

Report on Lab 3

- 李逸岩 519021911103
- lyy0627@sjtu.edu.cn.

1

0号进程 `init task` 会执行如内存, 页表, 信号量等初始化, CPU运行在内核态。初始化完成后会启动一个 `init` 进程(1号进程)和 `kthread` 进程(2号进程)。然后零号进程变为IDEL进程。

接下来运行1号进程, 会加载ELF文件运行, 保存寄存器, 设置栈指针寄存器, 设定异常寄存器等以进入用户态。还会运行Shell等第一个用户线程。进入用户态之后, 还需要进行用户态初始化, 运行线程管理线程(2号进程)。

在2号进程初始化线程管理调度完成后即成为守护进程, 让出CPU. 则只有Shell/登录等用户进程等待调度。然后就进入了用户态的用户进程了。

2

数据结构关系如下:

- `object`, 是对内核资源的抽象, 包括 `ref-count`, `type` 等。 `opaque` 字段是0长度数组, 好处是可以根据分配Struct的大小动态伸缩, 保持8对齐。 `object` 可以是: 进程, 线程, IPC, PMO, VMspace, Semaphore等。分配 `object` 时, 指定类型和 `opaque` 的大小, 就会自动分配出 `opaque + meta` 的大小, 并且返回 `opaque` .
- `object slot`, 是 `object` 插槽。 `slot id` 是下标, 还包括权限, 有效位, 所处进程等。 `slot table` 是插槽数组, 还包括 `bit map` 和大小。

`thread` 和 `cap_group` 在实验文档中介绍了。下面是一些工具函数:

- `cap_alloc`: `container_of` 宏获得 `object` 的头部。分配 `slot id` 后组装 `slot` 并且插入到 `cap_group` 中。

对于 `cap_group_init`, 初始化 `thread_list`, `slot_table`. 对 `sys_create_cap_group` 要分配 `object`. 初始化之后作为 `cap` 分配给当前进程, `cap_group` 本身作为新进程的第一个资源, 初始化 `vmSPACE` 作为新进程的第二个资源。对于 `create_root_cap_group`, 无需作为 `object` 分配给当前进程。

主要解决的BUG就是搞错宏的名字了 `TYPE_VMSpace` & `VMSpace_OBJ_ID`, 导致运行过程中出现了Page Fault. 解决后, 获得了该部分的10分, 并顺利运行到下一函数。

3

每个 `section` 都需要分配一个 `pobject`. 按照代码提示, 还需要记录 `slot id` 以在失败后及时释放资源。此外, 要注意页对齐问题, 无论起始地址还是终止地址都应该保持页对齐, 实际映射的大小为对齐后的首尾地址差。

主要需要理解 `vmregion`. 其是 `vmSpace` 的链成员, 和 `pmo` 一一对应。分配一个 `pmo` 就要分配一个 `vmregion` 在 `vmSpace` 中。一开始疑惑于为什么不直接把 `pmo` 映射给实际的 `vaddr`, 而是先 `kmalloc` 一段。后来在 `page_fault_handler` 中才解决了这个疑惑。即加载ELF的时候 `page table` 对应的是另一个进程, 而我们只能加载到一段物理内存中, 并且作为 `pobject` 将内核资源赋给线程。然后如果是 `PMO_DATA/DATA_NOCACHE` 类型, 只需要在线程自己的页表上映射 `vaddr` 到 `pmo_start` 即可解决。那么当切换到该线程执行, 已经映射好了。至于延迟映射的情况, 我们将在T6解决。

- 分配pmo, 记录 slot id.
- 将ELF段拷贝到实际物理地址(使用当前页表翻译)。
- 将pmo和 thread vmSPACE vaddr 映射好。

在完成这个部分之后, [ChCore] create initial thread done on 0 创建了第一个线程。

4

需要使用到 macro exception_entry. 作用是帮助0x80对齐。然后错误定义位于 irq_entry.h. 顺序严格按照表填写, h/t/el0_32/el0_64 后缀含义在注释和文档中已经标注。然后需要填写处理函数。按照文档调用即可。

5

将 vmSPACE 和 fault_addr 作为参数, 调用 handle_trans_fault 后, 返回值存在 ret 中。

6

正如T3所言, 线程拥有的虚拟地址空间被抽象为 vmSPACE, 其成员为 vmregion. 在PF发生后, 应该查询虚拟地址空间找到对应的 vmregion. 然后查询到对应的pmo. 我们只需要处理延迟分配的 PMO_ANONYM, PMO_SHM. 而立即分配的认为不应该发生缺页情况(在未实现换页的前提下)。

