lab 3：进程和线程

练习一：

> 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

- 代码补全： sys_create_cap_group

```
        /* cap current cap_group */
        /* LAB 3 TODO BEGIN */
        new_cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(struct
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(struct vmspace));

        /* LAB 3 TODO END */
```

- 思路:
  - 根据注释，以及题目的提示，使用`obj_alloc`分配对应的cap_group和vmspace对象
  - 然后再用cap_group_init对新分配的cap_group进行初始化，设置对应的参数，`BASE_OBJECT_NUM`和`args.badge`

    其中`args.badge`通过参考cap_group_init 的函数定义可以找到

- 代码补全： create_root_cap_group

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


        /* LAB 3 TODO END */
```

```
13
14          /* LAB 3 TODO BEGIN */
15          vmspace = obj_alloc(TYPE_VMSPACE, sizeof(struct vmspace));
16
17          /* LAB 3 TODO END */
18
19          /* LAB 3 TODO BEGIN */
20          slot_id = cap_alloc(cap_group, vmspace);
21          /* LAB 3 TODO END */
```

思路:

- 根据注释，以及题目的提示，补充的部分基本和`sys_create_cap_group`一致
- 不过多了一步"对分配得到的 vmspace 对象则需要调用 cap_alloc 分配对应的槽（slot）"，参考`cap_alloc`的定义， 调用对应`cap_alloc`即可实现对应操作

练习二:

> 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。

- 代码补全一:

```
1                   /* LAB 3 TODO BEGIN */
2                   /* Get offset, vaddr, filesz, memsz from image*/
3
4                   memcpy(data,
5                          (void *)((unsigned long)&binary_procmgr_bin_start
6                                   + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
7                                   + PHDR_OFFSET_OFF),
8                          sizeof(data));
9                   offset = (unsigned int)le32_to_cpu(*(u32 *)data);
10
11                  memcpy(data,
12                         (void *)((unsigned long)&binary_procmgr_bin_start
13                                  + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
14                                  + PHDR_VADDR_OFF),
15                         sizeof(data));
16                  vaddr = (unsigned int)le32_to_cpu(*(u32 *)data);
17
18
19                  memcpy(data,
20                         (void *)((unsigned long)&binary_procmgr_bin_start
21                                  + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
22                                  + PHDR_FILESZ_OFF),
23                         sizeof(data));
24                  filesz = (unsigned int)le32_to_cpu(*(u32 *)data);
25
26                  memcpy(data,
27                         (void *)((unsigned long)&binary_procmgr_bin_start
28                                  + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
29                                  + PHDR_MEMSZ_OFF),
```

```
                        sizeof(data));
                memsz = (unsigned int)le32_to_cpu(*(u32 *)data);
```

- 思路：

  根据代码注释，需要获取对应offset, vaddr, filesz, memsz from image

  参考从image获取flags的操作，将`PHDR_FLAGS_OFF`改成对应的`PHDR_OFFSET_OFF`等即可获取对应的数据

- 代码补全二：

```
                /* LAB 3 TODO BEGIN */

                create_pmo(memsz  ,PMO_DATA,root_cap_group, 0 ,&segment_pmo);

                /* LAB 3 TODO END */
```

- 思路：

  根据create_pmo的定义，填入参数memsz 表示对应pno的大小，pmo_type为`PMO_DATA`，立即分配物理内存，

  以及对应的cap_group的`root_cap_group`, 物理地址是零地址，以及作为存放返回值的pmo对象`segment_pmo`

- 代码补全三：

```
                /* 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 +  offset +ROOT_BIN_HDR_SIZE ), filesz);

```

- 思路：

  根据代码注释需要将elf文件中的内容copy到内存中，使用`mem_set`映射虚拟内存空间，在使用memcpu进行copy内容，

  其中文件内容大小为(void *)((unsigned long)&binary_procmgr_bin_start + offset +ROOT_BIN_HDR_SIZE)

- 代码补全四：

```
                /* 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 */
```

- 思路：

  设置vmr_flags，利用flags进行判断可读可写可执行权限，然后将对应的vmr权限位相或即可

练习三：

> 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。

- 代码补全：

