gdb) c
Continuing.

Breakpoint 1, 0x0000000052020106 in invoke_syscall (sysno=22, ar
g0=0,
    arg1=4, arg2=0, arg3=0, arg4=0)
    at tiny_libc/syscall.c:11
11          asm volatile(
(gdb) p $stvec
$1 = 1344278868
(gdb) x/10i $stvec
   0x50201154 <exception_handler_entry>:        sd      sp,8(tp)
 # 0x8
   0x50201158 <exception_handler_entry+4>:      ld      sp,0(tp)
 # 0x0
   0x5020115c <exception_handler_entry+8>:      addi    sp,sp,-2
```

2.3 还有多久才能走进用户态……

可确保stvec已经正确设置，但死活不跳转?

查看当前sstatus：

```
(gdb) p $sstatus
$2 = -9223372036854751232
(gdb) p/x $sstatus
$3 = 0x8000000000006000
(gdb) i b
```

意识到此时并未切换到用户态！还是运行在特权态，导致ecall会进入M态特权处理

S 模式处理例外的行为已和 M 模式非常相似。如果 hart 接受了异常并且把它委派给了 S 模式，则硬件会原子地经历几个类似的状态转换，其中用到了 S 模式而不是 M 模式的 CSR：

- 发生例外的指令的 PC 被存入 sepc，且 PC 被设置为 stvec。
- scause 按图 10.3 根据异常类型设置，stval 被设置成出错的地址或者其它特定异常的信息字。
- 把 sstatus CSR 中的 SIE 置零，屏蔽中断，且 SIE 之前的值被保存在 SPIE 中。
- 发生例外时的权限模式被保存在 sstatus 的 SPP 域，然后设置当前模式为 S 模式。

后续意识到内核程序切换到第一个程序时存在一个特权级切换的问题，即应该从内核态→用户态，阅读讲义可知这个过程是由`sret`实现的，考虑的处理方式是在switch_to时将ra置为`ret_from_exception`，并在例外的上下文中把`sepc`置为函数入口（在调用sret后pc会自动跳转到sepc指向的值）。

此部分参考的资料：

> RISC-V 寄存器 - 知乎 (zhihu.com)

2.4 tp指针的初始化

进入用户态后报错如下：

```
exception code: 5 , Load access fault , epc 50201
14a , ra 502010e8
### ERROR ### Please RESET the board ###
```

```
0x000000005ffcabb0 in ?? ()
(gdb) p/10i 5020114a
Item count other than 1 is meaningless i
n "print" command.
(gdb) x/10i 0x5020114a
   0x5020114a <ret_from_exception+98>: l
d       sp,8(tp) # 0x8
   0x5020114e <ret_from_exception+102>:s
ret
   0x50201152:  nop
   0x50201154 <exception_handler_entry>:
sd      sp,8(tp) # 0x8
   0x50201158 <exception_handler_entry+4
>:      ld      sp,0(tp) # 0x0
```

> Load access fault refers to using load or store to an unmapped or otherwise protected address.
>
> The atomic operations are given their own exception numbers (not sure why).
>
> And lastly, they can be caused by switching between privilege modes (U,S,H,M)

意识到此时tp寄存器被修改了，不再指向current_running，故在pcb_stack_init时加上：

```
    pt_regs->regs[4] = pcb;                         // tp
```

同时跟踪时发现gp变了，但全局指针不知道咋初始化，，，不能轻举妄动，先注释掉.....

```
$4 = (void *) 0x50501ee0
(gdb) s
165         ld ra, OFFSET_REG_RA(sp)
(gdb) s
168         ld gp, OFFSET_REG_GP(sp)
(gdb) p $ra
$5 = (void (*)()) 0x52010000
(gdb) p $gp
$6 = (void *) 0x5f771d98
(gdb) s
169         ld tp, OFFSET_REG_TP(tp)
(gdb) p $gp
$7 = (void *) 0x0
(gdb)
行 215，列 48    空格: 2    UTF-8    LF    GAS/AT&T x86/x64
```

解锁新报错：

```
exception code: 7 , Store/AMO access fault , epc 520100
00 , ra 52010000
### ERROR ### Please RESET the board ###
```

发现tp写成sp，且应该在内核栈指针+`OFFSET_SIZE`前把用户栈指针还原

> opensbi下的riscv64裸机编程2(中断与异常) - 腾讯云开发者社区-腾讯云 (tencent.com)

debug时的有趣发现：不是在kernel态的时候无法看到csr寄存器（至少证明我成功进入用户态了！）：

