Report of Lab3

王焕宇 522030910212

练习题 1

在 kernel/object/cap_group.c 中完善 sys_create_cap_group、create_root_cap_group 函数。在完成填写之后，你可以通过 Cap create pretest 测试点。

1. 首先使用 obj_alloc 分配 cap_group, 后使用 cap_group_init_user 从用户初始化，再通过 obj_alloc 分配vmspace
2. 同理，但使用 cap_group_init_common 初始化cap group

练习题 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。

1. 对于offset, vaddr, filesz, memsz, 使用 memcpy 进行分配，如下：

```
memcpy(data,
        (void *)((unsigned long)&binary_procmgr_bin_start
                + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                + PHDR_OFFSET_OFF),
        sizeof(data));
offset = (unsigned int)le64_to_cpu(*(u64 *)data);
```

2. 使用 create_pmo 分配内存，并将elf文件存入内存，如下：

```
kfree((void *)phys_to_virt(segment_pmo->start));
segment_pmo->start = virt_to_phys(((unsigned long)&binary_procmgr_bin_start) + offset);
segment_pmo->size = ROUND_UP(memsz, PAGE_SIZE);
```

3. flags的设定如下：

```
if (flags & PHDR_FLAGS_R) vmr_flags |= VMR_READ;
if (flags & PHDR_FLAGS_W) vmr_flags |= VMR_WRITE;
if (flags & PHDR_FLAGS_X) vmr_flags |= VMR_EXEC;
```

练习题 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。

```
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;
```

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。

根据代码导读中的 main 函数流程，首先完成必要的初始化(lock, uart, mm, interrupt, pmu)，后调用 create_root_thread 创建第一个进程，在其中初始化 root_cap_group，将 ELF 用户程序加载到地址空间中。之后，调用 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，从而完成从内核态向用户态的第一次切换。

练习题 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

对应表格填写label即可，如下：

```
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
```

练习题 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。如果正确完成这一部分，可以通过 Userland 测试点。这代表着程序已经可以在用户态与内核态间进行正确切换。

1. 进入 exception_enter，先在栈上保存所有寄存器，并保存系统寄存器sp_el0, elr_el1, spsr_el1
2. 进入 exception_exit，与enter相反，恢复系统寄存器和用户寄存器
3. 内核切换

```
add x24, x24, #OFFSET_LOCAL_CPU_STACK
```

思考题 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，并传入 stdout 描述符(定义在syscall_dispatcher.c)，其中 __sydio_write 进行系统调用，最终进入 chcore_stdout_write 函数

练习题 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。

调用 chcore_syscall2，如下：

```
chcore_syscall2(CHCORE_SYS_putstr, (vaddr_t)buffer, size);
```

练习题 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)。Hello ChCore! 使用 chcore-libc 的编译器进行对其进行编译，编译输出文件名命名为 hello_world.bin，并将其放入 ramdisk 加载进内核运行。内核启动时将自动运行 文件名为 hello_world.bin 的可执行文件。

编写hello_world.c，如下：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello ChCore!\n");
    return 0;
}
```

后使用 build/chcore-libc/bin/musl-gcc 编译为hello_world.bin，放入 ramdisk，进入 chcore 后运行 hello_world.bin即可