Lab4 实验报告

零、实验目的

1. 掌握系统调用的概念及流程
2. 实现进程间通信机制
3. 实现 fork 函数
4. 掌握页写入异常的处理流程

在用户态下，用户进程不能访问系统的内核空间，也就是说它一般不能存取内核使用的内存数据，也不能调用内核函数，这一点是由体系结构保证的。然而，用户进程在特定的场景下往往需要执行一些只能由内核完成的操作，如操作硬件、动态分配内存，以及与其他进程进行通信等。允许在内核态执行用户程序提供的代码显然是不安全的，因此操作系统设计了一系列内核空间中的函数，当用户进程需要进行这些操作时，会引发特定的异常以陷入内核态，由内核调用对应的函数，从而安全地为用户进程提供受限的系统级操作，我们把这种机制称为系统调用。
在 Lab4 中，我们需要实现上述的系统调用机制，并在此基础上实现进程间通信（IPC）机制和一个重要的进程创建机制 fork。在 fork 部分的实验中，我们会介绍一种被称为写时复制 （COW）的特性，以及与其相关的页写入异常处理。

一、思考题

Thinking 4.1

> 思考并回答下面的问题：
>
> - 内核在保存现场的时候是如何避免破坏通用寄存器的?

查看 `include/stackframe.h` 的 `SAVE_ALL` 宏：
其中大量使用了 `k0` 寄存器，使用 `k0` 保存了 CP0 的信息后，将各个通用寄存器的值压栈。

> - 系统陷入内核调用后可以直接从当时的 `$a0-$a3` 参数寄存器中得到用户调用 `msyscall` 留下的信息吗?

可以。从用户函数 `syscall_*()` 到内核函数 `sys_*()` 时，`$a0-$a3` 并没有改变。

> - 我们是怎么做到让 `sys` 开头的函数“认为”我们提供了和用户调用 `msyscall` 时同样的参数的?

在 `do_syscall` 中对相应的参数进行了保存，然后作为函数参数传递给 `sys_*`。

> - 内核处理系统调用的过程对 `Trapframe` 做了哪些更改？这种修改对应的用户态的变化是什么?

修改了 `Trapframe` 中 `epc` 的值，即变成原来的 +4，这样返回时从下一条指令开始执行。其次将返回值存入了 `$v0`，也就是 `reg[2]`。
这里要对上一次实验报告的 CP0 部分画的图进行纠正，第 29 号寄存器栈寄存器并非位于 CP0，而

是属于通用寄存器。`Trapframe` 中保存了 32 个通用寄存器、`hi, lo` 寄存器和部分 CP0 的寄存器（`status, badvaddr, cause, epc`）

Thinking 4.2

> 思考 `envid2env` 函数: 为什么 `envid2env` 中需要判断 `e->env_id != envid` 的情况？如果没有这步判断会发生什么情况？

追根溯源看看 `env_id` 是怎么打造的：

```
91 u_int mkenvid(struct Env *e) {
92         static u_int i = 0;
93         return ((++i) << (1 + LOG2NENV)) | (e - envs);
94 }
```

所以说 `e - envs` 这部分应该是有可能重叠的，应该是为了方便从数组下标直接获取进程控制块。而且可能有控制块本来错误未被销毁或者同步互斥问题，所以检查一下比较保险。

Thinking 4.3

> 思考下面的问题，并对这个问题谈谈你的理解：请回顾 `kern/env.c` 文件中 `mkenvid()` 函数的实现，该函数不会返回 0，请结合系统调用和 IPC 部分的实现与 `envid2env()` 函数的行为进行解释。

`mkenvid()` 保证了返回值不是 0，而 `envid2env()` 当 `envid == 0` 时返回 `curenv` 。

0 号进程是内核进程，是系统初始化时创建的第一个进程。它并不执行任何的实际程序代码，只是参与进程的切换和调度的管理。因此创建一个新的 `envid` 时，保证其与 0 号进程区分开。

Thinking 4.4

> 关于 `fork` 函数的两个返回值，下面说法正确的是：
> A、`fork` 在父进程中被调用两次，产生两个返回值
> B、`fork` 在两个进程中分别被调用一次，产生两个不同的返回值
> C、`fork` 只在父进程中被调用了一次，在两个进程中各产生一个返回值
> D、`fork` 只在子进程中被调用了一次，在两个进程中各产生一个返回值

答案是 C。

Thinking 4.5

> 我们并不应该对所有的用户空间页都使用 `duppage` 进行映射。那么究竟哪些用户空间页应该映射，哪些不应该呢？请结合 `kern/env.c` 中 `env_init` 函数进行的页面映射、`include/mmu.h` 里的内存布局图以及本章的后续描述进行思考。

`0~USTACKTOP` 需要使用 `duppage` 进行映射。
`USTACKTOP~UTOP` 之间的 `user exception stack` 用于进行页写入异常，不会在处理 COW 异常时调用 `fork()` ，因此不共享。
`UTOP` 以上的页面的内存与页表是所有进程共享的，用户进程无权限访问，无需 `duppage` 。

