实验 3：进程与线程

薛翔元 (521030910387)

> 练习题 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。
>
> 提示:
> 可以阅读 `kernel/object/capability.c` 中各个与 cap 机制相关的函数以及参考文档。

对于 `sys_create_cap_group` 函数，我们依次完成 `cap_group` 的分配，`cap_group` 的初始化，`vmspace` 的分配即可。

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

```
/* LAB 3 TODO BEGIN */
/* initialize cap group */
BUG_ON(cap_group_init(new_cap_group, BASE_OBJECT_NUM, args.badge));
/* LAB 3 TODO END */
```

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

对于 `create_root_cap_group` 函数，我们依次完成 `cap_group` 的分配，`cap_group` 的初始化，`vmspace` 的分配，`slot` 的分配即可。

```
/* LAB 3 TODO BEGIN */
cap_group = (struct 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, ROOT_CAP_GROUP_BADGE);
/* LAB 3 TODO END */
```

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

```
/* LAB 3 TODO BEGIN */
slot_id = cap_alloc(cap_group, vmspace);
/* LAB 3 TODO END */
```

> 练习题 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。
>
> 提示:
>
> - 程序头可以参考`kernel/object/thread_env.h`。
> - 内存分配操作使用 `create_pmo`，可详细阅读`kernel/object/memory.c`了解内存分配。
> - 本练习并无测试点，请确保对 elf 文件内容读取及内存分配正确。否则有可能在后续切换至用户态程序运行时出错。

我们首先从每个 `program header` 中读取 `offset`、`vaddr`、`filesz` 和 `memsz` 四个信息。

```
/* 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 */
```

然后根据 `memsz` 在 `root_cap_group` 内分配指定大小的 `segment_pmo`。

```
/* LAB 3 TODO BEGIN */
ret = create_pmo(memsz,
                 PMO_DATA,
                 root_cap_group,
                 0,
                 &segment_pmo);
/* LAB 3 TODO END */
```

接着根据 `offset` 和 `filesz` 将 ELF 文件加载到内存中。

```
/* LAB 3 TODO BEGIN */
/* Copy elf file contents into memory*/
memcpy(phys_to_virt(segment_pmo->start),
        (void *)((unsigned long)&binary_procmgr_bin_start
                + ROOT_BIN_HDR_SIZE + offset),
        filesz);
memset(phys_to_virt(segment_pmo->start + filesz),
        0,
        segment_pmo->size - filesz);
/* LAB 3 TODO END */
```

最后根据 `flags` 构造 `vmr_flags`，结合 `vaddr` 创建页表映射。

```
/* 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 */
```

> 练习题 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。

根据传入参数，依次修改线程上下文中 `SP_EL0`、`ELR_EL1` 和 `SPSR_EL1` 寄存器的值即可。

```
/* 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。

在 ChCore 内核启动后，依次初始化 `uart` 串口、`mm` 内存管理单元、`interrupt` 系统中断、`timer` 定时器、`pmu` 性能监控单元和 `sched` 调度器，接着调用 `create_root_thread` 函数创建用户进程 `procmgr` 以及相应用户线程，然后调用 `sched` 函数进行首次调度，选中首个用户线程，最后调用 `switch_context` 函数进行线程上下文的切换，其返回的 `thread_ctx` 地址传入 `eret_to_thread` 函数，通过调用 `__eret_to_thread` 汇编函数的方式，将 `thread_ctx` 地址写入 `sp` 寄存器并调用 `eret` 指令返回用户态，至此 ChCore 内核完成了首次从内核态向用户态的切换。

> 练习题 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。

我们根据 `irq_entry.S` 中的注释填写异常向量表，利用汇编进行语句跳转。

```
/* 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 */
```

对于 `irq_el1t`、`fiq_el1t`、`fiq_el1h`、`error_el1t`、`error_el1h`、`sync_el1t`，利用 `unexpected_handler` 函数处理异常。

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

对于 `sync_el1h`，利用 `handle_entry_c` 函数处理异常，并将其返回值写入 `ESR_EL1` 寄存器。