```
        /* 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 */
```

- 思路：

  根据代码注释，需要设置上下文中的`SP_EL0`，`SLR_EL1`，`SPSR_EL1`， 分别对应着线程的Stack pointer(sp)寄存器，异常链接寄存器，程序状态保存寄存器

  - 因为是用户态把stack赋值给SP_EL0
  - 把func赋给ELR_EL1 ， 在eret的时候可以正常跳转到func对应的地址
  - 把SPSR_EL1_EL0t 赋值给SPSR_EL1，实现特权级的切换

思考题四：

> 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。

- 解答：

  调用关系：

  create_root_thread() -> tread_init() -> obj_get()、obj_put()、 create_thread_ctx()、init_thread_ctx()

sched()

eret_to_thread(switch_context()) -> __eret_to_thread() -> exception_exit() -> eret

- 创建第一个线程、进行线程初始化的时候，包括分配cap_group,vmspace，然后进行上下文初始化等
- sched() 进行了一次调度，在调度队列中选出线程
- 然后通过switch_context() 进行上下文的切换，将cpu_info中记录的当前CPU线程的上下文记录为被选择的线程的上下文
- eret_to_thread（）通过调用 __eret_to_thread() -> exception_exit() -> eret,跳转到选出的线程的用户态

练习题五：

> 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。

- 补全代码：

```
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
    exception_entry irq_el0_32
    exception_entry fiq_el0_32
    exception_entry error_el0_32


    /* LAB 3 TODO END */

irq_el1t:
fiq_el1t:
fiq_el1h:
error_el1t:
error_el1h:
sync_el1t:
```

```
    /* LAB 3 TODO BEGIN */
    bl unexpected_handler

    /* LAB 3 TODO END */

sync_el1h:
    exception_enter
    mov x0, #SYNC_EL1h
    mrs x1, esr_el1
    mrs x2, elr_el1

    /* LAB 3 TODO BEGIN */
    /* jump to handle_entry_c, store the return value as the ELR_EL1 */

    bl  handle_entry_c


    /* LAB 3 TODO END */
```

- 根据代码注释，因此填写异常向量表
- 对于sycn_el1h的情况，跳转到handle_entry_c

  对rq_el1t、fiq_el1t、fiq_el1h、error_el1t、error_el1h、sync_el1t，跳转到unexpected_handler

练习题六：

> 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。显示如下结果

- 补全代码：exception_enter

```
.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 */

    mrs x21, sp_el0
    mrs x22, elr_el1
    mrs x23, spsr_el1

    /* LAB 3 TODO BEGIN */

    stp x30, x21, [sp, #16 * 15]
    stp x22, x23, [sp, #16 * 16]

    /* LAB 3 TODO END */

```

- 思路：

  先把sp减去 #ARCH_EXEC_CONT_SIZE，预留出存储上下文的空间

  把x0-x29通用寄存器中的值，存入到内存中

  用`mrs`把sp_el0 、elr_el1、 sprs_el1中的值存到寄存器中，并把x30 LinkResigter

  都存到内存中

- 补全代码：exception_exit

```
    /* LAB 3 TODO BEGIN */

    ldp x30, x21, [sp, #16 * 15]
    ldp x22, x23, [sp, #16 * 16]

    /* 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 */


```

- 思路：

  进行exception_exit的逆过程，这里不在赘述

- 代码补全：switch_to_cpu_stack

```
.macro switch_to_cpu_stack
    mrs     x24, TPIDR_EL1
    /* LAB 3 TODO BEGIN */
    add x24, x24, #OFFSET_LOCAL_CPU_STACK
    /* LAB 3 TODO END */
```

从TRIDR_EL1读取当前核的per_cpu_info ， 然后再加上local_cpu_stack的偏移量

思考题7：

> 尝试描述 printf 如何调用到 chcore_stdout_write 函数。

- 解答：

  在printf中利用`ret = vfprintf(stdout, fmt, ap);` 调用`vfprintf`函数

  在`vfprintf`函数中使用了 stdout (f) 的一系列操作

```
    /* the copy allows passing va_list* even if va_list is an array */
    va_copy(ap2, ap);
    if (printf_core(0, fmt, &ap2, nl_arg, nl_type) < 0) {
        va_end(ap2);
        return -1;
    }

