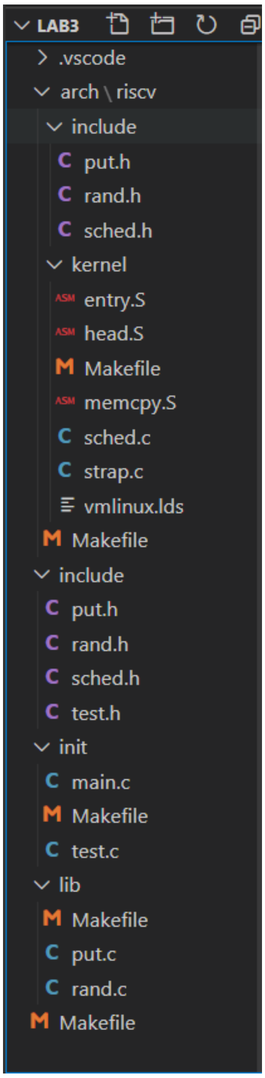
4.1

建立映射

```
cd 'C:\Users\administrater\Desktop\Archived Courses\OS\docker_vol\lab3'
docker run -it -v ${pwd}:/home/oslab/lab3 -u oslab -w /home/oslab 6014 /bin/bash
```

目录结构

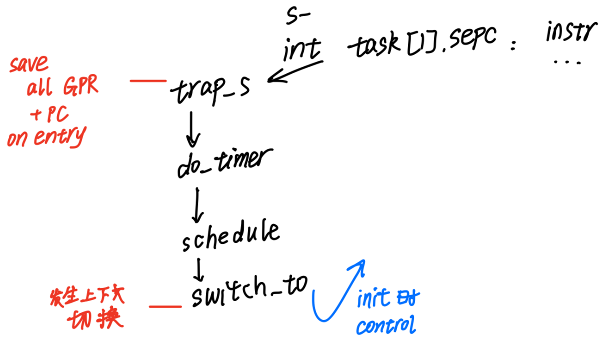


```
LAB3
  > .vscode
  v arch \ riscv
    v include
        put.h
        rand.h
        sched.h
    v kernel
        entry.S
        head.S
        Makefile
        memcpy.S
        sched.c
        strap.c
        vmlinux.lds
      Makefile
  v include
      put.h
      rand.h
      sched.h
      test.h
  v init
      main.c
      Makefile
      test.c
  v lib
      Makefile
      put.c
      rand.c
    Makefile
```

存在几个.h文件拷贝多份的现象，不是很好。我认为部分原因是代码层次划分的不是很清楚，比如sched.h里的函数在init文件夹中也需要被调用用于初始化。

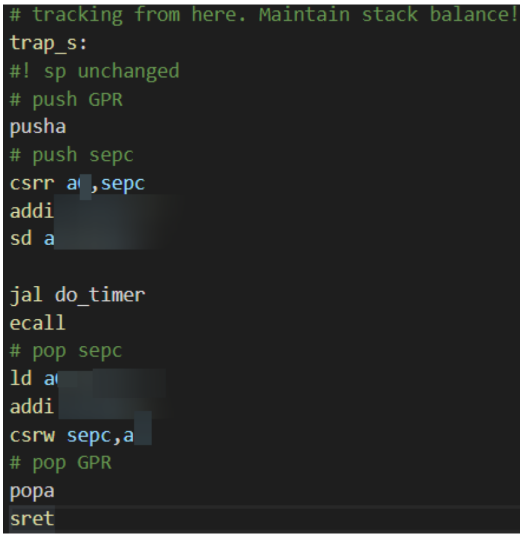
4.4 进程调度 调用分析

由于进程调度在实现上，是操作系统在*Supervisor Interrupt Handler*中"干了私活"，因此我们从*lab3*打下的基础，*entry.S*中的*trap_s*开始跟踪。最终的设计是这样的：