```
/* jump to handle_entry_c, store the return value as the ELR_EL1 */
bl handle_entry_c
msr elr_el1, x0
/* LAB 3 TODO END */
```

> 练习题 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。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
>
> ```
> Hello userland!
> ```

对于 `exception_enter` 函数，我们按照指定的寄存器结构，依次将 `x0` 到 `x30` 以及 `SP_EL0`、`ELR_EL1` 和 `SPSR_EL1` 寄存器的值存入栈中。

```
.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 */

.endm
```

对于 `exception_exit` 函数，我们依次将 `SP_EL0`、`ELR_EL1` 和 `SPSR_EL1` 以及 `x0` 到 `x30` 寄存器的值从栈中还原。

```
.macro  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 */

    eret
.endm
```

对于 `switch_to_cpu_stack` 函数，我们将 `TPIDR_EL1` 寄存器所保存的地址加上偏移量 `OFFSET_LOCAL_CPU_STACK` 即可正确读取 `cpu_stack` 所在地址。

```
.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
```

> 思考题 7: 尝试描述 `printf` 如何调用到 `chcore_stdout_write` 函数。
>
> 提示：`chcore_write` 中使用了文件描述符，`stdout` 描述符的设置在`user/chcore-libc/musl-libc/src/chcore-port/syscall_dispatcher.c` 中。

`printf` 函数将 `stdout` 文件指针作为参数传入 `vfprintf` 函数，`vfprintf` 函数调用 `stdout` 的 `write` 操作，该操作定义为 `__stdout_write` 函数，`__stdout_write` 函数调用 `__stdio_write` 函数，`__stdio_write` 函数将 `stdout` 的文件描述符传入 `SYS_writev` 系统调用，从而间接调用 `chcore_writev` 函数，`chcore_writev` 函数调用 `chcore_write` 函数，`chcore_write` 函数调用相应文件描述符的 `write` 操作，其中 `stdout` 是 1 号文件描述符，其 `fd_op` 指向 `stdout_op` 结构体，其中

`write` 操作即为 `chcore_stdout_write` 函数，至此 `printf` 函数成功调用 `chcore_stdout_write` 函数。

```
printf -> vfprintf -> __stdout_write -> __stdio_write -> SYS_writev -> chcore_writev
-> chcore_write -> 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。

利用 `chcore_syscall2` 函数，将 `buffer` 和 `size` 作为参数传入 `CHCORE_SYS_putstr` 系统调用即可。

```
/* LAB 3 TODO BEGIN */
chcore_syscall2(CHCORE_SYS_putstr, (uintptr_t)buffer, size);
/* LAB 3 TODO END */
```

> 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。

我们编写一个最小用户程序 `hello_chcore.c`，用于打印 `Hello ChCore!`。

```
#include <stdio.h>

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

利用 ChCore 的编译工具链编译 `hello_chcore.c`，并将二进制文件放入 `ramdisk` 目录下。

```
./build/chcore-libc/bin/musl-gcc hello_chcore.c -o hello_chcore.bin
mv hello_chcore.bin ./ramdisk
```

至此，运行 `make qemu` 可以正常进入 `shell`，运行 `make grade` 可以通过所有测试。

```
 ______     __  __     ______     __  __     ______     __         __
/\  ___\   /\ \_\ \   /\  ___\   /\ \_\ \   /\  ___\   /\ \       /\ \
\ \ \____  \ \  __ \  \ \___  \  \ \  __ \  \ \  __\   \ \ \____  \ \ \____
 \ \_____\  \ \_\ \_\  \/\_____\  \ \_\ \_\  \ \_____\  \ \_____\  \ \_____\
  \/_____/   \/_/\/_/   \/_____/   \/_/\/_/   \/_____/   \/_____/   \/_____/

Welcome to ChCore shell!
```

```
===============
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
```