Thinking 4.6

> 在遍历地址空间存取页表项时你需要使用到 `vpd` 和 `vpt` 这两个指针，请参考 `user/include/lib.h` 中的相关定义，思考并回答这几个问题：
>
> - `vpt` 和 `vpd` 的作用是什么？怎样使用它们？

`vpd` 是页目录基地址，加上页目录偏移即可指向对应的页目录项，即 `(*vpd) + (va >> 22)`。
`vpt` 是页表基地址，加上页表项偏移即可指向对应的页表项，即 `(*vpt) + (va >> 12)`

> - 从实现的角度谈一下为什么进程能够通过这种方式来存取自身的页表？

```
11 #define vpt ((const volatile Pte *)UVPT)
12 #define vpd ((const volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT)))
```

> - 它们是如何体现自映射设计的？

自映射，即页目录本身也是一个页表，它们在同一块地址空间。

> - 进程能够通过这种方式来修改自己的页表项吗？

不能，该部分只读，只能通过内核态修改。

Thinking 4.7

> 在 `do_tlb_mod` 函数中，你可能注意到了一个向异常处理栈复制 `Trapframe` 运行现场的过程，请思考并回答这几个问题：
>
> - 这里实现了一个支持类似于”异常重入“的机制，而在什么时候会出现这种“异常重入”？

异常重入机制允许操作系统在处理一个异常的同时能够响应并处理其他异常。这保证了不会在处理一个异常时被另一个异常中断，导致系统崩溃或数据丢失。

> - 内核为什么需要将异常的现场 `Trapframe` 复制到用户空间？

这个函数中， `EPC` 被设置为了用户态的异常处理函数 `env_user_tlb_mod_entry` ，因为我们的 MOS 系统遵循微内核的思想，该异常的处理不在内核态而是用户态，因此需要将异常的现场 tf 复制到用户空间。

Thinking 4.8

> 在用户态处理页写入异常，相比于在内核态处理有什么优势？

1. 符合微内核设计，使得内核的设计更加小巧。
2. 用户与用户间相互独立，不互相影响，如果某个进程对于tlb的处理出现问题不会影响其他用户
3. 提高性能，减少额外的上下文切换

Thinking 4.9

> 请思考并回答以下几个问题：

> - 为什么需要将 `syscall_set_tlb_mod_entry` 的调用放置在 `syscall_exofork` 之前？

执行 `syscall_exofork` 后父子进程将各自执行各自的进程，子进程涉及到对 COW 的修改，会触发写入异常，而 COW 中断依赖于 `syscall_set_tlb_mod_entry`。

> - 如果放置在写时复制保护机制完成之后会有怎样的效果？

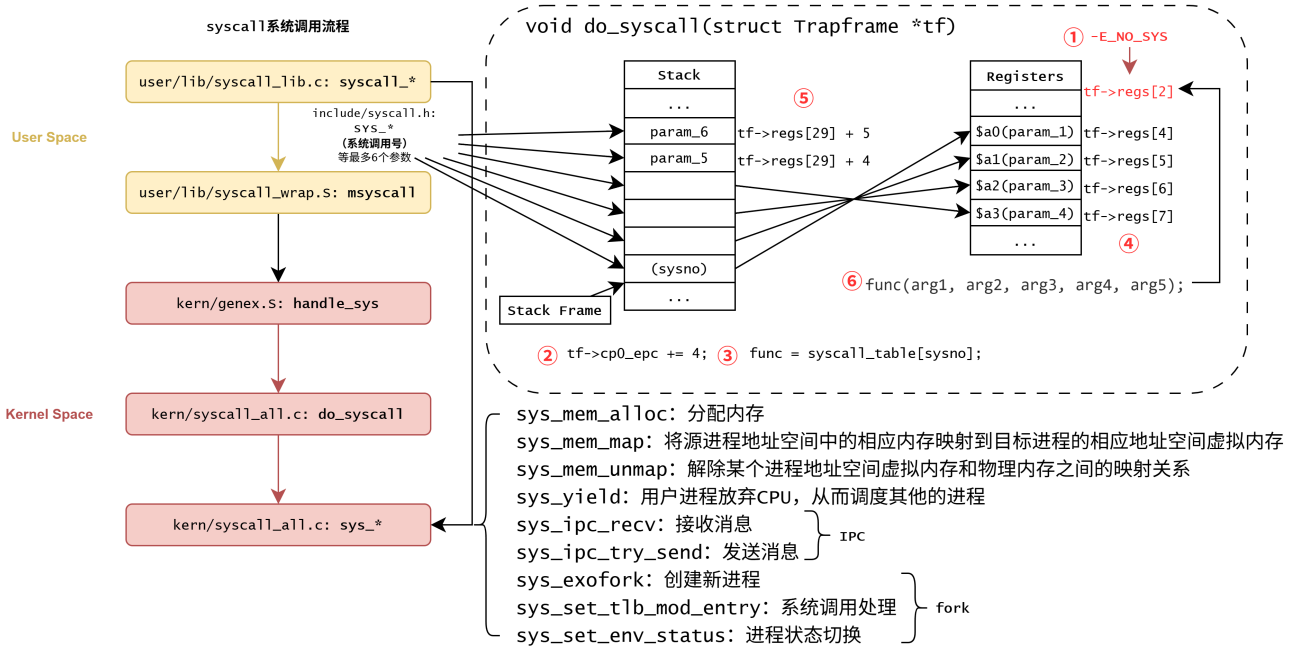
写时复制保护机制需要调用 `syscall_mem_map`，如果此时缺页异常则不能及时响应。