```
(gdb) s
invoke_syscall (sysno=22, arg0=0,
    arg1=0, arg2=0, arg3=0,
    arg4=0)
    at tiny_libc/syscall.c:11
11          asm volatile(
(gdb) p $stvec
Could not fetch register "stvec"; remote failure reply 'E
14'
(gdb)
```

2.5 interrupt_helper之后...

继续跟踪，发现在handle_syscall后回到了interrupt_helper，此后的返回地址似乎不对，理论上应该回到ret_from_exception

```
79      }
(gdb)
handle_syscall (regs=0x50501ee0,
    interrupt=0, cause=8)
    at ./kernel/syscall/syscall.c:16
16          regs->sepc += 4;
(gdb) p regs->sepc
$9 = 1375797408
(gdb) p/x regs->sepc
$10 = 0x520100a0
(gdb) s
17      }
(gdb) p/x regs->sepc
$11 = 0x520100a4
(gdb) n
interrupt_helper (
    regs=0x50501ee0, stval=0,
    scause=8)
    at ./kernel/irq/irq.c:22
22      }
(gdb) n
0x00000000502011d2 in ?? ()
(gdb) x/10i 0x502011d2
```

查看entry的实现：

```
 */
la ra, ret_from_exception

/* TODO: [p2-task3] call interrupt_helper
 * NOTE: don't forget to pass parameters for
 */
addi a0, sp, 0
csrr a1, stval
csrr a2, scause
call interrupt_helper
```

意识到调用完helper之后还需加上`ret`或者`jr ra`，返回到`ret_from_exception`。

终于不报错了，但啥也不打印?

```
Breakpoint 1, ret_from_exception ()
    at ./arch/riscv/kernel/entry.S:162
162         ld sp, PCB_KERNEL_SP(tp)  // recover kernel stack
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
exception_handler_entry ()
    at ./arch/riscv/kernel/entry.S:238
238         ret
(gdb) p $ra
$1 = (void (*)()) 0x502011d2 <exception_handler_entry+130>
(gdb)
```

意识到从helper回来后ra被改变，应该在helper返回后再把ret_from_exception存入ra寄存器。

TASK 4

1. 思路分析

1.1 set_timer的功能

看看set_timer是啥意思:

RISCV基础开发（十九）_| 南京养鸡二厂 (databusworld.cn)

这个函数的参数是要设置的比较器新值，所以需要将当前值先读取出来然后加上一个间隔值，如加上定时器工作频率值。查看头文件找到了`TIMER_INTERVAL`的宏，直接用√。

1.2 唤醒进程的时机?

二者皆是在调度器工作时唤醒进程，但仍有所区别：

- 在TASK 3中进程需通过`sys_yield`主动放弃cpu的使用权，此时调度器开始工作，顺便检查该被唤醒的进程；
- 引入了时钟中断，进程运行一段时间后会被自动挂起，调度器工作，也就是说在每次时钟中断到来时调度器会检查该被唤醒的进程。

2. Debug过程

一运行到wfi就寄？单步跟踪发现到例外入口会出错:

```
Breakpoint 2 at 0x5020110c: file ./arch/riscv/kernel/entry.S, line 276.
(gdb) n
201             asm volatile("wfi");
(gdb) s

Breakpoint 2, exception_handler_entry () 
    at ./arch/riscv/kernel/entry.S:276
276         SAVE_CONTEXT
(gdb) s
Remote connection closed
```

查看报错信息：

```
In:    uart@60000000
qemu: hardware error: sifive_test_write: write: addr=0x8 val=0x00000000504fffe0

CPU #0:
 pc       000000005020110c
 priv     0000000000000001
 mhartid  0000000000000000
 mstatus  00000000000001a0
 mip      00000000000000a0
 mie      0000000000000b3b
 mideleg  0000000000000222
 medeleg  000000000000b109
 mtvec    000000005ff92a00
 mepc     00000000502019b4
 mcause   8000000000000007
 sideleg  0000000000000000
 sedeleg  0000000000000000
 stvec    000000005020110c
 sepc     00000000502019b4
 scause   8000000000000005
 zero 0000000000000000 ra 00000000502019b0 sp 00000000504fffe0 gp 000000005f771d98
 tp 0000000000000000 t0 ffffffffffffffff t1 0000000050206a50 t2 000000005fff7420
 s0 0000000050500000 s1 0000000000000000 a0 0000000000000000 a1 0000000000000000
 a2 0000000000000000 a3 0000000000000000 a4 0000000000000000 a5 0000000000000000
```

看到tp为0，似乎是因为tp没有初始化，导致访问了不可访问的空间。内联汇编整一把：

