OS —— Lab4实验报告

22373407 王飞阳

一、思考题

Thinking 4.1

- 在 `SAVE_ALL` 中：

  先 `move k0,sp`，先把通用寄存器的sp 复制到 $\$k0$；

  再 `sw k0,TF_REG29(sp)、sw $2,TF_REG2(sp)`：保存现场需要使用 $\$v0$ 作为协寄存器到内存的中转寄存器，写到内存时需要 sp，所以在正式保存协寄存器和通用寄存器前先保存这两个寄存器。

  在系统调用异常处理函数时，会首先调用`SAVE_ALL`宏函数保护现场，将所有通用寄存器，CP0寄存器和当前PC寄存器的值保存到对应栈空间，从而保证在后续步骤不会破坏通用寄存器。

- 可以。

  从用户函数 `syscall_*()` 到内核函数 `sys_*()` 时，$\$a1$—$\$a3$未改变，$\$a0$在`handle_sys()`的时候被修改为内核函数的地址，但在内核函数 `sys_*()` 仅为占位符，不会被用到。同时，在内核态中可能使用这些寄存器进行一些操作计算，此时寄存器原有值被改变，因此再次以这些参数调用其他函数时需要重新以sp为基地址，按相应偏移从用户栈中取用这四个寄存器值。

  总之，一般情况下，还是从栈中取得这些参数更加保险。

- 函数共有6个参数，其中4个参数通过$\$a0$—$\$a3$寄存器传递，剩下两个参数的值保存在栈里，使得用户栈能够完整复制到内核栈空间中，从而保证了sys开头的函数得到了正确的参数。

- 第一，将栈中存储的EPC寄存器值增加4，这是因为系统调用后，将会返回下一条指令，而用户程序会保证系统调用操作不在延迟槽内，所以直接加4得到下一条指令的地址；

  第二，将返回值存入 $\$v0$。

  使得系统调用结束后用户态可以获得正确的系统调用返回值并从发生系统调用的下一条指令开始执行。

Thinking 4.2

在我们生成envid时，后十位为了方便从envs数组中直接取出Env，可能会有所重叠，但是这只能保证envid和e->env_id的低十位是相同的，如果高22位存在不同是无法检测出的，所以需要将envid与e->env_id进行比较。

envid的独一性取决于mkenvid里不断增长的 `i` ，所以如果不判断envid是否相同，会取到错误的或者本该被销毁的进程控制块。

Thinking 4.3

我们可以看到`mkenvid`函数为：

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

`++i` 保证函数返回值一定不会为0；

`envid2env` 函数为：

```
int envid2env(u_int envid, struct Env **penv, int checkperm) {
    struct Env *e;
    if(envid == 0){
        *penv = curenv;
        return 0;
    }
    else e = &envs[ENVX(envid)];
    if (e->env_status == ENV_FREE || e->env_id != envid) {
        return -E_BAD_ENV;
    }
    if(checkperm != 0){
        if(e != curenv && e->env_parent_id != curenv->env_id)
            return -E_BAD_ENV;
    }
    *penv = e;
    return 0;
}
```

`envid2env()` 的 `envid` 为0时返回 `curenv`；

由于 `curenv` 为内核态的变量，用户态不能获取 `curenv` 的 `envid`，所以用 0 代表 `curenv->envid`；

目的是方便用户进程调用 `syscall_*()` 时把当前进程的 `envid` 作为参数传给内核函数，即方便用户态在内核变量不可见的情况下调用内核接口。

Thinking 4.4

c

Thinking 4.5

- 在 0 ~ $USTACKTOP$范围的内存需要使用 `duppage` 进行映射;
- $USTACKTOP \sim UTOP$的 `user exception stack` 是用来进行页写入异常的，不会在处理COW异常时调用 `fork()` ,所以 `user exception stack` 这一页不需要共享;
- $USTACKTOP \sim UTOP$的的`invalid memory` 是为处理页写入异常时做缓冲区用的，所以同理也不需要共享;
- $UTOP$以上页面的内存与页表是所有进程共享的，且用户进程无权限访问，不需要做父子进程间的 `duppage`；其上范围的内存要么属于内核，要么是所有用户进程共享的空间，用户模式下只可以读取。除只读、共享的页面外都需要设置 `PTE_COW` 进行保护。

