ChCore 实验 3

班级：F2103003 姓名：黄伊新 学号：521030910070

1. **练习 1：在 kernel/object/cap_group.c 中完善 sys_create_cap_group、create_root_cap_group 函数**

   在 create_root_cap_group 函数中，主要是需要喂 root 分配 cap group 和 vmspace，这两个都是通过 obj_alloc 函数来进行，调用的参数都为已经设置好的宏，代码如下。

```
/* LAB 3 TODO BEGIN */
cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(struct cap_group));
/* LAB 3 TODO END */
BUG_ON(!cap_group);

/* LAB 3 TODO BEGIN */
/* initialize cap group, use ROOT_CAP_GROUP_BADGE */
cap_group_init(cap_group, BASE_OBJECT_NUM, ROOT_CAP_GROUP_BADGE);
/* LAB 3 TODO END */
slot_id = cap_alloc(cap_group, cap_group);

BUG_ON(slot_id != CAP_GROUP_OBJ_ID);

/* LAB 3 TODO BEGIN */
vmspace = obj_alloc(TYPE_VMSPACE, sizeof(struct vmspace));
/* LAB 3 TODO END */
BUG_ON(!vmspace);

/* fixed PCID 1 for root process, PCID 0 is not used. */
vmspace_init(vmspace, ROOT_PROCESS_PCID);

/* LAB 3 TODO BEGIN */
slot_id = cap_alloc(cap_group, vmspace);
/* LAB 3 TODO END */

```

   而在 sys_create_cap_group 函数中，我们需要完成的则和上面这个函数中的类似，但是调用的参数需要相应的换成 args 中的值。

2. **练习 2：在 kernel/object/thread.c 中完成 create_root_thread 函数，将用户程序 ELF 加载到刚刚创建的进程地址空间中**

   在本练习中，我们主要是需要进行 ELF 文件的加载，offset 等 program header 的加载主要可以参考提供的 flags 的加载方式进行。

   而对于 pmo 的申请，由于在对内存进行分配时需要按照页为粒度进行，故我们首先需要获得能够容纳这一段的所需的内存的大小。从 programd

header 中加载得到的 vaddr 对应的是 ELF 文件指定的这一部分数据需要分配在的虚拟地址，memsz 则是这个文件加载到内存后所需的内存空间大小，使用 ROUND_UP 和 ROUND_DOWN 即可得到囊括这一段空间所需要的页的个数。

然后是把 ELF 文件加载到内存空间中。首先使用 memset 将整片对应区域置零，以防 filesz 和 memsz 不等，其他地方需要 padding 为 0。然后使用 memcpy，将数据从 ELF 文件对应的地方加载到 pmo 开始的地方。特别的是此处从 program header 中得到的 offset 是从 program header 之后的 offset，所以需要在 binary_procmgr_bin_start 指定的地方的基础上加上之前的偏移量，才是真正的需要加载的 ELF 段的位置。

```
for (int i = 0; i < meta.phnum; i++) {
    unsigned int flags;
    unsigned long offset, vaddr, filesz, memsz;

    memcpy(data,
           (void *)((unsigned long)&binary_procmgr_bin_start
                    + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                    + PHDR_FLAGS_OFF),
           sizeof(data));
    flags = (unsigned int)le32_to_cpu(*(u32 *)data);

    /* LAB 3 TODO BEGIN */
    /* Get offset, vaddr, filesz, memsz from image*/
    memcpy(data,
      (void *)((unsigned long)&binary_procmgr_bin_start
        + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
        + PHDR_OFFSET_OFF),
      sizeof(data));
    offset = (unsigned long)le64_to_cpu(*(u64 *)data);

    memcpy(data,
      (void *)((unsigned long)&binary_procmgr_bin_start
        + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
        + PHDR_VADDR_OFF),
      sizeof(data));
    vaddr = (unsigned long)le64_to_cpu(*(u64 *)data);

    memcpy(data,
      (void *)((unsigned long)&binary_procmgr_bin_start
        + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
        + PHDR_FILESZ_OFF),
      sizeof(data));
    filesz = (unsigned long)le64_to_cpu(*(u64 *)data);

    memcpy(data,
      (void *)((unsigned long)&binary_procmgr_bin_start
        + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
        + PHDR_MEMSZ_OFF),
```

```
      sizeof(data));
    memsz = (unsigned long)le64_to_cpu(*(u64 *)data);

    if (memsz == 0) continue;
    /* LAB 3 TODO END */

    struct pmobject *segment_pmo;
    /* LAB 3 TODO BEGIN */
    // ret = create_pmo(ROUND_UP(memsz + vaddr, PAGE_SIZE) -
    ROUND_DOWN(vaddr, PAGE_SIZE), PMO_DATA, root_cap_group, 0, &
    segment_pmo);
    ret = create_pmo(ROUND_UP(memsz + vaddr, PAGE_SIZE) -
    ROUND_DOWN(vaddr, PAGE_SIZE), PMO_DATA, root_cap_group, 0, &
    segment_pmo);
    /* LAB 3 TODO END */

    BUG_ON(ret < 0);

    /* LAB 3 TODO BEGIN */
    /* Copy elf file contents into memory*/
    memset((void *)phys_to_virt(segment_pmo->start), 0,
    segment_pmo->size);
    memcpy((void *)(phys_to_virt(segment_pmo->start) + (vaddr &
    OFFSET_MASK)), (void *)((unsigned long)&
    binary_procmgr_bin_start + ROOT_PHDR_OFF + meta.phnum *
    ROOT_PHENT_SIZE + offset), filesz);
    /* LAB 3 TODO END */

    unsigned vmr_flags = 0;
    /* LAB 3 TODO BEGIN */
    /* Set flags*/
    vmr_flags = (((flags) & PHDR_FLAGS_X ? VMR_EXEC : 0) | ((flags
    ) & PHDR_FLAGS_W ? VMR_WRITE : 0) | ((flags) & PHDR_FLAGS_R ?
    VMR_READ : 0));
    /* LAB 3 TODO END */

    ret = vmspace_map_range(init_vmspace,
                            vaddr,
                            segment_pmo->size,
                            vmr_flags,
                            segment_pmo);
    BUG_ON(ret < 0);

}
```

