Lab4 实验报告

思考题部分

Thinking 4.1

- 在陷入内核态前，系统会先调用SAVE_ALL宏将当前用户态进程的所有寄存器内容保存到指定的位置，从而保证用户态下寄存器内的值不会被内核态的指令破坏。
- 可以，在执行msyscall时，$a0-$a3寄存器用于存储对应的前四个参数。而在do_syscall的过程中并没有修改这几个寄存器中的值，因此内核态应该能够直接从$a0-$a3寄存器中读取到msyscall留下的信息。
- 在执行sys开头的系列函数之前，do_syscall函数从$a0中取出了目标sys函数的代号，然后分别从$a1-$a3寄存器和用户栈中取出了sys函数所需要的参数。而这些参数都是执行msyscall时传入的，故sys和msyscall使用的是同样的参数。
- 在handle_sys函数中，将Trapframe中的epc值取出并加了4，这种修改对应模拟执行完毕该条需要系统调用的指令后跳转到了下一条指令。

Thinking 4.2

确实通过envid获取的进程控制块的正确性。否则如果转换的过程中出现问题，可能会修改到不该修改的进程控制块。

Thinking 4.3

因为在envid2env()函数中，传入的envid为0的话，会直接返回当前进程的进程控制块。如果在mkenvid()的过程中创造出了envid为0的进程控制块，在完成信息传输的过程中，发送方如果想要发送给envid为0的进程，则会错误地找到自己并尝试将信息传输给自己，从而无法正确地完成进程间的信息传递。

Thinking 4.4

C

Thinking 4.5

应该将`USTACKTOP`下的所有用户空间页进行映射。因为向上是每个进程独属的异常处理栈，该部分是不能映射给子进程的。

Thinking 4.6

- vpt的作用是获取当前进程页表的起始地址，vpd的作用是获取当前进程页目录的地址。在具体使用时，可以通过形如`vpt[vpn]`的形式获取对应虚拟页号的页表项，以及`vpd[vpn>>10]`的形式获取对应虚拟页号的页目录项。
- `vpt ((volatile Pte *)UVPT)`

上述部分是vpt宏定义的实现过程。可以看出，vpt实际就是将内存空间中用户页表的起始地址转化成了一个页表项的地址。这个页表项正是用户页表的第一个页表项，所以可以通过这种方式来存取自身页表。

- `#define vpd ((volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT)))`

从vpd的定义即可看出，vpd的定义等价于$PD_{base}=PT_{base}|(PT_{base}>>10)$，这种形式正是在lab2中呈现出的页表自映射的形式。

- 不能，从`UVPT`向上的空间对于用户态来说应该是只读的，虽然能够通过上述宏定义获取到地址，但是只读的限制和从原理上来看用户态都不应该有修改页表的能力。

Thinking 4.7

- 用户写入了cow标记但没有PTE_D标记的页表项对应的页时，会触发页写入异常。经过异常分发到达handle_tlb_mod，最后到达用户态的异常处理程序中。如果用户态对应的异常处理函数还会发生上述异常的话，就会产生异常重入。
- 如果不复制的话，用户态无法接触到KSTACKTOP处保存的上下文Trapframe。同时由于内核态时将cp0_epc设置为了cow_entry所在的地址，如果不保留异常的现场，则无法返回到触发页写入异常的指令，程序无法正常执行完毕。