Thinking 4.6

相关定义如下：

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

- 作用：在用户态下通过访问进程自己的物理内存获取用户页的页目录项页表项的 `perm`，用于 `duppage` 根据不同的 `perm` 类型在父子进程间执行不同的物理页映射；
- 使用：
  - vpd是**页目录首地址**，以vpd为基地址，加上页目录项偏移数即可指向**va对应页目录项**，即 `(*vpd) + (va >> 22)` 或 `vpd[va >> 22]`；
  - vpt是**页表首地址**，以vpt为基地址，加上页表项偏移数即可指向**va对应的页表项**，即 `(*vpt) + (va >> 12)` 或 `vpt[va >> 12]` 即 `vpt[VPN(va)]`；
- 根据vpt和vpd的宏定义，我们可以知道vpt就是指向了用户页表的虚拟地址，vpd指向了用户页目录的虚拟地址。通过偏移量，就可以获得对应的进程自身的页表页目录。
- 由是vpd的定义，`(UVPT + (PDX(UVPT) << PGSHIFT))`，vpd本身处在所映射的页表中的一个页面里，体现了自映射的设计。
- 不能。该区域对用户只读不写，若想要增添页表项，需要陷入内核进行操作。

Thinking 4.7

- 当出现COW异常时，需要使用用户态的系统调用发生中断，即中断重入；
- 由于处理COW异常时调用的 `handle_mod()` 函数把epc改为用户态的异常处理函数 `env_user_tlb_mod_entry`，退出内核中断后跳转到epc所在的用户态的异常处理函数。

  由于用户态把异常处理完毕后仍然在用户态恢复现场，所以此时要把内核保存的现场保存在用户空间的用户异常栈。

Thinking 4.8

- 解放内核，不用内核执行大量的页面拷贝工作；
- 内核态处理失误产生的影响较大，可能会使得操作系统崩溃；
- 用户状态下不能得到一些在内核状态才有的权限，避免改变不必要的内存空间；
- 同时微内核的模式下，用户态进行新页面的分配映射也更加灵活方便。

Thinking 4.9

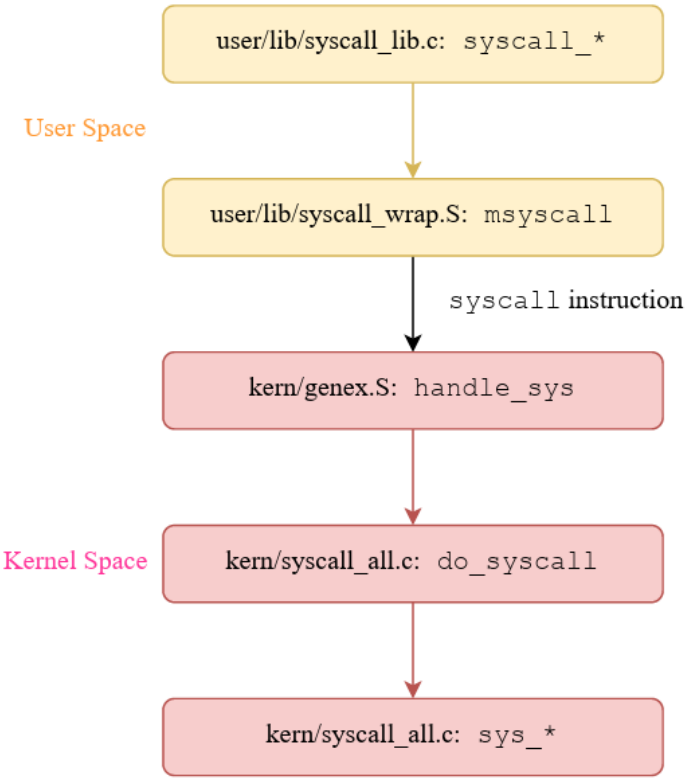
- `syscall_exofork()` 返回后父子进程各自执行自己的进程，子进程需要修改 `entry.S` 中定义的env指针，涉及到对COW页面的修改，会触发COW写入异常，COW中断的处理机制依赖于 `syscall_set_tlb_mod_entry`，所以将 `syscall_set_tlb_mod_entry` 的调用放置在 `syscall_exofork` 之前；
- 父进程在调用写时复制保护机制可能会引发缺页异常，而异常处理未设置好，则不能正常处理。

难点分析

一、系统调用

这部分是整个lab4实验的基础，在刚开始阅读内核实验指导书时往往会分不清什么时候在内核进行，什么时候在用户态进行，容易扰乱思路，现总结如下：

系统调用示意图：



- `sys*`均是内核函数。
- `msyscall`是用户函数。
- `syscall*`是用户函数。
- `duppage`是用户函数。
- `fork`是用户函数。