3. **练习 3: 在 kernel/arch/aarch64/sched/context.c 中完成 init_thread_ctx 函数，完成线程上下文的初始化**

    该练习主要是对 thread 中的 thread_ctx 赋上相应的值即可。

```
thread->thread_ctx->ec.reg[SP_EL0] = stack;
```

```
thread->thread_ctx->ec.reg[ELR_EL1] = func;
thread->thread_ctx->ec.reg[SPSR_EL1] = SPSR_EL1_EL0t;

```

4. **思考题 4: 思考内核从完成必要的初始化到第一次切换到用户态程序的过程是怎么样的？尝试描述一下调用关系**

   从初始化到第一次切换用户态程序的过程都可以在/kernel/arch/aarch64/main.c 中看见。内核在完成一系列的初始化，包括 lock 初始化，uart 初始化，cpu info 初始化，mm 初始化，exception vector 初始化，pmu 初始化，scheduler 初始化等，之后进行用户态线程的准备：首先调用 create_root_thread 函数，来创建第一个线程；然后又调用了 sched 函数，此时会选择刚刚创建的那个线程；最后通过 eret_to_thread 和 switch_context 来完成上下文的切换，进入刚刚创建的用户态线程中。这几个函数都由 sched.h 声明（具体代码未给出）。eret_to_thread 最终调用到了 irq_entry.S 中的函数，负责为其完成上下文的切换及准备，最终 eret 切换到用户态程序。

5. **练习 5: 按照前文所述的表格填写 kernel/arch/aarch64/irq/irq_entry.S 中的异常向量表，并且增加对应的函数跳转操作**

   关于异常向量表的填写，只需要识别出第几项对应的是那个跳转的标签即可，把他作为参数传入 exception_entry 函数来进行处理。而对于函数跳转，使用 bl 语句进行相关跳转即可。而在调用完 handle_entry_c 后，还需要将返回值存储回 ELR_EL1，也就是将 x0 中的值通过 msr 写入该特殊寄存器。

6. **填写 kernel/arch/aarch64/irq/irq_entry.S 中的 exception_enter 与 exception_exit，实现上下文保存的功能，以及 switch_to_cpu_stack 内核栈切换函数**

   首先是上下文的保存，此处也就是寄存器的保存，主要是通过 stp 来保存，对于没有成对的 x30 则和特殊寄存器一起保存，相应的可以进行后续的读取和恢复。

   内核栈的切换需要考察 smp.h 中的数据结构，可以发现提供了宏 OFFSET_LOCAL _CPU_STACK 来获取相对于该 struct 起始位置的偏移量，那么就可以得到存储栈的信息的地址值，用 ldr 即可读取。

7. **思考 7: 尝试描述 printf 如何调用到 chcore_stdout_write 函数**

   首先 printf 调用 vfprintf 函数，其文件描述符参数为 stdout，根据 stdout.c 中可以发现 stdout 中的写操作被定义为 ___stdout_write，通过再 vfpringf 中调用 f->write 跳转到了 ___stdout_write。根据 ___stdout_write 函数的代码，他使用相同的参数去调用了 ___stdio_write 函数。该函数调用 syscall，传入参数为 SYS_writev, f->fd, iov 和 iovcnt。而 syscall 在 syscall.h 中被定义为宏，等价于 ___syscall_ret(___syscall(___VA_ARGS___))，后续经过宏的转换，这个函数等价于在调用 ___syscall3 的函数，然后调用 ___syscall6，

在此处由传入的参数 SYS_writev 得到需要跳转到 chcore_writev，该函数在 fd.c 中，调用了 chcore_write，该函数则是调用了 stdout 对应的 fd_op 的 write 操作，该操作被定义为调用 chcore_stdout_write 函数，从而调用到 chcore_stdout_write 函数。

8. **在 put 函数中添加一行以完成系统调用，目标调用函数为内核中的 sys_putstr**

   通过 chcore_syscallx 来进行系统调用，由于需要传入两个参数，故为 syscall2，而我们需要调用的函数为 sys_putstr，使用宏来指向它，作为第一个参数传入即可。

```
chcore_syscall2(CHCORE_SYS_putstr, (long int)buffer, (long int)
    size);

```

9. **练习 9: 尝试编写一个简单的用户程序，其作用至少包括打印 Hello ChCore!**
   函数的编写比较简单，只需要调用 stdio.h 头文件，并使用其中声明的 printf 进行输出即可。编译则需要指定编译文件，使用的指令为 build/chcore-libc/bin/musl-gcc ramdisk/main.c -o ramdisk/hello_chcore.bin。