Chcore Lab3 实验报告

519021910913 黄喆敏

> 练习 1: 内核从完成必要的初始化到用户态程序的过程是怎么样的？尝试描述一下调用关系。

答：主要的调用关系如下：

```
main
├── uart_init
├── mm_init
├── arch_interrupt_init
├── create_root_thread
    ├── create_root_cap_group
    ├── __create_root_thread
    └── switch_to_thread
└── eret_to_thread
    └── switch_context
```

- 调用`uart_init`，`mm_init`，`arch_interrupt_init`，分别初始化uart、内存管理、异常向量模块。
- 调用`create_root_thread`创建根进程。其中先调用`create_root_cap_group`，创建根进程对应的`cap_group`，然后调用`__create_root_thread`，创建根进程对应的线程，并进行初始化。最后调用`switch_to_thread`，切换到这个线程。
- 调用`eret_to_thread(switch_context())`，进行上下文切换，并跳转到所选的线程。其中`__eret_to_thread`还调用了`exception_exit`，将context中的数据存到寄存器中。

> 练习 2: 在 `kernel/object/cap_group.c` 中完善 `cap_group_init`、`sys_create_cap_group`、`create_root_cap_group` 函数。

答：

`cap_group_init`：对于`cap_group`包含的每一个成员，
即`slot_table`，`thread_list`，`thread_cnt`，`pid`，我们分别进行初始化即可。`cap_group_name`仅为调试所用，因此不需要初始化。

```
int cap_group_init(struct cap_group *cap_group, unsigned int size, u64 pid)
{
        struct slot_table *slot_table = &cap_group->slot_table;
        /* LAB 3 TODO BEGIN */
        slot_table_init(slot_table, size);
        init_list_head(&cap_group->thread_list);
        cap_group->pid = pid;
        cap_group->thread_cnt = 0;
        /* LAB 3 TODO END */
        return 0;
}
```

`sys_create_cap_group`：根据语义，我们通过`obj_alloc`，分配全新的`cap_group`和`vmspace`对象。对于`cap_group`，我们使用刚才写的`cap_group_init`初始化。

```
int sys_create_cap_group(u64 pid, u64 cap_group_name, u64 name_len, u64 pcid)
{
        ......
        /* LAB 3 TODO BEGIN */
        /* cap current cap_group */
        new_cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(struct cap_group));
        /* LAB 3 TODO END */
        if (!new_cap_group) {
                r = -ENOMEM;
                goto out_fail;
        }
        /* LAB 3 TODO BEGIN */
        cap_group_init(new_cap_group, sizeof(struct cap_group), pid);
        /* LAB 3 TODO END */
        cap = cap_alloc(current_cap_group, new_cap_group, 0);
        if (cap < 0) {
                r = cap;
                goto out_free_obj_new_grp;
        }
        /* 1st cap is cap_group */
        if (cap_copy(current_thread->cap_group, new_cap_group, cap)
            != CAP_GROUP_OBJ_ID) {
                printk("init cap_group cap[0] is not cap_group\n");
                r = -1;
                goto out_free_cap_grp_current;
        }
        /* 2st cap is vmspace */
        /* LAB 3 TODO BEGIN */
        vmspace = obj_alloc(TYPE_VMSPACE, sizeof(struct vmspace));
        /* LAB 3 TODO END */
        ......
}
```

`create_root_cap_group`：对根进程的`cap_group`和`vmspace`进行初始化即可。

```
struct cap_group *create_root_cap_group(char *name, size_t name_len)
{
        struct cap_group *cap_group;
        struct vmspace *vmspace;
        int slot_id;
        /* LAB 3 TODO BEGIN */
        cap_group = obj_alloc(TYPE_CAP_GROUP, sizeof(struct cap_group));
        /* LAB 3 TODO END */
        BUG_ON(!cap_group);
        /* LAB 3 TODO BEGIN */
        cap_group_init(cap_group, sizeof(struct cap_group), ROOT_PID);
        slot_id = cap_alloc(cap_group, cap_group, 0);
        /* LAB 3 TODO END */
        BUG_ON(slot_id != CAP_GROUP_OBJ_ID);
        /* LAB 3 TODO BEGIN */
        vmspace = obj_alloc(TYPE_VMSPACE, sizeof(struct vmspace));
        /* LAB 3 TODO END */
        BUG_ON(!vmspace);

        /* fixed PCID 1 for root process, PCID 0 is not used. */
        /* LAB 3 TODO BEGIN */
        vmspace_init(vmspace);
        vmspace->pcid = ROOT_PCID;
        slot_id = cap_alloc(cap_group, vmspace, 0);
        /* LAB 3 TODO END */
        ......
}
```