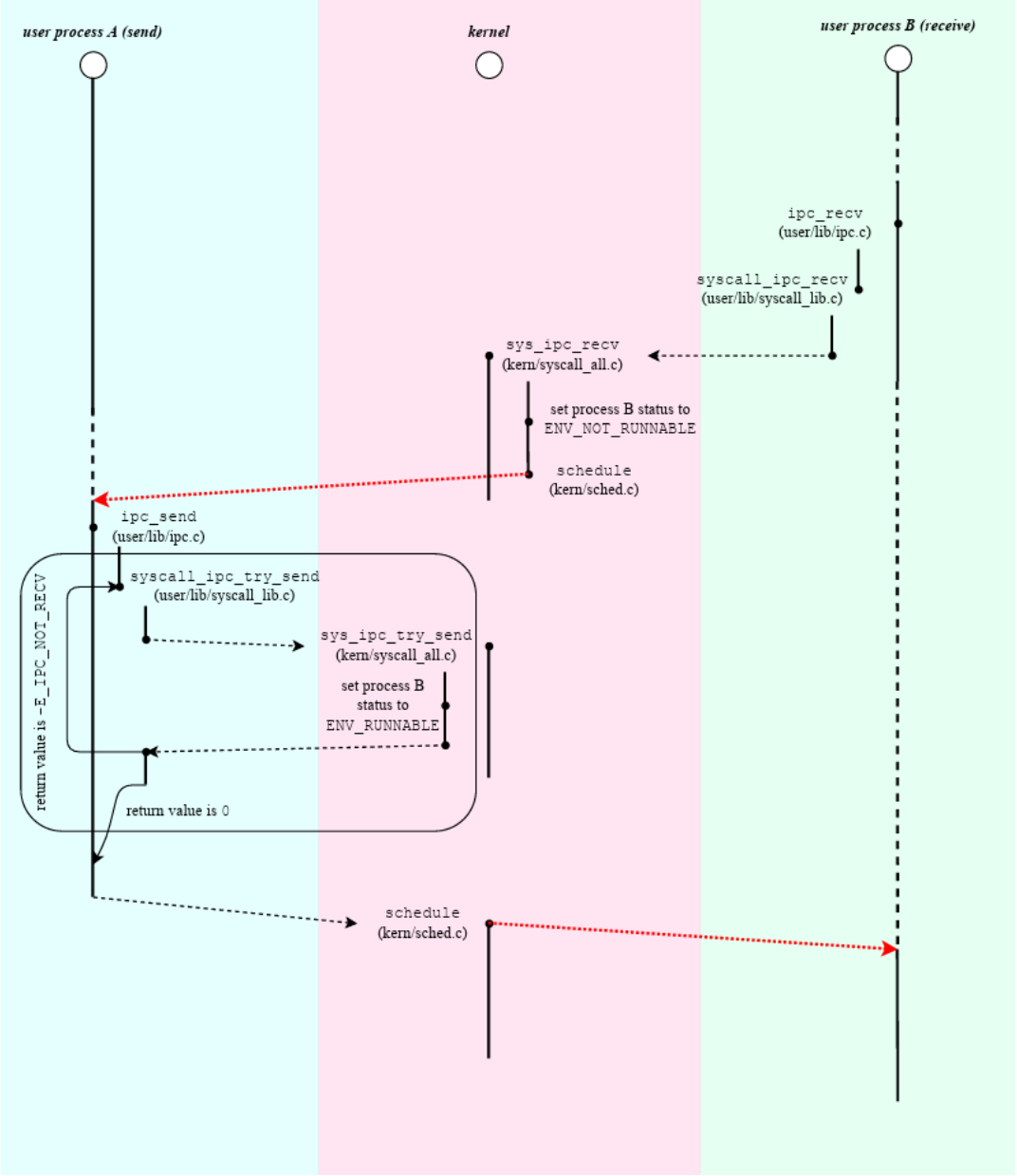
二、进程间通信机制(IPC)

IPC是微内核最重要的机制之一，其主要思想为借助所有的进程都共享同一个内核空间（主要为kseg0），来实现不同进程之间的通信。

对应的代码结构主要实现在`env`结构体中：

```
    // Lab 4 IPC
    u_int env_ipc_value;   // the value sent to us
    u_int env_ipc_from;    // envid of the sender
    u_int env_ipc_recving; // whether this env is blocked receiving
    u_int env_ipc_dstva;   // va at which the received page should be mapped
    u_int env_ipc_perm;    // perm in which the received page should be mapped
```