先考虑一般切换，初始化的之后再考虑

entry.S中的trap_s



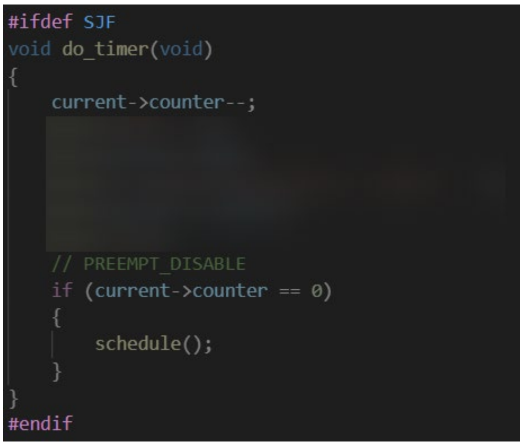
```
# tracking from here. Maintain stack balance!
trap_s:
#! sp unchanged
# push GPR
pusha
# push sepc
csrr a[redacted],sepc
addi [redacted]
sd a[redacted]

jal do_timer
ecall
# pop sepc
ld a[redacted]
addi [redacted]
csrw sepc,a[redacted]
# pop GPR
popa
sret
```

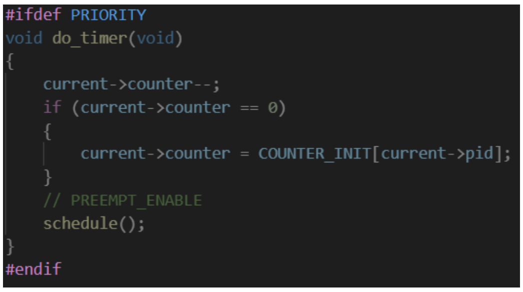
截图中 细节内容/调试输出 模糊处理，下同。

- 等到*do_timer*的层级调用返回，如果发生了进程切换，*sp*就变成下一个task的了，pop出来的也是下一个task的状态
- 没有内核栈，trap_s及其后调用的函数复用被trap进程的栈。不是很好，但简单。
- 接下来要注意保持栈平衡。尤其是pop时

sched.c中的do_timer



```
#ifdef SJF
void do_timer(void)
{
    current->counter--;
    [redacted]
    // PREEMPT_DISABLE
    if (current->counter == 0)
    {
        schedule();
    }
}
#endif
```



```
#ifdef PRIORITY
void do_timer(void)
{
    current->counter--;
    if (current->counter == 0)
    {
        current->counter = COUNTER_INIT[current->pid];
    }
    // PREEMPT_ENABLE
    schedule();
}
#endif
```

