4.1.1 建立映射

```
cd 'C:\Users\administrater\Desktop\Archived Courses\OS\docker_vol\lab5'
docker run -it -v ${pwd}:/home/oslab/lab5 -u oslab -w /home/oslab 6ca7 /bin/bash
```

4.1.2 代码结构

```
oslab@dca8ea2d2591:~/lab5$ ls -R
.:
Makefile  arch  hello.bin  hello.elf  include  init  lib

./arch:
riscv

./arch/riscv:
Makefile  kernel

./arch/riscv/kernel:
Makefile  entry.S  head.S  memcpy.S  memset.S  mtrap.c  sched.c  strap.c  vm.c  vmlinux.lds

./include:
asm_macro.h  mm_types.h  put.h  rand.h  sched.h  stddef.h  syscall.h  test.h  types.h  vm.h

./init:
Makefile  main.c  test.c

./lib:
Makefile  put.c  rand.c
```

较参考文件结构多出一些文件，主要是

- 参考用户代码仓库，定义的类型别名
- 内存帮助函数
- M mode 下的中断和异常处理
- 在多个汇编代码文件中使用的宏

虚拟内存映射

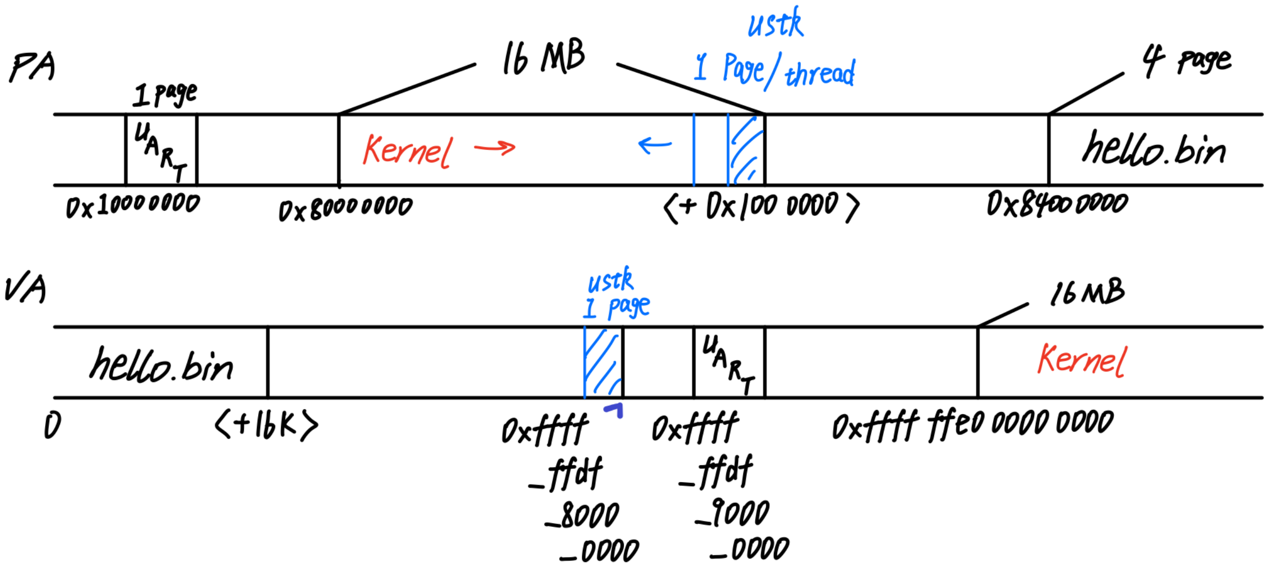
内核页表与lab4相同

```
Physical Address
-------------------------------------------
|     |UART|          |  Kernel 16MB  |
-------------------------------------------
      ^               ^
   0x10000000      0x80000000
      |               ├──────────────────────────────────────────┐
      |               |                        |
Virtual↓Address       ↓                        ↓
-----------------------------------------------------------------------
      |UART|          |  Kernel 16MB  |                  |  Kernel 16MB  |
-----------------------------------------------------------------------
      ^               ^                        ^
   0x10000000      0x80000000              0xffffffe000000000
```

- 0x80000000 处的恒等映射是必须的，否则会在写入 *satp* 启用分页的那条指令执行后，无法寻址到

下一条指令。
- 由于内核页表有虚拟地址到物理地址的恒等映射，因此用户页表的创建可以复用 *Bare mode* 下创建内核页表的例程 *create_mapping*