IPC流程图如下：



三、fork

在fork的实现过程中，个人认为比较困难的是`cow_entry`函数以及`fork`函数。

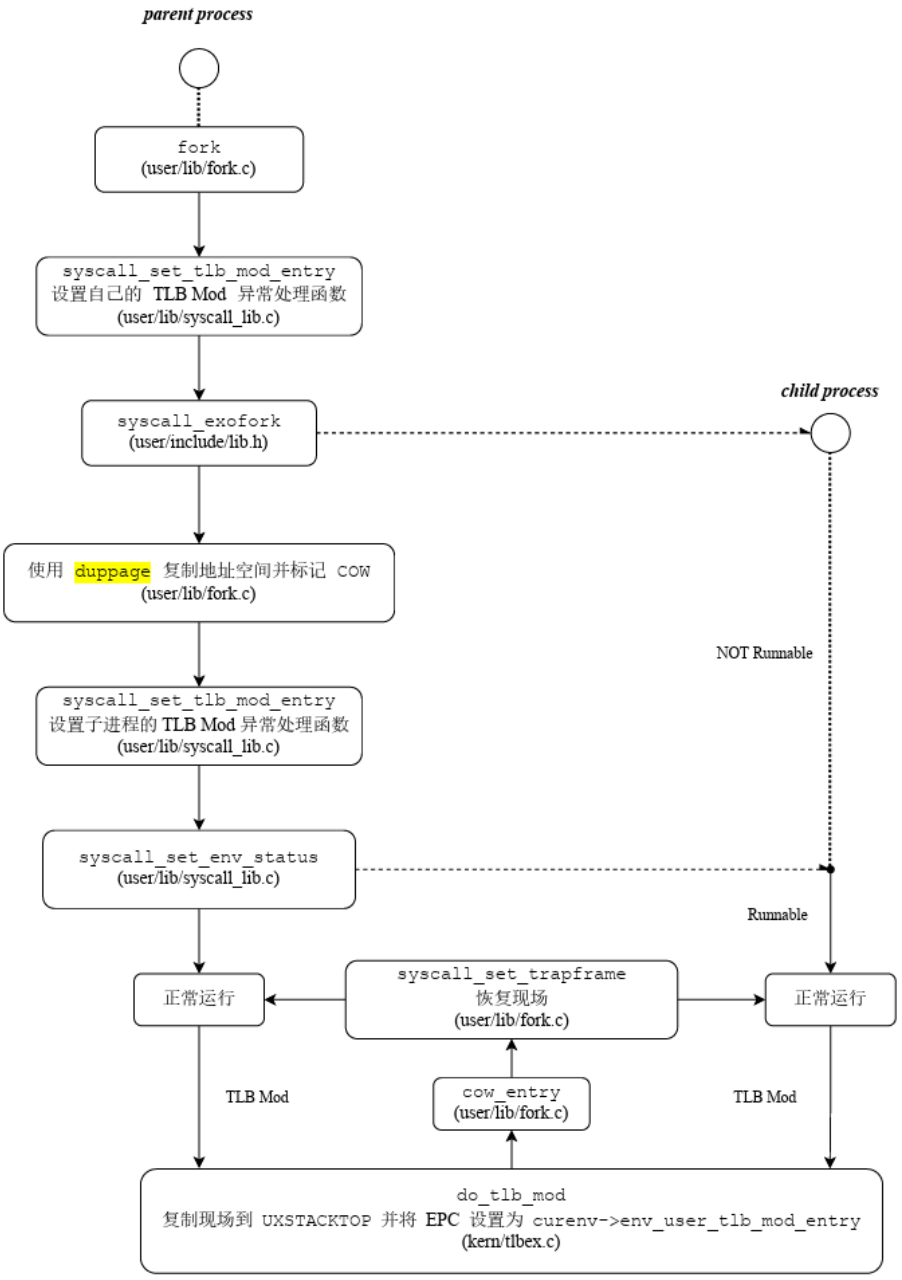
在`cow_entry`函数中，需要注意很多细节，尤其是在复制时需要将虚拟地址与页面大小对齐

```
memcpy((void *)tmp, (void *)ROUNDDOWN(va, PAGE_SIZE), PAGE_SIZE);
```

在`fork`函数中，个人觉得难点在于对vpd，vpt的理解和使用，同时需注意判断页目录和页表项获得的地址是否有效。

```
if ((vpd[i >> 10] & PTE_V) && (vpt[i] & PTE_V)) {
    duppage(child, i);
}
```

而对于整个fork实现过程的理解，可借用指导书的示意图：



实验体会

lab4主要实现用户态的一些系统调用、进程之间的通信，以及如何创建子进程（fork函数）。相较于前面的实验，无论是题量还是难度都有所提升。对于系统函数和进程通信相对要容易一点，而fork函数主要难点集中在其中内存映射等方面，所以需要好好理解页写入异常的处理，以及写时复制的机制等内存机制。因为涉及到用户进程，debug的难度要高一点。在刚完成lab4试验后，本人对整个实验都还没有较清晰的了解，直到在根据思考题完成实验之后，才对整个过程有了比较清楚的认识。