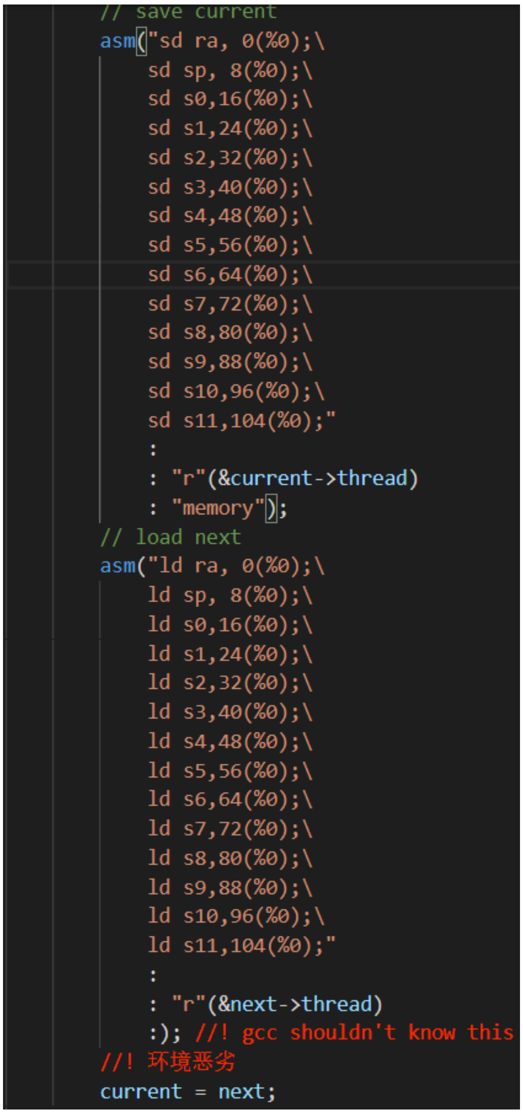
照实验指导 *4.4.3*翻译

sched.c中的schedule

照实验指导 *4.4.4*翻译，分离了重复的调试输出，应该适合阅读。

*sched.c*中的*switch_to*

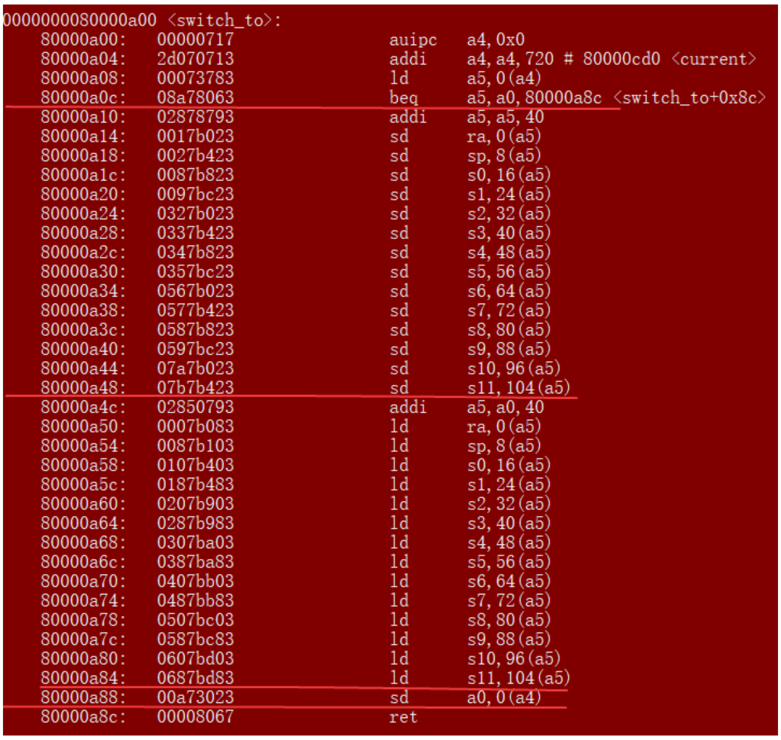
- 这里是当时实现的时候最头疼的地方。但最后做出来，挺naive的，路子挺野



```
// save current
asm("sd ra, 0(%0);\
    sd sp, 8(%0);\
    sd s0,16(%0);\
    sd s1,24(%0);\
    sd s2,32(%0);\
    sd s3,40(%0);\
    sd s4,48(%0);\
    sd s5,56(%0);\
    sd s6,64(%0);\
    sd s7,72(%0);\
    sd s8,80(%0);\
    sd s9,88(%0);\
    sd s10,96(%0);\
    sd s11,104(%0);"
    :
    : "r"(&current->thread)
    : "memory");
// load next
asm("ld ra, 0(%0);\
    ld sp, 8(%0);\
    ld s0,16(%0);\
    ld s1,24(%0);\
    ld s2,32(%0);\
    ld s3,40(%0);\
    ld s4,48(%0);\
    ld s5,56(%0);\
    ld s6,64(%0);\
    ld s7,72(%0);\
    ld s8,80(%0);\
    ld s9,88(%0);\
    ld s10,96(%0);\
    ld s11,104(%0);"
    :
    : "r"(&next->thread)
    :); //! gcc shouldn't know this
//! 环境恶劣
current = next;
```

从gcc的角度看，我们其实是破坏了C函数的调用规范， 把s*寄存器都摧毁了，还动了栈顶指针，非常恶劣。 但是我们是在进行系统编程。 除了帮我们算两个*task->thread*的偏移之外， gcc绝不能插手这一切。要小心gcc自作聪明。

