Lab3 Report

1. 在 kernel/object/cap_group.c 中完善 sys_create_cap_group、create_root_cap_group 函数
   a) sys_create_cap_group

   使用 obj_alloc 分配 cap_group 对象 然后进行初始化

```
/* cap current cap_group */
/* LAB 3 TODO BEGIN */
new_cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*new_cap_group));
/* LAB 3 TODO END */
if (!new_cap_group) {
        r = -ENOMEM;
        goto out_fail;
}
/* LAB 3 TODO BEGIN */
/* initialize cap group */
cap_group_init(new_cap_group, BASE_OBJECT_NUM, args.badge);
/* LAB 3 TODO END */
```

   分配 VMSPACE 对象

```
/* 2st cap is vmspace */
/* LAB 3 TODO BEGIN */
vmspace = obj_alloc(TYPE_VMSPACE, sizeof(*vmspace));
/* LAB 3 TODO END */

if (!vmspace) {
        r = -ENOMEM;
        goto out_free_obj_vmspace;
}

vmspace_init(vmspace, args.pcid);

r = cap_alloc(new_cap_group, vmspace);
```

   b) create_root_cap_group
      - 分配 cap_group 对象 并进行初始化
      - 分配 vmspace 对象，初始化后在 cap_group 中进行分配

```
/* LAB 3 TODO BEGIN */
cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*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(*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 */

BUG_ON(slot_id != VMSPACE_OBJ_ID);
```

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

   Program Header 如下图所示

Program header[10]

<table>
<tr><th colspan="2">Offset</th><th colspan="2">Size (bytes)</th><th rowspan="2">Field</th><th rowspan="2">Purpose</th></tr>
<tr><th>32-bit</th><th>64-bit</th><th>32-bit</th><th>64-bit</th></tr>
<tr><td colspan="2">0x00</td><td colspan="2">4</td><td>p_type</td><td>Identifies the type of the segment.
<table>
<tr><th>Value</th><th>Name</th><th>Meaning</th></tr>
<tr><td>0x00000000</td><td>PT_NULL</td><td>Program header table entry unused.</td></tr>
<tr><td>0x00000001</td><td>PT_LOAD</td><td>Loadable segment.</td></tr>
<tr><td>0x00000002</td><td>PT_DYNAMIC</td><td>Dynamic linking information.</td></tr>
<tr><td>0x00000003</td><td>PT_INTERP</td><td>Interpreter information.</td></tr>
<tr><td>0x00000004</td><td>PT_NOTE</td><td>Auxiliary information.</td></tr>
<tr><td>0x00000005</td><td>PT_SHLIB</td><td>Reserved.</td></tr>
<tr><td>0x00000006</td><td>PT_PHDR</td><td>Segment containing program header table itself.</td></tr>
<tr><td>0x00000007</td><td>PT_TLS</td><td>Thread-Local Storage template.</td></tr>
<tr><td>0x60000000</td><td>PT_LOOS</td><td rowspan="2">Reserved inclusive range. Operating system specific.</td></tr>
<tr><td>0x6FFFFFFF</td><td>PT_HIOS</td></tr>
<tr><td>0x70000000</td><td>PT_LOPROC</td><td rowspan="2">Reserved inclusive range. Processor specific.</td></tr>
<tr><td>0x7FFFFFFF</td><td>PT_HIPROC</td></tr>
</table></td></tr>
<tr><td></td><td>0x04</td><td></td><td>4</td><td>p_flags</td><td>Segment-dependent flags (position for 64-bit structure).
<table>
<tr><th>Value</th><th>Name</th><th>Meaning</th></tr>
<tr><td>0x1</td><td>PF_X</td><td>Executable segment.</td></tr>
<tr><td>0x2</td><td>PF_W</td><td>Writeable segment.</td></tr>
<tr><td>0x4</td><td>PF_R</td><td>Readable segment.</td></tr>
</table></td></tr>
<tr><td>0x04</td><td>0x08</td><td>4</td><td>8</td><td>p_offset</td><td>Offset of the segment in the file image.</td></tr>
<tr><td>0x08</td><td>0x10</td><td>4</td><td>8</td><td>p_vaddr</td><td>Virtual address of the segment in memory.</td></tr>
<tr><td>0x0C</td><td>0x18</td><td>4</td><td>8</td><td>p_paddr</td><td>On systems where physical address is relevant, reserved for segment's physical address.</td></tr>
<tr><td>0x10</td><td>0x20</td><td>4</td><td>8</td><td>p_filesz</td><td>Size in bytes of the segment in the file image. May be 0.</td></tr>
<tr><td>0x14</td><td>0x28</td><td>4</td><td>8</td><td>p_memsz</td><td>Size in bytes of the segment in memory. May be 0.</td></tr>
<tr><td>0x18</td><td></td><td>4</td><td></td><td>p_flags</td><td>Segment-dependent flags (position for 32-bit structure). See above <code>p_flags</code> field for flag definitions.</td></tr>
<tr><td>0x1C</td><td>0x30</td><td>4</td><td>8</td><td>p_align</td><td><code>0</code> and <code>1</code> specify no alignment. Otherwise should be a positive, integral power of 2, with <code>p_vaddr</code> equating <code>p_offset</code> modulus <code>p_align</code>.</td></tr>
<tr><td>0x20</td><td>0x38</td><td colspan="4"></td><td>End of Program Header (size).</td></tr>
</table>

