实验报告

思考题

Thinking 4.1

要想回答这个问题，我们先重新梳理一下系统调用的流程。

用户主动调用 用户sys_*

在user/lib/syscall_lib.c中：

```
void syscall_putchar(int ch) {
    msyscall(SYS_putchar, ch);
}

int syscall_print_cons(const void *str, u_int num) {
    return msyscall(SYS_print_cons, str, num);
}
...
```

在调用函数时，参数已经被压入了栈帧中对应的位置。

执行msyscall

在user/lib/syscall_wrap.S中：

```
LEAF(msyscall)
    syscall
    jr  ra
END(msyscall)
```

PC跳转进入异常入口

CPU执行syscall，直接跳转pc到异常入口

并执行entry.S中的:

```
exc_gen_entry:
    SAVE_ALL
    mfc0    t0, CP0_STATUS
    and     t0, t0, ~(STATUS_UM | STATUS_EXL | STATUS_IE)
    mtc0    t0, CP0_STATUS
    mfc0    t0, CP0_CAUSE
    andi    t0, 0x7c
```

```
    lw      t0, exception_handlers(t0)
    jr      t0
```

这个过程就回答了前两个问题。 SAVE_ALL保护了所有通用寄存器。

并且$a0-$a3没有被重新赋值。因此陷入内核调用后仍可以直接从当时的$a0-$a3参数寄存器中得到用户调用msyscall留下的信息。

得到异常类型

由traps.c中的exception_handlers:

```
void (*exception_handlers[32])(void) = {
    [0 ... 31] = handle_reserved,
    [0] = handle_int,
    [2 ... 3] = handle_tlb,
#if !defined(LAB) || LAB >= 4
    [1] = handle_mod,
    [8] = handle_sys,
#endif
};
```

调用异常处理函数

```
.macro BUILD_HANDLER exception handler
NESTED(handle_\exception, TF_SIZE + 8, zero)
    move    a0, sp
    addiu   sp, sp, -8
    jal     \handler
    addiu   sp, sp, 8
    j       ret_from_exception
END(handle_\exception)
.endm

BUILD_HANDLER sys do_syscall
```

传参与调用内核sys_*

```
void do_syscall(struct Trapframe *tf) {
    int (*func)(u_int, u_int, u_int, u_int, u_int);
    int sysno = tf->regs[4];
    if (sysno < 0 || sysno >= MAX_SYSNO) {
        tf->regs[2] = -E_NO_SYS;
        return;
    }
    tf->cp0_epc += 4;
```

```
    func = syscall_table[sysno];
    u_int arg1 = tf->regs[5];
    u_int arg2 = tf->regs[6];
    u_int arg3 = tf->regs[7];
    u_int arg4, arg5;
    arg4 = *((u_int *)tf->regs[29] + 4);
    arg5 = *((u_int *)tf->regs[29] + 5);
    tf->regs[2] = func(arg1, arg2, arg3, arg4, arg5);
}
```

这里可以回答后两个问题：

用户sys中的6个参数，第一个用来对应内核sys，其他参数按照原来的顺序依次用于传参，因此数值是完全对应的。

对于syscall后的改动：

1. 系统调用的（正常/异常）返回值会赋值给$2即$v0
2. tf->cp0_epc += 4;

Thinking 4.2

结合以下两段代码：

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

```
#define LOG2NENV 10
#define NENV (1 << LOG2NENV)
#define ENVX(envid) ((envid) & (NENV - 1))
```

可以知道ENVX只取了envid的后10位，也就是取了e-envs的值

这个值反映的就是e在envs中的相对位置，所以可以用来找到这个进程。

但是它并不构成双射。

例如0xF01,0x801,0x401等等envid都可以对应到envs[1],但是这些envid大多是不合法的。因此必须再进行反向确认。

Thinking 4.3

这个问题不难解释。

```
int envid2env(u_int envid, struct Env **penv, int checkperm) {
    struct Env *e;
    if (envid == 0) {
        *penv = curenv;
        return 0;
    }
    e = &envs[ENVX(envid)];
    ...
}
```

我们知道envid取0时，其含义即为curenv。

因此mkenvid避免了返回0，没有进程的envid为0，从而防止在envid2env(0, &env, 0)时，只能取到curenv而不能取envid确实为0的进程的情况。