    FLOCK(f);
    olderr = f->flags & F_ERR;
    if (f->mode < 1) f->flags &= ~F_ERR;
    if (!f->buf_size) {
        saved_buf = f->buf;
        f->buf = internal_buf;
        f->buf_size = sizeof internal_buf;
        f->wpos = f->wbase = f->wend = 0;
    }
    if (!f->wend && __towrite(f)) ret = -1;
    else ret = printf_core(f, fmt, &ap2, nl_arg, nl_type);
    if (saved_buf) {
        f->write(f, 0, 0);
        if (!f->wpos) ret = -1;
```

```
        f->buf = saved_buf;
        f->buf_size = 0;
        f->wpos = f->wbase = f->wend = 0;
    }
    if (f->flags & F_ERR) ret = -1;
    f->flags |= olderr;
    FUNLOCK(f);
    va_end(ap2);
    return ret;
```

其中， `f->write(f, 0, 0);` 对应stdout的write操作

而stdout的write操作被定义为 __stdout_write, 可见下面的代码

```
static unsigned char buf[BUFSIZ+UNGET];
hidden FILE __stdout_FILE = {
    .buf = buf+UNGET,
    .buf_size = sizeof buf-UNGET,
    .fd = 1,
    .flags = F_PERM | F_NORD,
    .lbf = '\n',
    .write = __stdout_write,
    .seek = __stdio_seek,
    .close = __stdio_close,
    .lock = -1,
};
FILE *const stdout = &__stdout_FILE;
FILE *volatile __stdout_used = &__stdout_FILE;
```

而**stdout_write中调用了**stdio_write函数

```
size_t __stdout_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct winsize wsz;
    f->write = __stdio_write;
    if (!(f->flags & F_SVB) && __syscall(SYS_ioctl, f->fd, TIOCGWINSZ, &wsz))
        f->lbf = -1;
    return __stdio_write(f, buf, len);
}

```

在__stdio_write中进一步调用了syscall(SYS_writev, f->fd, iov, iovcnt);的系统调用,也就是`stdout->fd`

```
size_t __stdio_write(FILE *f, const unsigned char *buf, size_t len)
{
    struct iovec iovs[2] = {
        { .iov_base = f->wbase, .iov_len = f->wpos-f->wbase },
        { .iov_base = (void *)buf, .iov_len = len }
    };
    struct iovec *iov = iovs;
    size_t rem = iov[0].iov_len + iov[1].iov_len;
    int iovcnt = 2;
    ssize_t cnt;
```

```
    for (;;) {
        cnt = syscall(SYS_writev, f->fd, iov, iovcnt);
        if (cnt == rem) {
            f->wend = f->buf + f->buf_size;
            f->wpos = f->wbase = f->buf;
            return len;
        }
        if (cnt < 0) {
            f->wpos = f->wbase = f->wend = 0;
            f->flags |= F_ERR;
            return iovcnt == 2 ? 0 : len-iov[0].iov_len;
        }
        rem -= cnt;
        if (cnt > iov[0].iov_len) {
            cnt -= iov[0].iov_len;
            iov++; iovcnt--;
        }
        iov[0].iov_base = (char *)iov[0].iov_base + cnt;
        iov[0].iov_len -= cnt;
    }
}

```

在`syscall_dispatcher.c`中，`fd_dic[fd1]->fd_op = &stdout_ops;`

其中`stdout_ops`中，把`write`操作定义为`chcore_stdout_write`

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。

- 代码补全：

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

- 思路：

  根据提示调用chcore_syscall2,填入对应put的系统调用号，以及buffer和size参数即可

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_chcore.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_chcore.bin 的可执行文件。

> 提示:
>
> ChCore 的编译工具链在 build/chcore-libc/bin 文件夹中。
> 如使用 cmake 进行编译，可以将工具链文件指定为 build/toolchain.cmake，将默认使用 ChCore 编译工具链。

思路以及做法：

- 写好hello_chcore.c文件

```
#include <stdio.h>
int main ()
{
     printf("Hello ChCore!") ;
}

```

- 命令行跳转到 `/home/os/OS-Course-Lab/build/chcore-libc/bin` 目录下，然后使用 `.\musl-gcc` 进行编译，输出为 `hello_chcore.bin` 文件
- 放在 `ramdisk` 目录下即可