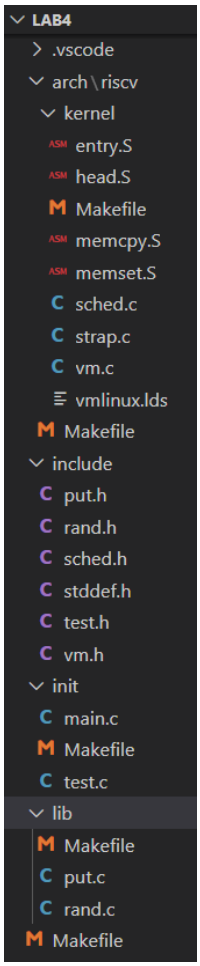
4.1.1 建立映射

```
cd 'C:\Users\administrater\Desktop\Archived Courses\OS\docker_vol\lab4'
docker run -it -v ${pwd}:/home/oslab/lab4 -u oslab -w /home/oslab 6ca7 /bin/bash
```

4.1.2 代码结构



相比实验指导的目录结构稍作简化：只留一个 include 目录在根下，所有 .h 文件都放在其中。否则 .h 文件存在数据冗余，修改后难以控制一致性。目测这样的简化能够应对后续实验的代码复杂度。

stddef.h 定义有类型别名。

4.2

为页表空间分配物理页

可分配空间始址用一个静态的指针 `char *pgtbl_memory;` 指示。运用 linker script 提供 symbol _end_lma ，指示 _end 的物理地址，初始化 pgtbl_memory 指针。一个小函数 `void *get_free_page()` 分配 pgtbl_memory 向后 1 页的内存，并更新 pgtbl_memory.

create_mapping

- Sv39 中 `39 = 3*9+12`

写 *create_1pg_mapping* 函数为一页空间建立页表映射， *create_mapping* 就简单地一页一页调用 *create_1pg_mapping* 函数。

```
自 va 解析三级页表的虚页号 uint64 vpn[3]
对第一级和第二级页表：
  用页表基址 pgtbl 和虚页号 vpn[i] 定位到页表项
  如果 PTE_VALID:
    用 PTE 的 PPN 信息更新 pgtbl，指向下一级页表的页表基址
  否则：
    allocate physical page to be pointed by this pte
    更新 pgtbl，指向下一级页表的页表基址
对第三级页表 (leaf level)：
  定位到页表项
  用 pa 解析出的 PPN 和 参数 perm 写 PTE，不要忘记置上 PTE_VALID
```

paging_init

先为根页表分配空间，通过 `get_free_page()`；并初始化清零，自己写了一个 naive 的 `void *memset(void *, int, size_t)`。然后按 4.5 要求进行映射调用。例如：

```
// kernel: to highest va; to identical va
uint64 pa = 0x80000000;
uint64 va1 = 0xffffffe000000000, va2 = pa;
uint64 sz = 16 << 20;
...
// text; symbol *_start 4k aligned
sz = &rodata_start - &text_start;
create_mapping(root, va1, pa, sz,
            PERM_READ | PERM_EXECUTE);
create_mapping(root, va2, pa, sz,
            PERM_READ | PERM_EXECUTE);
```

4.3

MMU: from Bare to Sv39

```
S_start:
jal paging_init # init page table
# Mode: Sv39
li s6, 8
slli s6, s6, 60
# Root pgtbl located at _end
la s7, _end
srli s7, s7, 12
# MMU: Sv39 Paging
or s8, s6, s7
csrw satp, s8
sfence.vma
# PC is still in PA. In order to switch to VA,
# we calc some known pos in VA and jr there
la s9, next
#TODO do not hard code
```

```
li a7, 0xffffffe000000000 - 0x0000000080000000
add s9, s9, a7
# jump to next
jr s9
next:
# PC is in VA now.
```

M模式下异常处理初始化

和 M Mode 的栈顶指针一起初始化好

```
# M mode entry
_start:
# MMU: Bare
csrw satp, zero
...
# M mode stack
la sp, stack_top
csrw mscratch, sp
```

4.4 修改 sched.c

`task_init()` 中，为各进程分配 task_struct ， 始址改成虚拟地址

```
// init current
current = (void *)0xffffffe000fc0000;
```

在进程调度时打印 task_struct 地址

```
void print_ts(struct task_struct *ts)
{
    puts(" [task struct: 0x");
    puth((uint64)ts);
    puts(", sp: 0x");
    puth((uint64)(ts->thread.sp));
    puts("]");
}
```

4.5 保护

验证这些属性是否成功被保护

S 模式的异常处理函数，把同步异常传到下面这个 C 函数进行打印

```
void error_handler(uint64 cause, uint64 epc)
{
    const char *msg;
    int oops = 0;
```

```
    switch (cause)
    {
    case 12: // INSTR_PF
        msg = "INSTR_PF";
        break;
    case 13: // LOAD_PF
        msg = "LOAD_PF";
        break;
    case 15: // STORE_PF
        msg = "STORE_PF";
        break;
    default:
        msg = "!!! UNEXPECTED ERROR !!!";
        oops = 1;
        break;
    }
    puts(msg);
    puts("    epc = 0x");
    puth(epc);
    puts("\n");
    while (oops)
        ;
}
```