dump最后的可执行文件*vmlinux*，确认符合预期。



```
000000080000a00 <switch_to>:
    80000a00:   00000717              auipc   a4, 0x0
    80000a04:   2d070713              addi    a4, a4, 720 # 80000cd0 <current>
    80000a08:   00073783              ld      a5, 0(a4)
    80000a0c:   08a78063              beq     a5, a0, 80000a8c <switch_to+0x8c>
    80000a10:   02878793              addi    a5, a5, 40
    80000a14:   0017b023              sd      ra, 0(a5)
    80000a18:   0027b423              sd      sp, 8(a5)
    80000a1c:   0087b823              sd      s0, 16(a5)
    80000a20:   0097bc23              sd      s1, 24(a5)
    80000a24:   0327b023              sd      s2, 32(a5)
    80000a28:   0337b423              sd      s3, 40(a5)
    80000a2c:   0347b823              sd      s4, 48(a5)
    80000a30:   0357bc23              sd      s5, 56(a5)
    80000a34:   0567b023              sd      s6, 64(a5)
    80000a38:   0577b423              sd      s7, 72(a5)
    80000a3c:   0587b823              sd      s8, 80(a5)
    80000a40:   0597bc23              sd      s9, 88(a5)
    80000a44:   07a7b023              sd      s10, 96(a5)
    80000a48:   07b7b423              sd      s11, 104(a5)
    80000a4c:   02850793              addi    a5, a0, 40
    80000a50:   0007b083              ld      ra, 0(a5)
    80000a54:   0087b103              ld      sp, 8(a5)
    80000a58:   0107b403              ld      s0, 16(a5)
    80000a5c:   0187b483              ld      s1, 24(a5)
    80000a60:   0207b903              ld      s2, 32(a5)
    80000a64:   0287b983              ld      s3, 40(a5)
    80000a68:   0307ba03              ld      s4, 48(a5)
    80000a6c:   0387ba83              ld      s5, 56(a5)
    80000a70:   0407bb03              ld      s6, 64(a5)
    80000a74:   0487bb83              ld      s7, 72(a5)
    80000a78:   0507bc03              ld      s8, 80(a5)
    80000a7c:   0587bc83              ld      s9, 88(a5)
    80000a80:   0607bd03              ld      s10, 96(a5)
    80000a84:   0687bd83              ld      s11, 104(a5)
    80000a88:   00a73023              sd      a0, 0(a4)
    80000a8c:   00008067              ret
```

初看lab3指导的时候，我其实有个困惑：

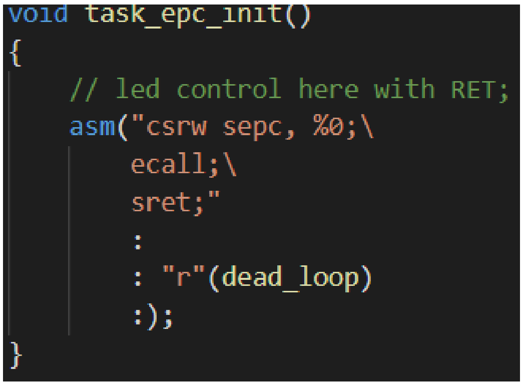
> 有点诡异，现在是在进行中断编程，被打断进程的所有寄存器都应该保存才对（包括 temporary）

后来自己做做，想明白，这其实 是保护C函数的调用规范，保callee-saved registers， 剩下的工作，由切换到下一个task的栈之后，函数逐级返回，退栈时完成。尤其是*trap_s*的最后部分，会恢复所有的通用寄存器和*epc*。

*sched.c*中的*task_init*

特别注意 栈不平 的问题。第一次切换到子进程时，由于子进程根本就没有被reschedule过，它的sp指在栈底，从*switch_to*逐级走正常的C函数返回流程，栈就会underflow.