   按照要求，仿照函数中读取 elf 头的示例将 program 头内容进行读取，为 Segments 分配 PMO

```
/* 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);
/* 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 */
```

先清除 segment_pmo 区域，然后将文件内容写入 pmo

```
/* 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 */
```

根据 flag 设置标志位，进行 vmspace 的映射

```
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 */

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

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
   SP_EL0 设置为栈地址
   ELR_EL1 设置为 func 地址 -> 从 EI1 返回之后，PC 指向 func 函数
   SPSR_EL1 设置为 SPSR_EL1_EL0t，设置使用 EL0 的 SP

```
/* 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 */
```

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

   内核完成初始化之后调用 Create_root_thread 从 procmgr 读取应用信息，之后调用 create_toot_capgroup(其中分配并初始化 cap_group 和 vmspace 对象)，然后创建 pmo 并进行映射，将文件内容填充到 pmo，设置 flag；初始化栈，然后从 root_capgroup 中分配线程对象，创建第一个线程。通过 shed()函数调用创建的 root 进程

   调用 switch_context() 初始化新的线程并完成线程上下文的切换，eret_to_thread(switch_context())接收到该函数的返回值(被选择线程的 thread_ctx 地址)，最终调用 kernel/arch/aarch64/irq/irq_entry.S 中的 __eret_to_thread 函数，将 thread_ctx 地址写入 sp，调用 exception_exit 函数将内存中的线程上下文恢复到硬件(CPU)，最终调用 eret 返回到用户态。

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
   根据 lab 文档进行填写，并注意以下注释

```
The selected stack pointer can be indicated by a suffix to the Exception Level:
 - t: SP_EL0 is used
 - h: SP_ELx is used

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 */
```

TPIDR 为线程信息存储位置，加上对应的 offset 并读取该地址内容用于恢复 sp

```
.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
```

对于其他的异常，直接跳转到 unexpected_handler

```
irq_el1t:
fiq_el1t:
fiq_el1h:
error_el1t:
error_el1h:
sync_el1t:
    /* LAB 3 TODO BEGIN */
    bl unexpected_handler
    /* LAB 3 TODO END */
```

同步异常则进行处理

```
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
    str x0, [sp, #16 * 16]
    /* LAB 3 TODO END */
    exception_exit
```

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

Enter 函数，sp 减少 ARCH_EXEC_CONT_SIZE，保留存储寄存器数值的空间，然后再将对应寄存器数值逐个存放在这篇区域，Exit 函数则相反，从内存中恢复，然后 sp 加上 ARCH_EXEC_CONT_SIZE

```
.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 */
```

```
.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
```

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

```
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;
}
```

Vfprintf 调用 f->write (f 为 stdout)

Stdout 中的 write 被定义为__stdout_write

```
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);
}
```

调用__stdio_write 函数 __stdio_write 中调用 syscall(SYS_writev, ……)

```
for (;;) {
    cnt = syscall(SYS_writev, f->fd, iov, iovcnt);
```

SYS_writev 的调用被捕获并处理，调用 chcore_writev 函数

```
case SYS_writev: {
        return chcore_writev(a, (const struct iovec *)b, c);
}
```

Chcore_writev 函数调用 chcore_write 函数

```
ret = chcore_write(fd,
                   (void *)((iov + iov_i)->iov_base),
                   (size_t)(iov + iov_i)->iov_len);
```

Chcore_write 函数调用对应 fd(stdout)的 fd_op->write 函数，而 stdout 对应 fd 被定义为 &stdout_ops:

```
ssize_t chcore_write(int fd, void *buf, size_t count)
{
        if (fd < 0 || fd_dic[fd] == 0)
                return -EBADF;
        return fd_dic[fd]->fd_op->write(fd, buf, count);
}
```