而envid2env(envid, &env, 0)在IPC的sys_ipc_try_send中被调用，若存在envid为0的进程，将影响进程间通信的正常实现。

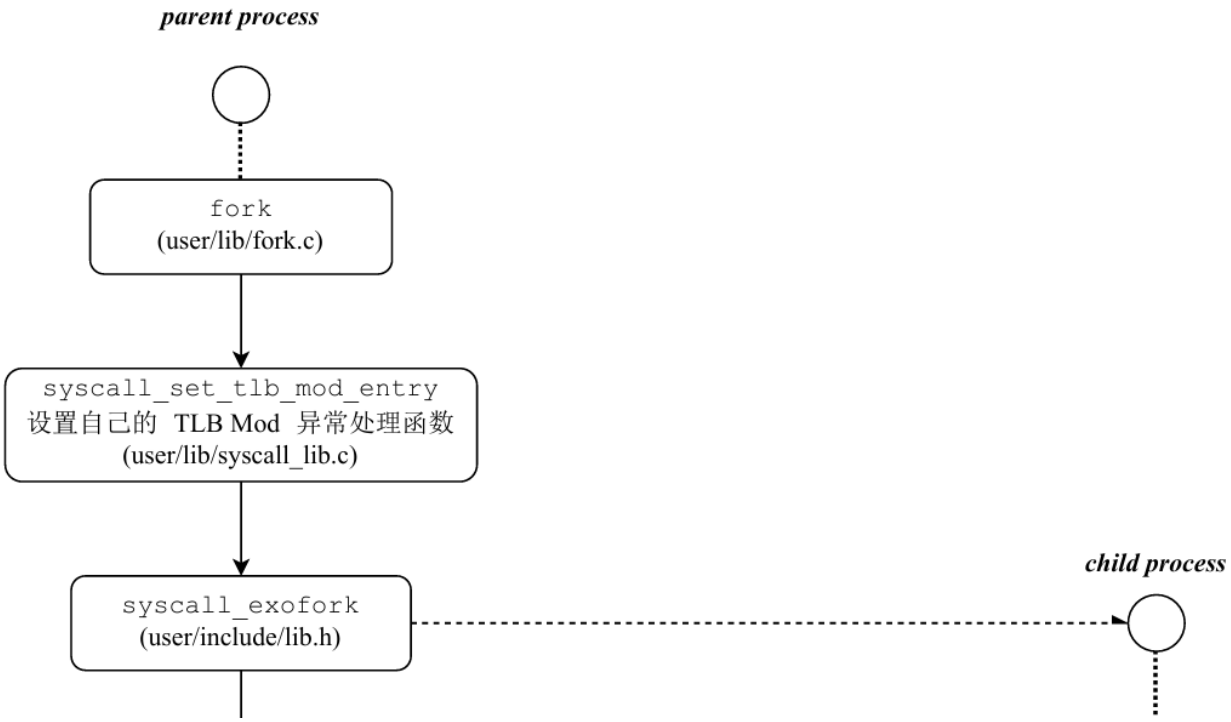
Thinking 4.4

答案为C。

参见任务指导书：

> • fork 之前只有父进程存在。
>
> • fork 之后，父子进程同时开始执行fork之后的代码段

以及此图：



可以确定fork代码本身是父进程执行的。

fork在父、子进程的返回值分别是子进程pid与0。

Thinking 4.5

一般认为，用户空间从0到USTACKTOP需要被映射。

在env_init中:

```
...
base_pgdir = (Pde *)page2kva(p);
map_segment(base_pgdir, 0, PADDR(pages), UPAGES, ROUND(npage * sizeof(struct
Page), PAGE_SIZE), PTE_G);
map_segment(base_pgdir, 0, PADDR(envs), UENVS, ROUND(NENV * sizeof(struct Env),
PAGE_SIZE), PTE_G);
...
```

进程的UPAGES与UENVS已经映射到pages与envs了，因此不需要再在duppage中进行继承。

而一些进程临时占用的区域往往在fork时是无效的，因此也不会被复制。

Thinking 4.6

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

vpt就是用户页表起始位置UVPT，指针为Pte*类型，即`vpt+1`即指向下一页表项。

vpd则是利用了自映射，PDX号即为页目录在二级页表中的位置。

> 也可写作 #define vpd ((const volatile Pde *)UVPT + (PDX(UVPT)))

已知： vpn: va的前20位 PTX：va的中间10位 PDX：va的前10位

则有： va对应的页目录项： vpd[PDX(va)] va对应的页表项： vpt[vpn]

这便是用户态访问页表项的基本方法。

进程无权修改页表项。为什么？ 在env_init中我们只额外赋予了PTE_G， 即`map_segment(base_pgdir, 0, PADDR(pages), UPAGES, ROUND(npage * sizeof(struct Page), PAGE_SIZE), PTE_G);` 详见map_segment中的page_insert中的 `*pte = page2pa(pp) | perm | PTE_C_CACHEABLE | PTE_V;`。