> 练习 3: 在`kernel/object/thread.c`中完成`load_binary`函数，将用户程序 ELF 加载到刚刚创建的进程地址空间中。

答：我们取出ELF文件中的各个段，将其拷贝到对应的内存空间中。拷贝时我们按照`p_filesz`的长度，直接采用`memcpy`即可。

```
/* load binary into some process (cap_group) */
static u64 load_binary(struct cap_group *cap_group, struct vmspace *vmspace,
                       const char *bin, struct process_metadata *metadata)
{
        ......
        /* load each segment in the elf binary */
        for (i = 0; i < elf->header.e_phnum; ++i) {
                pmo_cap[i] = -1;
                if (elf->p_headers[i].p_type == PT_LOAD) {
                        seg_sz = elf->p_headers[i].p_memsz;
                        p_vaddr = elf->p_headers[i].p_vaddr;
```

```
                        /* LAB 3 TODO BEGIN */
                        u64 vm_start = ROUND_DOWN(p_vaddr, PAGE_SIZE);
                        u64 vm_end = ROUND_UP(p_vaddr + seg_sz, PAGE_SIZE);
                        seg_map_sz = vm_end - vm_start;
                        pmo_cap[i] = create_pmo(
                                seg_map_sz, PMO_DATA, cap_group, &pmo);

                        char *elf_start = bin + elf->p_headers[i].p_offset;
                        char *pmo_start = (char *)phys_to_virt(pmo->start)
                                        + p_vaddr - vm_start;
                        memcpy(pmo_start, elf_start, elf->p_headers[i].p_filesz);
                        flags = PFLAGS2VMRFLAGS(elf->p_headers[i].p_flags);
                        ret = vmspace_map_range(
                                vmspace, vm_start, seg_map_sz, flags, pmo);
                        /* LAB 3 TODO END */
                        BUG_ON(ret != 0);
                }
        }
        ......
}
```

> 练习 4: 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。

答：根据注释提示，后缀分别为`el1t`，`el1h`，`el0_64`，`el0_32`，因此按照要求填写即可。

异常向量表如下：

```
/* 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 */
```

根据要求，对于`sync_el1h`类型的异常，跳转至`handle_entry_c`；对于其他类型的异常，跳转至`unexpected_handler`。因此函数跳转如下：

```
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 */
        bl handle_entry_c
       /* LAB 3 TODO END */
       exception_exit /* Lab4: Do not unlock! */
```

> 练习 5: 填写`kernel/arch/aarch64/irq/pgfault.c`中的`do_page_fault`，需要将缺页异常转发给`handle_trans_fault`函数。

答：

```
void do_page_fault(u64 esr, u64 fault_ins_addr)
{
        ......
        switch (fsc) {
        case DFSC_TRANS_FAULT_L0:
        case DFSC_TRANS_FAULT_L1:
        case DFSC_TRANS_FAULT_L2:
        case DFSC_TRANS_FAULT_L3: {
                int ret;
                /* LAB 3 TODO BEGIN */
                handle_trans_fault(current_thread->vmspace, fault_addr);
                /* LAB 3 TODO END */
                ......
               }
        }
}
```

> 练习 6: 填写`kernel/mm/pgfault_handler.c`中的`handle_trans_fault`，实现`PMO_SHM`和`PMO_ANONYM`的按需物理页分配。

答：我们模仿`kernel/object/memory.c`的写法，进行分配页。首先调用`get_page_from_pmo`，寻找PMO中对应的物理页。`pa==0`时，物理页未分配，我们分配物理页后，使用`map_range_in_pgtbl`，增加页表映射；`pa!=0`时，物理页已分配。根据注释，`PMO_ANONYM`和`PMO_SHM`采用相同的方法处理，直接修改页表映射即可。

```
int handle_trans_fault(struct vmspace *vmspace, vaddr_t fault_addr)
{
        ......
        switch (pmo->type) {
        case PMO_ANONYM:
        case PMO_SHM: {
                ......
                /* LAB 3 TODO BEGIN */
                pa = get_page_from_pmo(pmo, index);
                /* LAB 3 TODO END */
                if (pa == 0) {
                        /* Not committed before. Then, allocate the physical
                         * page. */
                        /* LAB 3 TODO BEGIN */
                        vaddr_t kva = (vaddr_t)get_pages(0);
                        pa = virt_to_phys((void *)kva);
                        memset((void *)kva, 0, PAGE_SIZE);
                        commit_page_to_pmo(pmo, index, pa);
                        if (map_range_in_pgtbl(vmspace->pgtbl,
                                               fault_addr,
                                               pa,
                                               PAGE_SIZE,
                                               vmr->perm)
                            < 0) {
                                return -ENOMAPPING;
                        }
                        /* LAB 3 TODO END */
                } else {
                        /* LAB 3 TODO BEGIN */
                        ret = map_range_in_pgtbl(vmspace->pgtbl,
                                                 fault_addr,
                                                 pa,
                                                 PAGE_SIZE,
                                                 vmr->perm);
                        /* LAB 3 TODO END */
                }
```