二、难点分析

项目结构

```
.
├── include
├── init
├── kern
│   ├── syscall_all.c // lab4：内核态系统调用
│   └── tlbex.c
├── kernel.lds
├── lib
├── Makefile
├── tests
├── tools
└── user
    ├── lib
    │   ├── fork.c // lab4：fork 函数
    │   ├── ipc.c // lab4：IPC 机制
    │   ├── syscall_lib.c
    │   └── syscall_wrap.S // lab4：msyscall函数
    └── user.lds
```

syscall 系统调用



- `int sys_mem_alloc(u_int envid, u_int va, u_int perm)`：为 `envid` 对应的进程控制块及虚拟地址 `va` 分配物理内存。
- `int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm)`：将 `srcid` 对应的进程控制块地址空间的 `srcva` 对应内存映射到 `dstid` 对应的进程控制块地址空间的 `dstva` 对应内存。
- `int sys_mem_unmap(u_int envid, u_int va)`：解除 `envid` 对应的进程控制块地址空间的 `va` 对应虚拟内存和物理内存的映射关系。
- `void __attribute__((noreturn)) sys_yield(void)`：实现进程对 CPU 放弃。

IPC 机制

- `int sys_ipc_recv(u_int dstva)`：接收来自 `dstva` 的消息。首先将 `env_ipc_recving` 设为 1，接着 `env_ipc_dstva` 赋值，阻塞当前进程为 `ENV_NOT_RUNNABLE`，最后 `yield`（`schedule(1)`）。
- `int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm)`：将发送方的 `srcva` 映射到接收方的 `dstva`，传递 `value`，`perm`，接收方为 `envid`。找到接收方进程 `envid`，如果对方可接收则成功，否则不成功；接着将 `value`、`perm` 填入，将 `env_ipc_from` 设为发送方（`curenv`），将 `env_ipc_recving` 设为 0，`env_status` 设为 `ENV_RUNNABLE`；最后找 `srcva` 对应的物理页面，映射到 `dstva`。（当 `srcva` 为 0 时不传递物理页面）

fork

协助 fork 完成的函数

内核态：

- `int sys_exofork(void)`：创建子进程，同步父进程信息。复制当前进程的 `tf` 到子进程的进程控制块，子进程的 `$v0` 寄存器设为 0，子进程设为 `ENV_NOT_RUNNABLE`，子进程同步父进程的 `pri`。
- `int sys_set_tlb_mod_entry(u_int envid, u_int func)`：注册 `envid` 对应进程控制块的页写入异常函数为 `func`。

- `int sys_set_env_status(u_int envid, u_int status)`：设置子进程为可以运行的状态，加入可调度进程队列。

用户态：

- `static void duppage(u_int envid, u_int vpn)`：父进程将地址空间中需要与子进程共享的页面映射给子进程。对于只读页面（不具有 `PTE_D` 权限位），按照只读权限映射给子进程；对于写时复制页面（具有 `PTE_COW` 权限位），这是之前 `duppage` 的结果，且本次 `fork` 前未被写过；对于共享页面（具有 `PTE_LIBRARY` 权限位），在父子进程中映射到相同的物理页，对其修改的结果相互可见；对于可写页面（具有 `PTE_D` 权限位，且不是上面任意一种），在父子进程都使用 `PTE_COW` 权限位进行保护。
- `void do_tlb_mod(struct Trapframe *tf)`：设置保存的现场中 `EPC` 的值。
- `static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf)`。

fork 函数

`int fork(void)`：使用 `syscall_exofork` 创建一个子进程，遍历父进程地址空间进行 `duppage`，用 `syscall_set_tlb_mod_entry` 设置子进程的异常处理函数确保页写入异常可以被正常处理，用 `syscall_set_env_status` 设置子进程为可调度。

三、实验体会

整个 lab4 围绕着系统调用进行，包括 fork 机制依托于系统调用，理清了用户态和内核态的函数调用关系就还挺好把握的。

四、原创说明

本报告参考了一下博客或资料：

1. os-lab4实验报告 | hugo
2. BUAA-OS-lab4 | YannaのBlog