我的解决方法：把子进程的thread.ra偷偷指到这个函数：

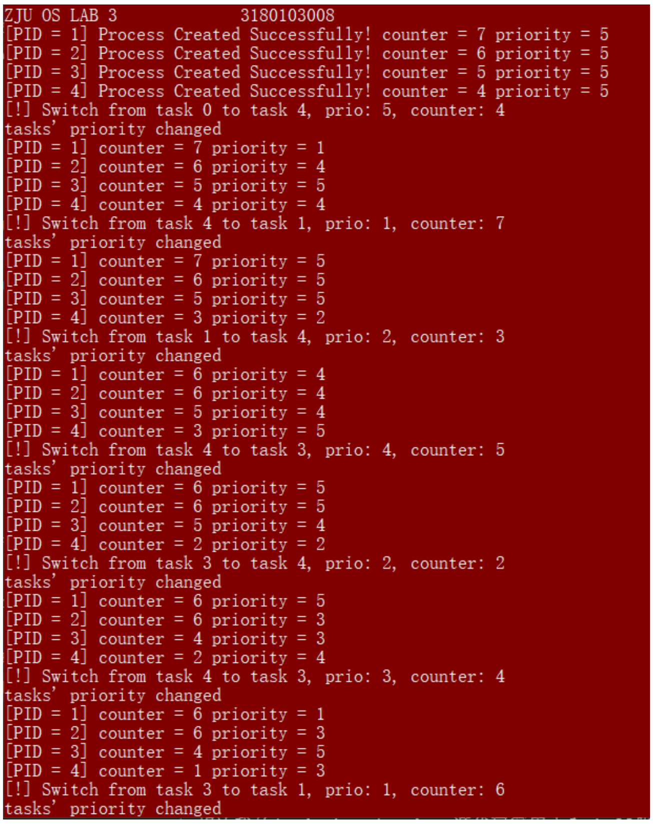


```
void task_epc_init()
{
    // led control here with RET;
    asm("csrw sepc, %0;\
        ecall;\
        sret;"
        :
        : "r"(dead_loop)
        :);
}
```

初始化*epc*为task将要执行的函数，然后直接接*trap_s*的*sret*返回，跳过中间层级。

4.5 测试

Priority 示例



```
ZJU OS LAB 3              3180103008
[PID = 1] Process Created Successfully! counter = 7 priority = 5
[PID = 2] Process Created Successfully! counter = 6 priority = 5
[PID = 3] Process Created Successfully! counter = 5 priority = 5
[PID = 4] Process Created Successfully! counter = 4 priority = 5
[!] Switch from task 0 to task 4, prio: 5, counter: 4
tasks' priority changed
[PID = 1] counter = 7 priority = 1
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 4 priority = 4
[!] Switch from task 4 to task 1, prio: 1, counter: 7
tasks' priority changed
[PID = 1] counter = 7 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 5
[PID = 4] counter = 3 priority = 2
[!] Switch from task 1 to task 4, prio: 2, counter: 3
tasks' priority changed
[PID = 1] counter = 6 priority = 4
[PID = 2] counter = 6 priority = 4
[PID = 3] counter = 5 priority = 4
[PID = 4] counter = 3 priority = 5
[!] Switch from task 4 to task 3, prio: 4, counter: 5
tasks' priority changed
[PID = 1] counter = 6 priority = 5
[PID = 2] counter = 6 priority = 5
[PID = 3] counter = 5 priority = 4
[PID = 4] counter = 2 priority = 2
[!] Switch from task 3 to task 4, prio: 2, counter: 2
tasks' priority changed
[PID = 1] counter = 6 priority = 5
[PID = 2] counter = 6 priority = 3
[PID = 3] counter = 4 priority = 3
[PID = 4] counter = 2 priority = 4
[!] Switch from task 4 to task 3, prio: 3, counter: 4
tasks' priority changed
[PID = 1] counter = 6 priority = 1
[PID = 2] counter = 6 priority = 3
[PID = 3] counter = 4 priority = 5
[PID = 4] counter = 1 priority = 3
[!] Switch from task 3 to task 1, prio: 1, counter: 6
tasks' priority changed
```

体会 & references

- LAST_TASK定义的不好
- 实验指导 task[0] counter被初始化为0，挺诡异的。变成了 先有鸡还是先有蛋 的问题
- 用于测试的task本身代码过于简单，无法从task本身的运行输出，区分获得控制(pc)的是哪个task。建议改进

汇编与C的互相访问比较困难。

- C访问寄存器 gcc内联汇编 https://blog.csdn.net/lwx62/article/details/82796364
- 汇编访问C结构体 定位困难
- 相比我的*switch_to*，Linux源代码采用宏和shell脚本预处理的方法结合，避免hard code，提高可维护性。https://blog.csdn.net/p0x1307/article/details/44492457
- 做子进程fork的时候，gcc由我的代码推断出一个memcpy，并因为没有实现而报了个错。后来，参考 https://elixir.bootlin.com/linux/v4.20/source 提供memcpy实现