ChCore Lab3

Problem 1

练习 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。

代码

```
cap_t sys_create_cap_group(unsigned long cap_group_args_p)
{
        ……
         /* cap current cap_group */
         /* LAB 3 TODO BEGIN */
         new_cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*new_cap_group));
         /* LAB 3 TODO END */
         ……
         /* LAB 3 TODO BEGIN */
         /* initialize cap group */

         cap_group_init(new_cap_group,
                        BASE_OBJECT_NUM,
                        args.badge);
         /* LAB 3 TODO END */
        ……
         /* 2st cap is vmspace */
         /* LAB 3 TODO BEGIN */
         vmspace = obj_alloc(TYPE_VMSPACE, sizeof(*vmspace));
         /* LAB 3 TODO END */

……
}
```

```
struct cap_group *create_root_cap_group(char *name, size_t name_len)
{
      ……
        /* LAB 3 TODO BEGIN */
        cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*cap_group));
        /* LAB 3 TODO END */
        ……

        /* LAB 3 TODO BEGIN */
        /* initialize cap group, use ROOT_CAP_GROUP_BADGE */
        cap_group_init(cap_group,
                       BASE_OBJECT_NUM,
                       /* Fixed badge */ ROOT_CAP_GROUP_BADGE);

        /* LAB 3 TODO END */
        ……

        /* LAB 3 TODO BEGIN */
        vmspace = obj_alloc(TYPE_VMSPACE, sizeof(*vmspace));
        /* LAB 3 TODO END */
      ……
}
```

注释

利用 obj_alloc 实现内核对象的分配，利用 cap_group_init 实现对 cap_group 的初始化

Problem 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。

代码

```
void create_root_thread(void)
{…
                /* 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_OF),
        sizeof(data));
memsz = (unsigned long)le64_to_cpu(*(u64 *)data);
/* LAB 3 TODO END */

struct pmobject *segment_pmo;
/* LAB 3 TODO BEGIN */
ret = create_pmo(ROUND_UP(memsz, 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),
        (void *)(((unsigned long)&binary_procmgr_bin_start)
                + ROOT_BIN_HDR_SIZE + offset),
```

```
                                filesz);
                /* LAB 3 TODO END */

                unsigned vmr_flags = 0;
                /* LAB 3 TODO BEGIN */
                /* Set flags*/
                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;
                /* LAB 3 TODO END */
        ......
}
```

注释

利用 memcpy 函数获取 elf 文件的元数据，创建 pmo，再根据元数据将 elf 文件拷贝进 pmo。最后设置标识。

Problem 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。

代码

```
void init_thread_ctx(struct thread *thread, vaddr_t stack, vaddr_t func,
                     u32 prio, u32 type, s32 aff)
{
        /* Fill the context of the thread */

        /* LAB 3 TODO BEGIN */
        /* 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_EL0t;
        /* LAB 3 TODO END */

......
}
```

注释

根据提示设置线程结构体的 SP_EL0, ELR_EL1,SPSR_EL1.

Problem 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。

内核在完成初始化后调用 create_root_thread 函数来创建第一个用户态线程，它所进行的操作如下：

1. 首先使用 memcpy 函数从 binary_procmgr_bin_start 的二进制镜像文件中读取数据。

2. 调用了 create_root_cap_group 函数创建第一个 cap_group 进程，进行了相应的内存、vmspace、slot id 分配。

3. 调用了 create_pmo 函数创建了一个物理内存对象作为用户栈。

4. 调用 obj_alloc 函数创建了第一个线程对象。

5. 处理 ELF 文件，包括循环遍历 ELF 程序头部信息，创建 PMO，映射虚拟内存等。

6. 调用 thread_init 函数初始化线程，线程加载着信息中记录的 elf 程序（实际上就是 procmgr 系统服务）。其中调用了 init_thread_ctx 函数用于初始化线程的上下文。

7. 执行线程测试，并将线程加入就绪队列。

至此执行完了第一个用户进程与第一个用户线程的创建。接下来在 main.c 中，调用了 sched() 与 eret_to_thread(switch_context()) 函数，选择调度创建的第一个线程，通过 switch_context()函数进行线程上下文的切换。 switch_context() 返回被选择线程的 thread_ctx 地址，即 target_thread->thread_ctx 。

eret_to_thread 最终调用了 irq_entry.S 中的 ___eret_to_thread 函数，将 target_thread->thread_ctx 写入 sp 寄存器后调用了 exception_exit 函数， exception_exit 恢复了用户态的寄存器状态并调用 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

Problem 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。

代码

```
EXPORT(el1_vector)

    /* LAB 3 TODO BEGIN */

    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 //Synchronous 32 -bit EL0

    exception_entry irq_el0_32

    exception_entry fiq_el0_32 // FIQ 32-bit EL0

    exception_entry error_el0_32

    /* LAB 3 TODO END */
```