然后我们需要查询延迟分配的是否曾经被分配过, 这需从pmo中查询页下标对应的PA. 如果为0, 说明未被分过。则分配物理页, memset, 记录到pmo的对应index上, 然后将这段物理页映射到当前页表的 fault_addr 上。如果已经分配过物理页(比如Shared情况), 则映射到当前页表的 fault_addr 上。

这里bug主要来源于一个误区。 vmregion.start 表示这一段的起始虚拟地址, 而 fault_addr 只是实际上缺页的虚拟地址, 我们只需要换上 fault_adrr 所在页即可。因此 map 的 va 参数应该填写实际缺页地址(Round down), 而不是 vmr.start.

7

在系统调用前, 需要保存通用寄存器以及和异常级别有关的 sp_el0, elr_el1, spsr_el1. 系统调用退出时, 应该恢复这些寄存器。

8

在 raw_syscall 中, 会根据参数的个数调用系统调用, 第一个参数为系统调用类型, 然后 chcore 会查询系统调用表, 调用 syscall.c 或其他文件中的系统调用函数。在kernel中, 对于输入输出使用之前实现过的 uart 来完成, 对于线程退出则将 thread 的状态设为EXIT即可。

完成第8题后, make_grade 能够获得满分。

```
Grading lab 3...(may take 90 seconds)
qemu-system-aarch64: terminating on signal 15 from pid 131086 ()
qemu-system-aarch64: terminating on signal 15 from pid 132372 (killall)
qemu-system-aarch64: terminating on signal 15 from pid 133573 ()
qemu-system-aarch64: terminating on signal 15 from pid 134833 ()
qemu-system-aarch64: terminating on signal 15 from pid 136053 (killall)
qemu-system-aarch64: terminating on signal 15 from pid 137358 ()
GRADE: Cap create pretest: 10
GRADE: Bad instruction 1: 10
GRADE: Bad instruction 2: 10
GRADE: Fault: 20
GRADE: User Application: 20
GRADE: Put, Get and Exit: 30
Score: 100/100
```

9

使用LRU换页策略。划定一块区域为实际内存，使用链表作为LRU队列，每次映射都将其提到队首。当实际内存满需要换页，则释放队尾的页面并且换页，无论是否脏页，换出时都 `malloc` 一段内存视为磁盘写回，以便捷的保证LRU的正确性。

本实现为了能在 `pgfault_handler.c` 中完成功能，只实现了对延迟分配的内存的LRU换页。因为一个物理页可能被多个虚拟页映射，因此为每个实际内存页都分配了 `vmregion` 链表，储存指向该页的 `vmregion` 和对应的 `pmo index`。

如果实际内存页被驱逐，除写回外还要取消映射，然后将作为磁盘的内存地址写入 `pmo`。由于取消映射，下次仍将 **Page Fault** 并发现 `pa != 0` 且 `pa` 不在实际内存范围内，即可判断此页面被驱逐。此时的行为是获得(可能伴随驱逐老页面)实际内存页，更新所有指向“用作Disk的内存的”的 `vmregion` 的 `PMO` 到实际内存页，并且重新映射。正确性的关键是绝不映射非实际内存范围的地址到页表。如果在实际内存范围内，则简单映射并更新 `vmregion` 链表。如果发现 `pa == 0`，则索取实际内存页，清空并映射。

找到空闲内存的方式是遍历内存页链表，每个节点都含有其 `index`，`real_memory + PAGE_SIZE * index` 即为实际内存。

在 `pagefault_handler.c` 中，被 `#if LRU_TEST` 包裹的程序段是LRU Replacer的实现。将对应的宏改为 `True` 之后即可测试正确性。本地测试如下：

- LRU POOL SIZE == 1，调用了驱逐和恢复的程序段，能顺利进入 `user land` 并挂起；
- LRU POOL SIZE > 1，未调用驱逐程序段，也可正常进入 `user land`。

```
[INFO] [ChCore] uart init finished
[INFO] physmem_map: [0xc0000, 0x3d000000)
[INFO] [ChCore] mm init finished
[INFO] [ChCore] interrupt init finished
[INFO] Cap_create Pretest Ok!
[INFO] [ChCore] create initial thread done on 0
Test: Test: Successfully map for pa not 0
Test: Test: Successfully map
Success to user land!

Back to kernel.
Lab 3 hang.
```