```
/* STDOUT */
fd1 = alloc_fd();
assert(fd1 == STDOUT_FILENO);
fd_dic[fd1]->type = FD_TYPE_STDOUT;
fd_dic[fd1]->fd = fd1;
/* LAB 3 READ BEGIN */
fd_dic[fd1]->fd_op = &stdout_ops;
/* LAB 3 READ END */
```

```
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,
};
```

```
static ssize_t chcore_stdout_write(int fd, void *buf, size_t count)
{
        char buffer[STDOUT_BUFSIZE];
        size_t size = 0;

        for (char *p = buf; p < (char *)buf + count; p++) {
                if (size + 2 > STDOUT_BUFSIZE) {
                        put(buffer, size);
                        size = 0;
                }

                if (*p == '\n') {
                        buffer[size++] = '\r';
                }
                buffer[size++] = *p;
        }

        if (size > 0) {
                put(buffer, size);
        }

        return count;
}
```

Stdout_ops 中的 write 函数被定义为 chcore_stdout_write，因此在上面 fd_dic[fd]->fd_op->write(fd, buf, count) 时便已经调用到了 chcore_stdout_write 函数

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，最终在 sys_putstr 中调用 uart_send 函数(在 lab1 中也用到了)

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

```
const void *syscall_table[NR_SYSCALL] = {
        [0 ... NR_SYSCALL - 1] = sys_null_placeholder,

        /* Character IO */
        [CHCORE_SYS_putstr] = sys_putstr,
        [CHCORE_SYS_getc] = sys_getc,
```

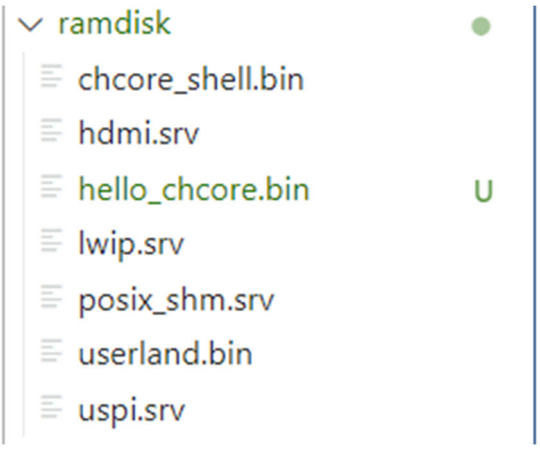
```
for (i = 0; i < copy_len; ++i) {
        uart_send((unsigned int)buf[i]);
}
```

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!”

```
#include <stdio.h>

int main(int argc, char const *argv[]) {
    printf("Hello ChCore!\n");
    return 0;
}
```

对源文件使用 chcore 编译工具链进行编译 放入 ramdisk



运行 make qemu 的结果

```
load library complete
load library complete
load library complete
[WARN] SYS_rt_sigprocmask is not implemented.
[WARN] SYS_membarrier is not implmeneted.
Hello ChCore!
[lwip] Host at 192.168.0.3 mask 255.255.255.0 gateway 192.168.0.1
 ______    __  __    ______    __  __    ______    __        __
/\  ___\  /\ \_\ \  /\  ___\  /\ \_\ \  /\  ___\  /\ \      /\ \
\ \ \____ \ \  __ \ \ \___  \ \ \  __ \ \ \  __\  \ \ \____ \ \ \____
 \ \_____\ \ \_\ \_\ \/\_____\ \ \_\ \_\ \ \_____\ \ \_____\ \ \_____\
  \/_____/  \/_/\/_/  \/_____/  \/_/\/_/  \/_____/  \/_____/  \/_____/


Welcome to ChCore shell!
$ [WARN] SYS_rt_sigprocmask is not implemented.
[WARN] SYS_membarrier is not implmeneted.
[lwip] TCP/IP initialized.
[lwip] Add netif 0x5aaad5c2eed0
[lwip] register server value = 0
```

最终运行结果

```
[100%] Completed 'kernel'
[100%] Built target kernel
Succeeded to build all targets
To use chcore toolchain, please input the commnad
        export CMAKE_TOOLCHAIN_FILE=/home/voider/Desktop/OS/build/toolchain.cmake
make[1]: Leaving directory '/home/voider/Desktop/OS'
===============
Grading lab 2...(may take 10 seconds)
GRADE: Cap create pretest: 20
GRADE: Userland: 40
GRADE: Successful printf: 20
GRADE: Hello ChCore: 20
===============
Score: 100/100
voider@voider:~/Desktop/OS$
```