Thinking 4.8

更加符合微内核的实际理念，将处理过程作为服务放在用户态处理，能够较好地保证内核的稳定性。

Thinking 4.9

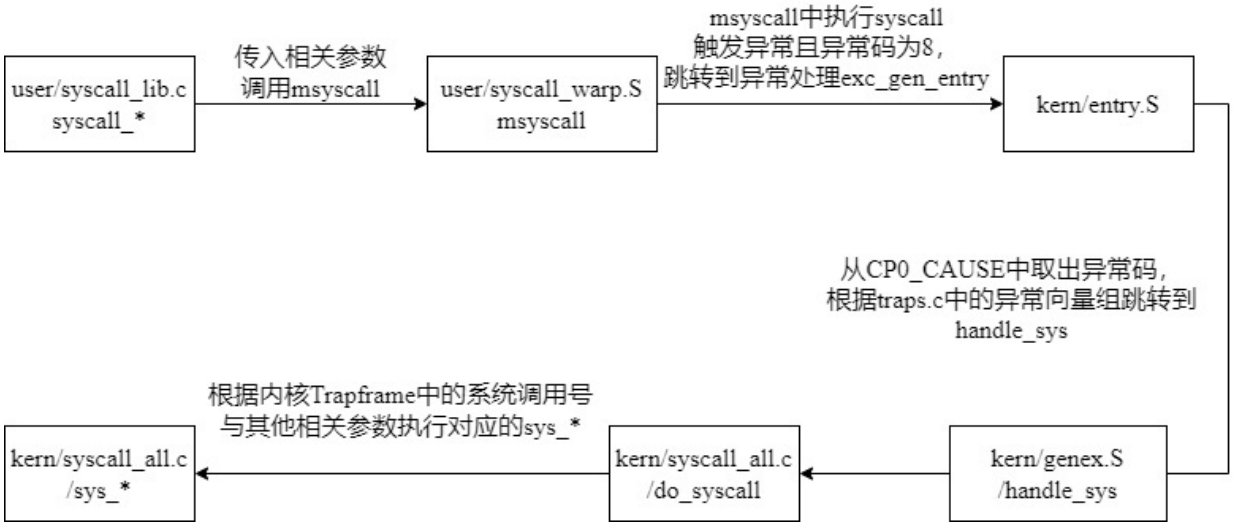
- 因为在sys_exofork的过程中就有可能出现tlb_mod的情况，所以要在syscall_exofork之前完成tlb_mod_entry的设置。
- 同上，如果在写时复制保护机制完成之后也会产生相同的影响。

实验难点分析&理解

系统调用的流程

实现一个系统调用需要如下完成如下的流程:

1. 在用户的进程中，部分函数需要利用syscall完成相应的功能，调用syscall_*
2. syscall_*将自己的参数传入msyscall，并附带系统调用号
3. msyscall执行syscall,触发8号异常，系统进入内核态
4. 在异常分发过程中，根据异常号结合异常向量组，跳转到handle_sys
5. handle_sys执行对应的解决程序do_syscall.
6. do_syscall根据系统调用号调用对应的sys_*函数完成对应操作并返回



因此。如果需要新增系统调用，则需要修改以下部分：

1. 在user/syscall_lib.c 中新增对应的syscall_*函数，将参数传入msyscall中
2. 在include/syscall.h中添加对应的enum
3. 在syscall_all.c 的syscall_table指定对应的sys_*函数
4. 在syscall_all.c 中实现sys_*函数。

IPC执行的流程

1. 接收方执行syscall_ipc_recv，最终跳转到内核中执行sys_ipc_recv。在sys_ipc_recv中需要完成：

- 检测接收地址合法性
- 设置env_ipc_recving为1，表明自己进入等待接受的状态
- 设置env_ipc_dstva为目标地址
- 将进程控制块的状态调整至ENV_NOT_RUNNABLE，并移出调度队列
- 使用调度函数切换至其他进程，放弃CPU

2. 发送方执行syscall_ipc_try_send，最终跳转到内核中执行sys_ipc_try_send。在sys_ipc_try_send中需要完成：

- 检测发送地址的合法性（如果需要建立映射的话，即srcva不为0）
- 根据env_id获取接收方的进程控制块e，检测e的接受状态(env_ipc_recving)是否为1。
- 设置目标进程控制块的相关信息（value,from,perm）等，并将env_ipc_recving置为0，表示传输结束。
- 设置目标进程控制块状态位RUNNABLE，将其重新放回调度队列中。
- 如果需要建立映射（通过物理页传递数据），则利用page_insert建立映射。