用户页表



- 0xffffffe000000000 处到内核的高地址映射，必须与内核页表相同，否则 内核 trap_handler 在 对换用户的*satp* 与内核的*satp* 后，会无法寻址到下一条指令。
- 用户态测试程序 *hello.bin* 的虚拟地址应该放在哪？指导未写明。并且遗憾的是，这段代码不是位置无关的。起初我让虚拟地址 从 0x400000 开始，内核 trap_handler 报 PF，debug 发现程序访问了非常低的虚拟地址，如下图所示。根据 *user*文件夹下的 *linker script*，*Location Counter*初始化为*0*. 遂将 VA 放在 *0*
- 给每个task各创建了一份页表，因此几个task的用户栈虚拟地址可以一样
- 用户栈的物理地址应该安置在哪？虽然地址空间很大，可并不是所有物理地址都可以访问。访问无效的物理地址时，报 *Store/AMO access fault*（不是 PF）. 碰了几次壁后，得到上图 PA 的安排.
- 用户页表其实未必要有到 UART 的映射，取决于如何实现 *sys_write*。见 处理 *ECALL_U* 一节。

```
0000000000000058 <putchar>:
  58:   00000797                auipc   a5,0x0
  5c:   7e078793    a5 = &tail  addi    a5,a5,2016 # 838 <tail>
  60:   0007a703                lw      a4,0(a5)
  64:   0017069b                addiw   a3,a4,1      a4 = tail
  68:   00d7a023                sw      a3,0(a5)
  6c:   45000793  a5 = buffer   li      a5,1104
  70:   00e787b3                add     a5,a5,a4
  74:   00a78023                sb      a0,0(a5)
  78:   00000513                li      a0,0
  7c:   00008067                ret        return 0
```

strap 的进入与退出

这里耦合很强，要设计得很仔细。敲定一处程序流程时，脑子里还得留意是否和其他地方有冲突；是否有其他地方也要修改；不然就又会陷入莫名其妙的调试麻烦之中，才能搞清发生了什么。在不断击打之下强行培养大局意识，很锻炼人。

主要有两个棘手的限制：

- 内核页表没有用户数据和栈的虚拟内存映射。因此切换为内核 *satp*后，只能使用内核栈指针 *ssp*，

且不能访问用户数据。

- *sstatus* 默认设定下，不仅用户不能访问内核页面，内核也不能够访问用户页面。因此 内核 trap_handler 继续使用用户 *satp* 也不是很好的主意

我的处理：

```
trap_s:
csrrw sp, sscratch, sp # use S.sp, save U.sp
pusha
set_pt_regs #! before any C function
jal swap_satp # use Kernel PT
...
secp_ret:
jal swap_satp # use User PT
# pop GPR from kernel stack
popa
# [only when ecall_u]
[get_pt_regs] #! after any C function
csrrw sp, sscratch, sp # use U.sp, save S.sp
sret
```

- 每个task在用户空间和内核空间都有栈区
- sscratch来直接保存内核栈指针 *ssp*
- 一进入内核 trap_handler，即刻对换 *ssp* 和 用户栈指针 *usp*；最后 *sret* 前再对换回来

4.2 处理 ECALL_U

M mode entry 准备工作

```
# delegate INSTR_PF, LOAD_PF, STORE_PF, ECALL_U to S mode
li s3, 0xB100
csrs medeleg, s3
```

两个难点：

- *pt_regs* 传参：定义了一个全局的 *pt_regs regs;* 汇编宏 *set_pt_regs* 把 trap 时的 `a*` 寄存器存入全局变量，供稍后 ecall 访问；ecall 返回时，在 *popa* 后额外 *get_pt_regs* 更新 `a*` 寄存器为系统调用返回值
    - 默认进入 trap_handler 时硬件清除全局中断使能，不会有嵌套中断，因此只需一个实例
    - 另一种可能是，复用 *pusha* 后栈的存储。但依赖于寄存器的 push 顺序，没有尝试