注释

利用 exception_entry 配置异常向量表

Problem 6

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。

代码

```
.macro exception_enter

  /* LAB 3 TODO BEGIN */
```

```
    sub sp, sp, #ARCH_EXEC_CONT_SIZE

    stp x0, x1, [sp, #16 * 0]

    stp x2, x3, [sp, #16 * 1]

    stp x4, x5, [sp, #16 * 2]

    stp x6, x7, [sp, #16 * 3]

    stp x8, x9, [sp, #16 * 4]

    stp x10, x11, [sp, #16 * 5]

    stp x12, x13, [sp, #16 * 6]

    stp x14, x15, [sp, #16 * 7]

    stp x16, x17, [sp, #16 * 8]

    stp x18, x19, [sp, #16 * 9]

    stp x20, x21, [sp, #16 * 10]

    stp x22, x23, [sp, #16 * 11]

    stp x24, x25, [sp, #16 * 12]

    stp x26, x27, [sp, #16 * 13]

    stp x28, x29, [sp, #16 * 14]

    /* LAB 3 TODO END */

……

    /* LAB 3 TODO BEGIN */
```

```
    stp x30, x21, [sp, #16 * 15]

    stp x22, x23, [sp, #16 * 16]

    /* LAB 3 TODO END */

.endm
```

```
.macro  exception_exit

    /* LAB 3 TODO BEGIN */

    ldp x22, x23, [sp, #16 * 16]

    ldp x30, x21, [sp, #16 * 15]

    /* LAB 3 TODO END */

    msr     sp_el0, x21

    msr     elr_el1, x22

    msr     spsr_el1, x23

    /* LAB 3 TODO BEGIN */

    ldp x0, x1, [sp, #16 * 0]

    ldp x2, x3, [sp, #16 * 1]

    ldp x4, x5, [sp, #16 * 2]
```

```
    ldp x6, x7, [sp, #16 * 3]

    ldp x8, x9, [sp, #16 * 4]

    ldp x10, x11, [sp, #16 * 5]

    ldp x12, x13, [sp, #16 * 6]

    ldp x14, x15, [sp, #16 * 7]

    ldp x16, x17, [sp, #16 * 8]

    ldp x18, x19, [sp, #16 * 9]

    ldp x20, x21, [sp, #16 * 10]

    ldp x22, x23, [sp, #16 * 11]

    ldp x24, x25, [sp, #16 * 12]

    ldp x26, x27, [sp, #16 * 13]

    ldp x28, x29, [sp, #16 * 14]

    add sp, sp, #ARCH_EXEC_CONT_SIZE

    /* LAB 3 TODO END */

    eret

.endm
```

```
.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
```

注释

exception_enter 异常进入，先将 sp 下移，扩大栈空间用以保存各个寄存器的值。Sp 指令意为 store pair, 成对加载寄存器到内存，值得一提的是 sp_el0, spsr_el1, elr_el1 这三个特殊寄存器不能直接保存，需要用通用寄存器作为“中转”再保存到内存。

exception_exit 异常退出，是上述过程的逆过程

switch_to_cpu_stack: 首先从 TPIDR_EL1 线程指针/ID 寄存器获取当前 cpu 的 offset，再加上 #OFFSET_LOCAL_CPU_STACK ，这样 sp 才会指向 cpu_stack 地址。

Problem 7

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

解答

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。

___stdio_write 函数中会进行系统调用 syscall ，

```
syscall(SYS_writev, f->fd, iov, iovcnt);
```

在___stdout_write.c 文件中找到了对应函数，发现返回值调用了___stdio_write 函数：

```
return ___stdio_write(f, buf, len);
```

在 stdio.c 中可以看到 stdout 对应的 options：

```
struct fd_ops stdout_ops = {
        .read = chcore_stdio_read,
        .write = chcore_stdout_write,
        .close = chcore_stdout_close,
        .poll = chcore_stdio_poll,
        .ioctl = chcore_stdio_ioctl,
    .fcntl = chcore_stdio_fcntl,
};
```

对应的 chcore_stdout_write 就是最终调用的函数，其中的核心函数 put 进行了系统调用

Problem 8

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。

代码

```
static void put(char buffer[], unsigned size)
{
        /* LAB 3 TODO BEGIN */
        chcore_syscall2(CHCORE_SYS_putstr, (vaddr_t)buffer, size);
        /* LAB 3 TODO END */
}
```

注释

此处使用 chcore_syscalll2 调用 sys_putstr，并把 buffer 和 size 作为传入参数

Problem 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以下字符(测试将以此为得分点)。
Hello ChCore!

代码

```
 #include <stdio.h>

int main(int arg, char const *argv[]){

     printf("Hello ChCore!\n");

     return 0;

}
```

注释

运行./build/chcore-libc/bin/musl-gcc hello_chcore.c -o hello_chcore.bin 即可成功编译