> 练习 7: 按照前文所述的表格填写 `kernel/arch/aarch64/irq/irq_entry.S` 中的 `exception_enter` 与 `exception_exit`，实现上下文保存的功能。

答：此处实现了上下文保存。我们可以从`kernel/include/aarch/aarch64/machine/registers.h`得知寄存器的编号、数量，每两个一对保存即可。

```
.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    x11, sp_el0
       mrs    x12, elr_el1
       mrs    x13, spsr_el1
       /* LAB 3 TODO BEGIN */
        stp     x30, x11, [sp, #16 * 15]
        stp     x12, x13, [sp, #16 * 16]
       /* LAB 3 TODO END */
.endm

.macro exception_exit
       /* LAB 3 TODO BEGIN */
        ldp     x12, x13, [sp, #16 * 16]
        ldp     x30, x11, [sp, #16 * 15]
       /* LAB 3 TODO END */
       msr    sp_el0, x11
       msr    elr_el1, x12
       msr    spsr_el1, x13
       /* LAB 3 TODO BEGIN */
        ldp     x28, x29, [sp, #16 * 14]
        ldp     x26, x27, [sp, #16 * 13]
        ldp     x24, x25, [sp, #16 * 12]
        ldp     x22, x23, [sp, #16 * 11]
        ldp     x20, x21, [sp, #16 * 10]
        ldp     x18, x19, [sp, #16 * 9]
        ldp     x16, x17, [sp, #16 * 8]
        ldp     x14, x15, [sp, #16 * 7]
        ldp     x12, x13, [sp, #16 * 6]
        ldp     x10, x11, [sp, #16 * 5]
        ldp     x8, x9, [sp, #16 * 4]
        ldp     x6, x7, [sp, #16 * 3]
```

```
        ldp     x4, x5, [sp, #16 * 2]
        ldp     x2, x3, [sp, #16 * 1]
        ldp     x0, x1, [sp, #16 * 0]
        add     sp, sp, #ARCH_EXEC_CONT_SIZE
       /* LAB 3 TODO END */
       eret
.endm
```

> 练习 8: 填写 `kernel/syscall/syscall.c` 中的 `sys_putc`、`sys_getc`，`kernel/object/thread.c` 中的 `sys_thread_exit`，`libchcore/include/chcore/internal/raw_syscall.h` 中的 `__chcore_sys_putc`、`__chcore_sys_getc`、`__chcore_sys_thread_exit`，以实现 `putc`、`getc`、`thread_exit` 三个系统调用。

答:

`sys_putc` / `sys_getc`：分别调用`uart_send`和`uart_recv`即可。

```
void sys_putc(char ch)
{
        /* LAB 3 TODO BEGIN */
        uart_send(ch);
        /* LAB 3 TODO END */
}

u32 sys_getc(void)
{
        /* LAB 3 TODO BEGIN */
        return uart_recv();
        /* LAB 3 TODO END */
}
```

`sys_thread_exit`：我们使用`thread_deinit`销毁当前thread即可，在调用`thread_deinit`前，先要将`thread_ctx->state`和`thread_ctx->thread_exit_state`分别设成`TS_EXIT`和`TE_EXITED`。

```
void sys_thread_exit(void)
{
        /* LAB 3 TODO BEGIN */
        current_thread->thread_ctx->state = TS_EXIT;
        current_thread->thread_ctx->thread_exit_state = TE_EXITED;
        thread_deinit(current_thread);
        /* LAB 3 TODO END */
        printk("Lab 3 hang.\n");
        while (1) {
        }
        /* Reschedule */
        sched();
        eret_to_thread(switch_context());
```

```
}
```

以下几个函数，根据参数的数量不同，我们分别调用`__chcore_syscall0`和`__chcore_syscall1`进行处理。

`__chcore_sys_putc` / `__chcore_sys_getc`：调用`internal/syscall_num.h`中的`__CHCORE_SYS_putc` / `__CHCORE_SYS_getc`。

```
static inline void __chcore_sys_putc(char ch)
{
        /* LAB 3 TODO BEGIN */
        __chcore_syscall1(__CHCORE_SYS_putc, ch);
        /* LAB 3 TODO END */
}
static inline u32 __chcore_sys_getc(void)
{
        /* LAB 3 TODO BEGIN */
        return __chcore_syscall0(__CHCORE_SYS_getc);
        /* LAB 3 TODO END */
}
```

`__chcore_sys_thread_exit`：调用`internal/syscall_num.h`中的`__CHCORE_SYS_thread_exit`。

```
static inline void __chcore_sys_thread_exit(void)
{
        /* LAB 3 TODO BEGIN */
        __chcore_syscall0(__CHCORE_SYS_thread_exit);
        /* LAB 3 TODO END */
}
```