```
// set tp to prepare for entering exception_handler_entry
uint64_t pid0_addr = (uint64_t)&pid0_pcb;
asm volatile(
    "nop\n"
    "mv tp, %0\n"
    :
    :"r"(pid0_addr)
);
```

再调了一些无关紧要的小bug就过了！

TASK 5

1. 思路分析

- ”创建的线程共享数据段和代码段“
  创建子线程的系统调用传入了子线程要执行的函数的入口地址，**把这个入口地址当作新的entry_point即可达到”共享代码段“的效果**，那么我们怎么知道数据段在哪？又如何共享？

查看反汇编代码：

```
...

0000000052000036 <test>:
    52000036:   1101                    addi    sp,sp,-32
    52000038:   ec06                    sd  ra,24(sp)
    5200003a:   e822                    sd  s0,16(sp)
    5200003c:   1000                    addi    s0,sp,32
    5200003e:   87aa                    mv  a5,a0
    52000040:   872e                    mv  a4,a1
    52000042:   fef42623            sw  a5,-20(s0)
    52000046:   87ba                    mv  a5,a4
    52000048:   fef42423            sw  a5,-24(s0)
    5200004c:   a8a5                    j   520000c4 <test+0x8e>
    5200004e:   00001797            auipc   a5,0x1
    52000052:   0b278793            addi    a5,a5,178 # 52001100 <_edata>
    52000056:   439c                    lw  a5,0(a5)
    52000058:   0017871b            addiw   a4,a5,1
    5200005c:   0007069b            sext.w  a3,a4
    52000060:   00001717            auipc   a4,0x1
    52000064:   0a070713            addi    a4,a4,160 # 52001100 <_edata>
```

从中可以得知其访问全局数据都是以当前的pc值为标杆，在此基础上作偏移，**相对寻址得到全局数据**，故只要在ra中存入正确的函数入口，在返回时会将pc置为入口地址，进而可以达到"共享数据段"的效果。

- ”创建的线程拥有独立的堆栈“

  只需为每个子线程重新alloc一个堆栈，并将之视作一个崭新的新“进程”（Linux中不对线程和进程作区分，也即以为着其可直接使用pcb结构体存储相关信息），仿照kernel对普通用户程序初始化的思路初始化其堆栈（只是需要注意修改入口和a0、a1等寄存器，达到传参的效果）。

2. 杂记

2.1 无效准备

先查询了linux的系统调用号（只找到了创建子进程的）：

```
313 /* kernel/fork.c */
314 #define __NR_set_tid_address 96
315 __SYSCALL(__NR_set_tid_address, sys_set_tid_address)
316 #define __NR_unshare 97
317 __SYSCALL(__NR_unshare, sys_unshare)
```

> linux系统调用相关源码导读：
>
> RISCV64 架构增加一个系统调用 (biscuitos.github.io)
>
> linux系统调用号所在头文件：
>
> include/uapi/asm-generic/unistd.h · master · mirrors / torvalds / linux · GitCode

发现咱先前写的似乎也没按照标准的来，那我就自由发挥了，随便分配一个系统调用号……把创建子进程的函数`create_thread`放在sche.c里。

2.2 测试用例的设计

初稿思路：

- 父、子线程分别打印自己的身份信息；
- 引入全局变量cur指示当前求和项，sum代表部分和，data是待求和数组（在main中初始化）；
- 子线程循环求和，父线程循环判断是否求和结束；
- 子线程求和结束后进入死循环，父线程打印求和结果后进入死循环。

在初稿代码上主要做了如下优化：

- 子线程间无法区分：传参指示子线程id（1、2）
- 起初子线程和主函数运行的输出放在同一行，快速切换时根本看不清……

  →利用参数id在主线程的打印位置基础上做偏移
- 下一次打印的时候使用空串“刷新”一下当前行（参考飞机测试用例的blank）