Lab3：进程管理

陈景韬 523030910028

练习题1

第一个block处，按照要求，调用obj_alloc为new_cap_group分配一个新的类型为TYPE_CAP_GROUP的对象，用sizeof()传入大小。

```
new_cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*new_cap_group));
```

其中关于object_type类型的定义可以在/include/object/object.h中找到：

```
enum object_type {
        TYPE_NO_TYPE = 0,
        TYPE_CAP_GROUP,
        TYPE_THREAD,
        TYPE_CONNECTION,
        TYPE_NOTIFICATION,
        TYPE_IRQ,
        TYPE_PMO,
        TYPE_VMSPACE,
#ifdef CHCORE_OPENTRUSTEE
        TYPE_CHANNEL,
        TYPE_MSG_HDL,
#endif /* CHCORE_OPENTRUSTEE */
        TYPE_PTRACE,
        TYPE_NR,
};
```

用于定义对象类型。

第二个block处，按提示调用在cap_group.c中定义的cap_group_init_user函数，利用用户参数对其进行初始化，最后一个函数传入&args。然后其实就可以了，因为后面发现cap_group_init_user中也已经有pid设置的实现。

```
cap_group_init_user(new_cap_group, BASE_OBJECT_NUM, &args);
```

第三个Block处，vmspace分配对象。

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

第四个block处，和前面sys_create一样即可：

```
cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(*cap_group));
```

第五个block处，按提示调用cap_group_init_common并传入ROOT_CAP_GROUP_BADGE。

```
cap_group_init_common(cap_group, BASE_OBJECT_NUM, ROOT_CAP_GROUP_BADGE);
```

第六个block处，同样为vmspace分配对象：

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

第七个block处，模仿前文为vmspace分配槽id：

```
slot_id = cap_alloc(cap_group, vmspace);
```

两个函数实现在内核态创建一般/初始能力组的功能。

练习题2

第一个block处，模仿前文对flags的获取，从镜像中获取几个量：

```
memcpy(data,
        (void *)((unsigned long)&binary_procmgr_bin_start
                + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                + PHDR_OFFSET_OFF),
        sizeof(data));
offset = (unsigned int)le32_to_cpu(*(u32 *)data);

...

memcpy(data,
        (void *)((unsigned long)&binary_procmgr_bin_start
                + ROOT_PHDR_OFF + i * ROOT_PHENT_SIZE
                + PHDR_MEMSZ_OFF),
        sizeof(data));
memsz = (unsigned int)le32_to_cpu(*(u32 *)data);
```

第二个block处，先利用memsz计算需要分配的pmo大小，然后获取镜像中的内容字段的起始点存储在segment_content_kvaddr中，可以检查一下filesz和memsz的关系是否合法，然后创建一个pmo并赋予其所有权利。有关权利的定义可以在memory.h中找到。

```
size_t pmo_size = ROUND_UP(memsz, PAGE_SIZE);

vaddr_t segment_content_kvaddr = ((unsigned long)&binary_procmgr_bin_start) + offset;

BUG_ON(filesz != memsz);

ret = create_pmo(PAGE_SIZE, PMO_DATA, root_cap_group, 0, &segment_pmo, PMO_ALL_RIGHTS);
```

第三个block处，不知道之前segment_pmo之前有没有分配内存空间，这里先释放一下，然后再把它指回到现在这个镜像的内容上，分配start和size，利用刚刚计算的segment_content_kvaddr。

```
kfree((void *)phys_to_virt(segment_pmo -> start));

segment_pmo -> start = virt_to_phys(segment_content_kvaddr);

segment_pmo -> size = pmo_size;
```

第四个block处，用前面得到的PHDR_FLAGS_?来一一对应到虚拟内存空间的权限即可。

```
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

需要为线程分配上下文，分别给SP_EL0，ELR_EL1，SPSR_EL1分配线程的堆栈地址，异常处理函数地址和当前程序状态。

```
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;
```

因为当前在用户态线程模式所以赋值为SPSR_EL1_EL0t。

思考题4

首先调用create_root_thread(object/thread.c)，在其中我们按顺序执行：

- 利用镜像地址参数读取elf头部
- 调用create_root_cap_group()创建根能力组
- 调用obj_get()初始化虚拟地址
- 调用create_pmo()创建物理内存对象
- 调用vmspace_map_range()建立p-v映射
- 调用obj_alloc()创建线程对象
- 读取程序头部
- 类似，创建物理内存对象，建立映射
- 调用commit_page_to_pmo()为物理内存对象分配页
- 调用prepare_env()准备相关环境
- 调用sched_enqueue()将线程放入队列等待

练习题5

参照上面给出的异常向量表或者下文的异常变量名填入异常表条目：

```
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
```

按照要求在sync_el1t等错误信息下跳转到unexpected_handler：

```
bl unexpected_handler
```

在sync_el1h时，要求将返回值放入elr_el1，这里要结合我们下一题的代码，因为马上回调用exception_exit，在其中恢复系统寄存器的时候会用内存中的存储值来覆盖elr_el1，于是我们找到对应elr_el1的内存地址，将x0存入：