所以其属性只有PTE_G | PTE_C_CACHEABLE | PTE_V。

Thinking 4.7

```
void do_tlb_mod(struct Trapframe *tf) {
    struct Trapframe tmp_tf = *tf;
    if (tf->regs[29] < USTACKTOP || tf->regs[29] >= UXSTACKTOP) {
```

```
            tf->regs[29] = UXSTACKTOP;
        }
        // 切换为异常栈
        tf->regs[29] -= sizeof(struct Trapframe);
        *(struct Trapframe *)tf->regs[29] = tmp_tf;
        // 将异常现场tmp_tf压入栈
        Pte *pte;
        page_lookup(cur_pgdir, tf->cp0_badvaddr, &pte);
        if (curenv->env_user_tlb_mod_entry) {
            tf->regs[4] = tf->regs[29];
            tf->regs[29] -= sizeof(tf->regs[4]);
            // 设置栈指针为参数$a0
            tf->cp0_epc = curenv->env_user_tlb_mod_entry;
            // 进入异常处理函数
            // 返回epc是异常处理entry，参数tf和本函数的tf一致
        } else {
            panic("TLB Mod but no user handler registered");
        }
    }
```

我们可以看到异常栈的使用与tf的嵌套保存实现了类似于异常重入的处理。 当COW处理函数自身某个环节触发异常时就会触发异常重入。

这时候，新的异常环境被压入异常栈，并被处理，返回到原异常环境，重新处理原异常。

内核需要将异常的现场Trapframe复制到用户空间，由用户态实现现场的恢复。

Thinking 4.8

在用户态处理页写入异常，相比于在内核态：

1. 主观能动性更强，用户可以相对自由地控制异常的产生与处理
2. 减少状态的切换，提高运转效率
3. 与内核独立开来，保证内核的安全

Thinking 4.9

如果`syscall_set_tlb_mod_entry`的调用放置在`syscall_exofork`之后，甚至放置在写时复制保护机制完成之后， syscall_exofork中可能产生的tlb_mod异常将没有处理函数，即`panic("TLB Mod but no user handler registered");`

难点分析

我认为lab4的难度主要在于理解FORK和TrapFrame。

关于系统调用，Thinking 4.1的流程解释已经相对清晰了。

关于IPC，核心内容就是进程状态的变化，进程的切换，data的传递或页的共享。

fork

借助于系统调用，用户态下的函数可以实现许多安全的内核操作。这避免了内核态与用户态的频繁切换，不需多余的SAVE与RESTORE。

fork函数就是用户态下的，能够由用户主动产生新进程的函数。

此外，fork函数通过复用异常机制，实现了COW操作。

```
int fork(void) {
    u_int child;
    u_int i;
    if (env->env_user_tlb_mod_entry != (u_int)cow_entry) {
        try(syscall_set_tlb_mod_entry(0, cow_entry));
    } // 填入COW异常处理函数
    child = syscall_exofork();
    // env_alloc + 赋值tf,pri,status
    // 最后额外更改 $v0
    if (child == 0) {
        env = envs + ENVX(syscall_getenvid());
        // envid中ENVX的含义
        return 0; // child的fork结束
    } for (i = 0; i < PDX(UXSTACKTOP); i++) {
        if (vpd[i] & PTE_V) {
            for (u_int j = 0; j < PAGE_SIZE / sizeof(Pte); j++) {
                u_long va = (i * (PAGE_SIZE / sizeof(Pte)) + j) << PGSHIFT;
                if (va >= USTACKTOP) {
                    break;
                } if (vpt[VPN(va)] & PTE_V) {
                    duppage(child, VPN(va));
                    // 全部标记 COW
                }
            }
        }
    } syscall_set_tlb_mod_entry(child, cow_entry);
    syscall_set_env_status(child, ENV_RUNNABLE);
    // 先设置好COW异常处理函数，再RUNNABLE
    return child;
}
```

这便是fork的全流程。整体看起来还比较清晰？

TrapFrame

这是一块很大的内容，存储着进入内核态前的信息。

陷阱帧也是栈结构，支持重入。

tf->regs是我们访问基础寄存器的途径， 常用的有： tf->regs[2]代表返回值 tf->regs[4..7] 代表传入参数 tf->reg[29]代表sp

注意：可以通过*(struct Trapframe *)tf->regs[29]来得到上一次保存环境时的栈帧。

此外，tf->cp0_epc 代表eret要返回到的PC。将其设置为某个函数的指针，便可以实现函数的跳转（并非调用）
tf->cp0_badvaddr代表出现异常的虚拟地址，往往在处理与内存相关的异常时有帮助。

实验体会

理解Trapframe的过程很痛苦，哪怕是现在我也仍然有很多地方不明所以。

在写思考题的过程中，我总是想要诉之以理，结果却连自己都无法信服。

用户态陷入异常后，进程是否切换？此时的tf存着什么？在异常处理过程中处于内核态吗？那又如何理解"调用"用户态的函数？...

诸多谜题解决起来确实困难，但回到实验指导书重新阅读可能会有曾未发现的收获。