对于同步异常，返回到下一条指令继续执行。 在系统初始化完毕，进入 `os_test` 函数后，试着向 text 段写数据，如下。预期触发 STORE_PF，操作无效，系统继续正常运行。

```
*((uint64 *)puts) = 0;
```

4.6 输出

```
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
ZJU OS LAB 4              3180103008
trying to write data to text section:
STORE_PF    epc = 0xffffffe0000003f0
STORE_PF    epc = 0xffffffe0000003f8
STORE_PF    epc = 0xffffffe000000400
STORE_PF    epc = 0xffffffe000000408
end of testing
[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: 0xffffffe000fc0000, sp: 0xffffffe000fc1000] to task 4 [task struct: 0xffffffe000fc4000, sp: 0xffffffe000
fc5000], 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 [task struct: 0xffffffe000fc4000, sp: 0xffffffe000fc5000] to task 1 [task struct: 0xffffffe000fc1000, sp: 0xffffffe000
fc2000], 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 [task struct: 0xffffffe000fc1000, sp: 0xffffffe000fc2000] to task 4 [task struct: 0xffffffe000fc4000, sp: 0xffffffe000
```

- task_struct 和 sp 确实都在高地址空间；具体值和示例不一致，因初始化 current 不同
- 打印了 4 次 STORE_PF. 检查了反汇编，

```
*((uint64 *)puts) = 0;
```

被翻译成 sh 指令，一次 16 位，因此出现 4 次 STORE

体会

主要的 bug 都来自访存时的地址搞不清楚。

调试

- 根据 *linker script*，*location counter* 以及各 section 的 VMA 使用的都是 0xffffffe000000000 开始的高地址空间；但装载的位置即 LMA 却在 0x0000000080000000 开始的 16 M 空间。
- lma != vma; 在刚刚载入， MMU 还在 Bare Mode 的时候，GDB 甚至不认得 PC 指向的是代码位置，给调试带来了很大困难
    - 速查了一下 gdb 手册 (auto overlay debug)，如果要在这种情况下让 GDB 自动识别，还得写个特殊结构的 C struct ，做不来
    - 用地址来指定断点 `b *0x800000d4`
    - examine memory as instructions: x/7i 0x80000000

寻址方式

伪指令 la 是 PC-relative addressing. 具体而言，通过 *auipc* 后 *addi* 指令向寄存器装载地址。地址偏移在 link 时解析：把散落于各个 .o 文件的各 sections 收集，并按照*linker script* 排好后，得到*symbol table*. *symbol table* 中 *symbol* 的地址都是 *VMA*（高地址）. 对于每一个重定位项，查此处的*location counter* （也是 *VMA* 高地址），计算与*symbol*的差。

```
ffffffe0000041c0 g       .bss   0000000000000000 bss_end
ffffffe000003010 g     O .bss   0000000000000008 current
ffffffe000000438 g     F .text  0000000000000068 puti
ffffffe000003000 g       .bss   0000000000000000 bss_start
ffffffe000003fc0 g     O .bss   0000000000000200 task
ffffffe000007000 g       .bss   0000000000000000 stack_top
ffffffe000000718 g     F .text  0000000000000018 task_epc_init
0000000080007000 g       *ABS*  0000000000000000 _end_lma
ffffffe000000f50 g       .text  0000000000000000 memset
ffffffe000000b74 g     F .text  000000000000001c get_free_page
ffffffe000000398 g     F .text  000000000000002c start_kernel
ffffffe000000000 g       .text  0000000000000000 text_start
ffffffe000002000 g       .data  0000000000000000 data_start
ffffffe000000688 g     F .text  0000000000000090 error_handler
ffffffe000000260 g       .text  0000000000000000 trap_s
ffffffe000000910 g     F .text  0000000000000050 print_ts
ffffffe000000d04 g     F .text  0000000000000128 paging_init
ffffffe000007000 g       .bss   0000000000000000 _end
ffffffe000003000 g     O .bss   0000000000000004 initialize
ffffffe000000b90 g     F .text  00000000000000ec create_lpg_mapping
```

这有助于位置无关代码*PIC code*，但也会产生初看 weird 的现象。

- 比如 M mode 初始化时，尽管我还在 0x80000000 的物理地址空间，我仍然可以用 *la* 指令装符号表上看过去是 0xfff... 的 *stack_top*，并且仍然得到 *stack_top* 的物理地址 0x800...
- C 语言 *vm.h* 中 *extern const char _end*，意思是让 *linker* 确定 *_end* 的地址。*linker* 会使用 VMA, 也就是说 *&_end == 0xfff...* 而不能得到物理地址 0x800... 这不符合 paging_init 的要求。可以想象当时对此一无所知的我吃了多大苦头 QAQ

reference

https://elixir.bootlin.com/linux/v4.20/source

Clipboard for debug

```
docker exec -it -u oslab -w /home/oslab 248a /bin/bash
```

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

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