- *sys_write* 实现：需要将用户态传递的字符串打印到屏幕上。为此，内核应能对传来的 *buf* 解引用，以访问其内容。然而由于前面提到了 两条限制，这么一个想当然的过程都变得 tricky 了。
    - 在这个函数里，内核改用用户的*satp*；同时还得设置 `CSRs[sstatus].SUM` 位，以 *permit Supervisor User Memory access*，使得内核能读写用户页面
    - 内核需要能写 UART 口。这就是为什么用户页表也有到 UART 的映射，但标记为内核页面，以示保护
    - 另一种可能是，内核改用用户的*satp*，但只是把 buf 内容复制到内核自己的缓冲区；然后切换回内核*satp*，打印内核缓冲区内容。但内核缓冲区该多大又是另一个问题。

```
    swap_satp();
    for (size_t i = 0; i < count; i++)
```

```
    {
      *(uint64 *)UART_VA = (unsigned char)(*buf++);
    }
    swap_satp();
```

4.3 调度

进程状态段变动

```
    uint64 usp;
    uint64 sepc;
    uint64 ssp;
    uint64 satp;
```

switch_to 关键代码：

```
asm(
    "sd ra,  0 (%0);\
    sd sp, 120(%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);\
    csrr s0, sepc;\
    sd s0, 112(%0);\
    csrr s0, sscratch;\
    sd s0,  8 (%0);"
    :
    : "r"(&current->thread)
    : "memory");
```

```
asm(
    "ld s1,  8 (%0);\
    csrw sscratch, s1;\
    ld s1, 112(%0);\
    csrw sepc, s1;\
    ld ra,  0 (%0);\
    ld sp, 120(%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)
```

此时的 *satp* 应是内核*satp*，因此这里没做 *satp* 的切换。

两个用户进程 *satp* 的切换是在 `current = next;` 后的 *swap_satp()* 中隐式完成的：

```
  void swap_satp()
  {
    uint64 saved_satp = get_satp();
    if (saved_satp != current->mm.satp)
```

```
    {
        set_satp(current->mm.satp);
        current->mm.satp = saved_satp;
    }
}
```

进程初始化

```
task[i]->thread.ssp = task[i] + TASK_SIZE;
task[i]->thread.usp = 0xffffffdf80000000;
task[i]->thread.ra = (uint64)task_pre_run;
task[i]->thread.sepc = ucode_va;
uint64 root_ppn = u_paging_init(task[i]) >> 12;
task[i]->mm.satp = ((uint64)8 << 60) | root_ppn;
```

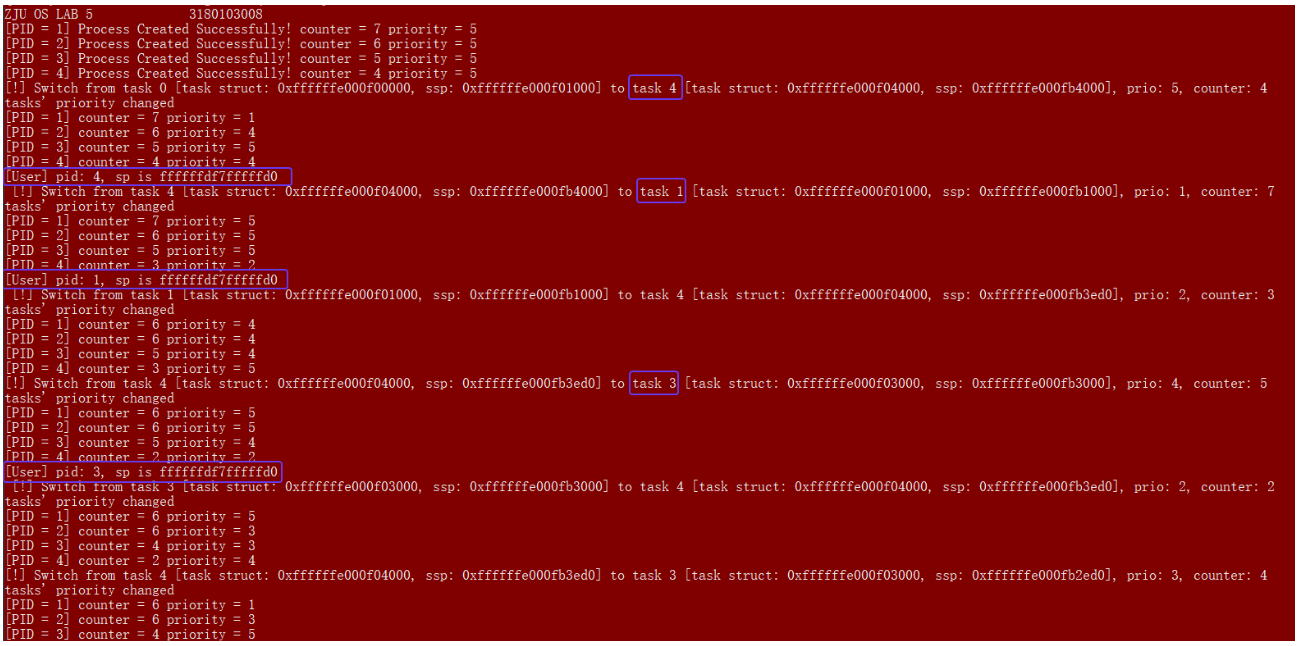
```
/* control led here after `ret` in S-mode task

A U-mode task is to run. This piece of code is for preparation:
* prep PT for U-mode task
* ecall from S-mode informs M-mode the completion of STI processing;
* CSRs[sstatus].SPP = 0b0, CSRs[sstatus].SPIE = 1
* control passed to U-mode task after `sret`
*/
void task_pre_run()
{
    swap_satp();
    asm("csrr s11, sepc;\
        ecall;\
        csrw sepc, s11;\
        csrrw sp, sscratch, sp;\
        csrc sstatus, %0;\
        csrs sstatus, %1;\
        sret;"
        :
        : "r"(0b100000000), "r"(0b10000)
        :);
}
```

