ChCoreLab3

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

利用好上下文中出现的函数和README中的提示完成即可，如：

```
new_cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(struct cap_group));
```

```
/* initialize cap group */
cap_group_init(new_cap_group, BASE_OBJECT_NUM, args.badge);
```

```
vmspace = obj_alloc(TYPE_VMSPACE, sizeof(struct vmspace));
```

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。

首先，要 `Get offset, vaddr, filesz, memsz from image`，主要参考上面flags的获取方式，并理解offset与各个信息量的对应关系即可。

注意，offset打印出来发现它会出现0的情况，这决定了elf文件的读取时，偏移量需要避免与前面各个信息量（如filesz，memsz等）冲突，所以额外加这样的偏移 `meta.phnum * ROOT_PHENT_SIZE`：

```
memcpy((void *)phys_to_virt(segment_pmo->start) + (vaddr & OFFSET_MASK),
       (void *)((unsigned long)&binary_procmgr_bin_start + ROOT_PHDR_OFF
+   offset + meta.phnum * ROOT_PHENT_SIZE), filesz);
```

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。

可以根据下面的 `arch_set_thread_stack` 等函数推断使用方法。

```
/* SP_EL0, ELR_EL1, SPSR_EL1*/
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_USER;
```

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。

在配置页表并开启 MMU 后，首先调用 `arch_interrupt_init` 初始化异常向量表，然后调用 `create_root_thread` 创建第一个进程，在其中调用 `create_root_cap_group` 创建 `root_cap_group`，最后将 ELF 用户程序加载到地址空间中，完成进程创建。在完成必要的初始化后，内核首先调用了 `sched()` 来调度初始化的第一个进程。然后通过`switch_context()` 进行进程上下文的切换。`eret_to_thread()`将 被选择进程的上下文地址写入`sp`寄存器后，调用了 `exception_exit` 函数，最后 `exception_exit` 调用 eret 返回用户态，从而完成了从内核态向用

户态的第一次切换。

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。

结合文件中几个异常的Label进行填写即可：

```
EXPORT(el1_vector)

    /* LAB 3 TODO BEGIN */
    // Set the vector table base address

    exception_entry sync_el1t
    exception_entry irq_el1t
    exception_entry fiq_el1t
    exception_entry error_el1t

    exception_entry sync_el1h
    exception_entry irq_el1h
    exception_entry fiq_el1h
    exception_entry error_el1h

    exception_entry sync_el0_64
    exception_entry irq_el0_64
    exception_entry fiq_el0_64
    exception_entry error_el0_64

    exception_entry sync_el0_32
    exception_entry irq_el0_32
    exception_entry fiq_el0_32
    exception_entry error_el0_32

    /* LAB 3 TODO END */
```

练习 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。

`stp` 是store pair的意思，两个寄存器一起被存入内存。将各寄存器保存至内存中即可。`exception_exit` 为 `exception_enter` 的相反操作。

```
.macro switch_to_cpu_stack
    mrs     x24, TPIDR_EL1
    /* LAB 3 TODO BEGIN */
    add x24, x24, #OFFSET_LOCAL_CPU_STACK
    /* LAB 3 TODO END */
    ldr x24, [x24]
    mov sp, x24
.endm
```

思考 7: 尝试描述 `printf` 如何调用到 `chcore_stdout_write` 函数。

在调用 `printf` 函数时，会调用 `vfprintf` 函数，并将 `stdout` 作为文件描述符参数传递进去。`stdout` 对应的 `write` 操作被定义为 __stdout_write。在 `__stdio_write` 函数中会进行系统调用 `syscall`，并将 `f->fd` 作为参数传入，即 `stdout->fd`，也就是 `1`。我们可以在 `syscall_dispatcher.c` 文件中看到，`` `fd_dic[fd1]->fd_op = &stdout_ops; ``，而在 `stdout_ops` 中将 `write` 操作定义为 `chcore_stdout_write`。

```
int printf(const char *restrict fmt, ...)
{
  int ret;
  va_list ap;
  va_start(ap, fmt);
  ret = vfprintf(stdout, fmt, ap);
  va_end(ap);
  return ret;
}
```

`printf` 函数调用了 `vfprintf`，其中文件描述符参数为 `stdout`。这说明在 `vfprintf` 中将使用 `stdout` 的某些操作函数。

练习 8: 在其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`。使用 `chcore_syscallx` 函数进行系统调用。

使用 `chcore_syscall2`， 注意buffer需要进行类型转换。

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以下字符(测试将以此为得分点)。

在`build/chcore-libc/bin`中创建 `hello_chcore.c` 文件，然后在其中 `#include <stdio.h>`，在main函数中调用prinf打印字符串。

运行 `./musl-gcc hello_chcore.c -o hello_chcore.bin`