写时复制机制

目的：避免父进程在创建子进程时对共享页面的大量复制造成的内存空间浪费。

具体实现步骤——

1. 父/子进程在尝试写入没有PTE_D但有PTE_COW权限位的页表项对应的物理页时触发页写入异常

2. 异常分发根据异常号定位到handle_mod，handle_mod保存当前的上下文（寄存器内容），传入到tlbex.c中的do_tlb_mod中
3. do_tlb_mod将当前的tf保存副本到栈中后修改cp0_epc的值，保证在用户态跳转至cow_entry处。
4. 在cow_entry中完成写时复制的具体操作

```
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
    u_int va = tf->cp0_badvaddr;                                    //获取触发页写入异常对应的虚拟地址
    u_int perm;

    perm = (vpt[VPN(va)])&0xfff;                                    //获取对应页表项权限位
    if((perm&PTE_COW)==0){                                          //如果没有COW位，报错
        user_panic("perm does not include PTE_COW");
    }
    perm &= ~PTE_COW;
    perm |= PTE_D;
    syscall_mem_alloc(0,UCOW,PTE_D);                                //在UCOW处申请新的一页，并赋予可写
    memcpy((void *)UCOW,(const void *)ROUNDDOWN(va,BY2PG),BY2PG);   //将原来映射的物理页内容复制到新页
    syscall_mem_map(0,UCOW,0,va,perm);                              //更改映射为新的物理页，达到可修改的
目的
    syscall_mem_unmap(0,UCOW);                                      //解除UCOW对该新页的映射
    int r = syscall_set_trapframe(0, tf);                           //恢复保存好的现场，保证能正常回到用
户态
    user_panic("syscall_set_trapframe returned %d", r);
}
```

Fork的整体流程

1. 设置自身的页写入异常函数为已经完成的cow_entry，以满足写时复制机制的要求。
2. 执行syscall_exofork()系统调用，完成子进程的申请与初始化。

```
int sys_exofork(void) {
    struct Env *e;
    try(env_alloc(&e, curenv->env_id));                          //申请一个新的进程控制
块，并设置其父子关系
    e->env_tf=*((struct Trapframe *)KSTACKTOP - 1);              //完成寄存器内容的复制
    e->env_tf.regs[2]=0;                                         //设置子进程该函数的返回
值为0，以区分父子进程
    e->env_status=ENV_NOT_RUNNABLE;                              //子进程还未完全准备完
毕，仍然不能运行
    e->env_pri=curenv->env_pri;                                  //完成优先级的复制
    return e->env_id;                                            //对于父进程来说，
exofork应返回子进程的envid
}
```

对于子进程来说，fork函数基本已经结束，对父进程来说，还需要完成以下的操作：

3. 利用duppage,完成父进程到子进程应该共享的界面的映射(below USTACKTOP)。

```
static void duppage(u_int envid, u_int vpn) {
    int r=0;
    u_int addr;
    u_int perm;
    addr=vpn*BY2PG;
    perm=vpt[vpn]&0xfff;                                         //利用vpt宏定义获得虚拟页号
vpn对应的页表项并获取其perm
    if(((perm&PTE_D)>0)&&((perm&PTE_LIBRARY)==0)){               //找出父进程中可写且不是共享
页面(library)的页表项对应
        perm = (perm & ~PTE_D) | PTE_COW;                          的页，为其标记上COW(写时复
制的标志)
        r=1;
    }
    syscall_mem_map(0, addr, envid, addr, perm);                 //将子进程对应的虚拟地址映射
到父进程的页中
    if(r){
        syscall_mem_map(0, addr, 0, addr, perm);                 //如果perm发生变化(上了
cow),也要修改父进程中的页表项
    }
}
```

4. 为子进程设置对应的页写入异常函数(cow_entry)。
5. 设置子进程的状态ENV_RUNNABLE，并在sys_set_env_status的具体过程中将其加入调度队列中，使子进程获得运行机会。

实验体会

在完成本次实验的过程中，由于用户态，系统调用功能的进一步拓展，难点也相应地集中出现在了用户态和内核态的沟通和转换过程。无论是完成IPC，还是完成fork，都需要在用户态下触发某种异常，再经过msyscall，异常分发之后完成相应的处理。也因此，在完成本单元的任务时最重要的就是处理好从用户态到内核态的整套流程。在课下利用vscode完成相应代码的编写时，借助于简单的搜索，并列文档查看等工具，处理整个流程还相对容易一些。在lab4-1的上机过程中，在CLI，通过vim操作则使整个过程变得相当困难。

在用户态到内核的过程中只要两边互相对应的函数有不匹配的地方，就会造成错误，这就对书写代码的准确性做出了严格的要求。