refactor

- printf: 从代码仓库的用户源码里借鉴了 *printf*的实现，改造得到 *my_printk*
- error_handler & shutdown
- 部分函数使用了函数属性声明。It'll be dependent on GNU-GCC
  - *noreturn* 简化部分函数的反汇编
  - *format* 编译器帮助检查 *printf*参数

printf 有可能自己写一个，但得基于参数全部经由压栈传递的假设。根据之前 debug 过程的观察，not the case. 可能存在的寄存器传参使得"爬调用栈"的实现不robust. 我想这也是为什么存在 *va_list*这些编译器 built_in 的可变参数支持的缘故。

4.5 完成截图



```
ZJU OS LAB 5             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 [task struct: 0xffffffe000f00000, ssp: 0xffffffe000f01000] to task 4 [task struct: 0xffffffe000f04000, ssp: 0xffffffe000fb4000], 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
[User] pid: 4, sp is ffffffdf7ffffd0
 [!] Switch from task 4 [task struct: 0xffffffe000f04000, ssp: 0xffffffe000fb4000] to task 1 [task struct: 0xffffffe000f01000, ssp: 0xffffffe000fb1000], 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
[User] pid: 1, sp is ffffffdf7ffffd0
 [!] Switch from task 1 [task struct: 0xffffffe000f01000, ssp: 0xffffffe000fb1000] to task 4 [task struct: 0xffffffe000f04000, ssp: 0xffffffe000fb3ed0], 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 [task struct: 0xffffffe000f04000, ssp: 0xffffffe000fb3ed0] to task 3 [task struct: 0xffffffe000f03000, ssp: 0xffffffe000fb3000], 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
[User] pid: 3, sp is ffffffdf7ffffd0
 [!] Switch from task 3 [task struct: 0xffffffe000f03000, ssp: 0xffffffe000fb3000] to task 4 [task struct: 0xffffffe000f04000, ssp: 0xffffffe000fb3ed0], 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 [task struct: 0xffffffe000f04000, ssp: 0xffffffe000fb3ed0] to task 3 [task struct: 0xffffffe000f03000, ssp: 0xffffffe000fb2ed0], 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
```

实验指导的实例输出中，进程调度似乎没有使用调度算法，因此略有不同。但可以看出，用户态程序的 pid 获取 和 字符串打印 是没有问题的，证明完成了实验目的。

体会 & reference

Linux 内核通过把宏运用到极致，减少了代码冗余，提高了代码的组织水平； 但需要对编译工具的深刻理解，估计也得很长时间的积累， 作为玩具 OS 的实现难以学习

- https://elixir.bootlin.com/linux/v4.20/source
- https://stackoverflow.com/questions/10060168/is-asmlinkage-required-for-a-c-function-to-be-called-from-assembly

Clipboard for debug

```
riscv64-unknown-linux-gnu-gdb vmlinux
target remote localhost:1234

b *0x400034
b ecall_u if regs.a7 == 64

riscv64-unknown-elf-objdump -S --disassemble=my_printk vmlinux
```