```
bl handle_entry_c
str x0, [sp, #16 * 16]
```

练习题6

进入异常处理流程，要下陷到内核态，将普通寄存器和系统寄存器依次存入内存：

```
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]

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

在退出时则做相反的操作，将其从内存中取出：

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

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
```

在栈切换函数中，将x24加上一个到线程上下文的offset并赋给sp，达成切换堆栈的目的：

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

思考题7

找到printf的定义，在/Thirdparty/musl-libc/src/stdio/printf.c：

```
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;
}
```

这其中调用了vfprintf，后者被定义在/Thirdparty/musl-libc/src/stdio/vfprintf.c

```
int vfprintf(FILE *restrict f, const char *restrict fmt, va_list ap)
{
	va_list ap2;
	int nl_type[NL_ARGMAX+1] = {0};
	union arg nl_arg[NL_ARGMAX+1];
	unsigned char internal_buf[80], *saved_buf = 0;
	int olderr;
	int ret;

	/* 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;
}
```

核心调用在于 `f->write(f, 0, 0)` ，追溯这里的f实际上是我在 `printf.c` 中传入的stdout流，因此查看./stdout.c：

```
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,
};
```

其中write部分调用了__stdout_write，继续找到./__stdout_write.c：

```
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;
        }
```

这里调用了syscall系统服务来使用SYS_writev，在/Lab3/user/chcore-libc/libchcore/porting/overrides/src/chcore-port/syscall_dispatcher.c中我们找关于SYS_writev的部

分。因为这里传入了三个参数，在__syscall3()下找：

```
case SYS_writev: {
        return __syscall6(SYS_writev, a, b, c, 0, 0, 0);
}
```

进一步调用__syscall6()，继续去找：

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

进一步调用chcore_writev()，可以在./fd.c中找到：

```
ssize_t chcore_writev(int fd, const struct iovec *iov, int iovcnt)
{
        int iov_i;
        ssize_t byte_written, ret;

        if ((ret = iov_check(iov, iovcnt)) != 0)
                return ret;

        byte_written = 0;
        for (iov_i = 0; iov_i < iovcnt; iov_i++) {
                ret = chcore_write(fd,
                                   (void *)((iov + iov_i)->iov_base),
                                   (size_t)(iov + iov_i)->iov_len);
                if (ret < 0) {
                        return ret;
                }

                byte_written += ret;
                if (ret != (iov + iov_i)->iov_len) {
                        return byte_written;
                }
        }
        return byte_written;
}
```

进一步调用chcore_write，在fd.c中同样可以找到：

```
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);
}
```

fd_dic[fd]是一个fd_desc指针，后者定义可以在fd.h中找到：

```
struct fd_desc {
        /* Identification used by corresponding service */
        union {
                int conn_id;
                int fd;
        };
        /* Baisc informantion of fd */
        int flags; /* Flags of the file */
        cap_t cap; /* Service's cap of fd, 0 if no service */
        enum fd_type type; /* Type for debug use */
        struct fd_ops *fd_op;

        /* stored termios */
        struct termios termios;

        /* Private data of fd */
        void *private_data;
};
```

可以看到其指向的fd_op是一个fd_ops类型的指针，后者的定义同样在fd.h中找到：

```
struct fd_ops {
        ssize_t (*read)(int fd, void *buf, size_t count);
        ssize_t (*write)(int fd, void *buf, size_t count);
        ssize_t (*pread)(int fd, void *buf, size_t count, off_t offset);
        ssize_t (*pwrite)(int fd, void *buf, size_t count, off_t offset);
        int (*close)(int fd);
        int (*poll)(int fd, struct pollarg *arg);
        int (*ioctl)(int fd, unsigned long request, void *arg);
        int (*fcntl)(int fd, int cmd, int arg);
};

extern struct fd_ops epoll_ops;
extern struct fd_ops socket_ops;
extern struct fd_ops file_ops;
extern struct fd_ops event_op;
extern struct fd_ops timer_op;
extern struct fd_ops pipe_op;
extern struct fd_ops stdin_ops;
extern struct fd_ops stdout_ops;
extern struct fd_ops stderr_ops;
```

因为我们传入的是stdout，因此执行的应该是stdout_ops中的write()，后者在stdio.c中可以找到：

```
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,
};
```

因此我们调用到了chcore_stdout_write()，也被定义在stdio.c中：

```
static ssize_t chcore_stdout_write(int fd, void *buf, size_t count)
{
        /* TODO: stdout should also follow termios flags */
        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;
}
```

练习题8

查看put，只传入了两个参数，因此使用chcore_syscall2()调用CHCORE_SYS_putstr：

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

练习题9

编写测试程序如下：

```
#include <stdio.h>

int main(){
    printf("Hello ChCore!");
    return 0;
}
```

注意两个c大写，然后用 `./build/chcore-libc/bin/musl-gcc` 的工具编译，并将结果放入 `./ramdisk` ， 命名为 `hello_world.bin` ， 启动内核可以看到成功输出。