1. 思考题

Thinking 4.1

> **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？
> - 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？
> - 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？
> - 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？
>
> ■

- 使用 `SAVE_ALL`，把通用寄存器 `sp` 复制到 `$k0`；保存现场需要使用 `$v0` 作为协寄存器到内存的中转寄存器，写到内存时需要 `sp`，所以正式保存协寄存器和通用寄存器先保存这两个寄存器。
- 从用户函数 `syscall_*()` 到内核函数 `sys_*()` 时，`$a1-$a3` 未改变，`$a0` 在 `handle_sys()` 的时候被修改为内核函数的地址，但在内核函数 `sys_*()` 仅为占位符，不会被用到。同时，在内核态中可能使用这些寄存器进行一些操作计算，此时寄存器原有值被改变，因此再次以这些参数调用其他函数时需要重新以 `sp` 为基地址，按相应偏移从用户栈中取用这四个寄存器值。
- 用户调用时的参数：
  1. 用户进程的寄存器现场的 `$a1-$a3`
  2. 用户栈(栈指针为**用户现场**的 `sp`)的参数 `$a4、$a5`
- 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`。

Thinking 4.2

> **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？ ■

- 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，`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

> **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。 ■

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

- `++i` 保证一定不会为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

> **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 ■

- 选C

Thinking 4.5

> **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。 ■

- 在用户进程的内存空间中，`0` 到 `USTACKTOP` 范围内的内存需要使用 `duppage` 进行映射，以确保父子进程间的写时复制（COW）机制正常工作。而 `USTACKTOP` 到 `UTOP` 之间的 `user exception stack` 是用于处理页写入异常的，不会在 COW 异常时调用 `fork()`，因此这一页不需要共享。同样，`USTACKTOP` 到 `UTOP` 之间的 `invalid memory` 是作为页写入异常的缓冲区使用的，也不需要共享。`UTOP` 以上的内存（如内核空间和共享区域）是所有进程共享的，且用户进程无权限修改，因此无需进行 `duppage` 映射。除只读和共享的页面外，其余可写页面均需设置 `PTE_COW` 标志位，以实现写时复制保护。

Thinking 4.6

> **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？
> - 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？
> - 它们是如何体现自映射设计的？
> - 进程能够通过这种方式来修改自己的页表项吗？
>
> ■

- vpd 是**页目录首地址**，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)]`；
- vpd的地址在UVPT和UVPT + PDMAP之间，说明将页目录映射到了某一页表位置(即实现了自映射);
- 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

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

- 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

> **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？ ■

- 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

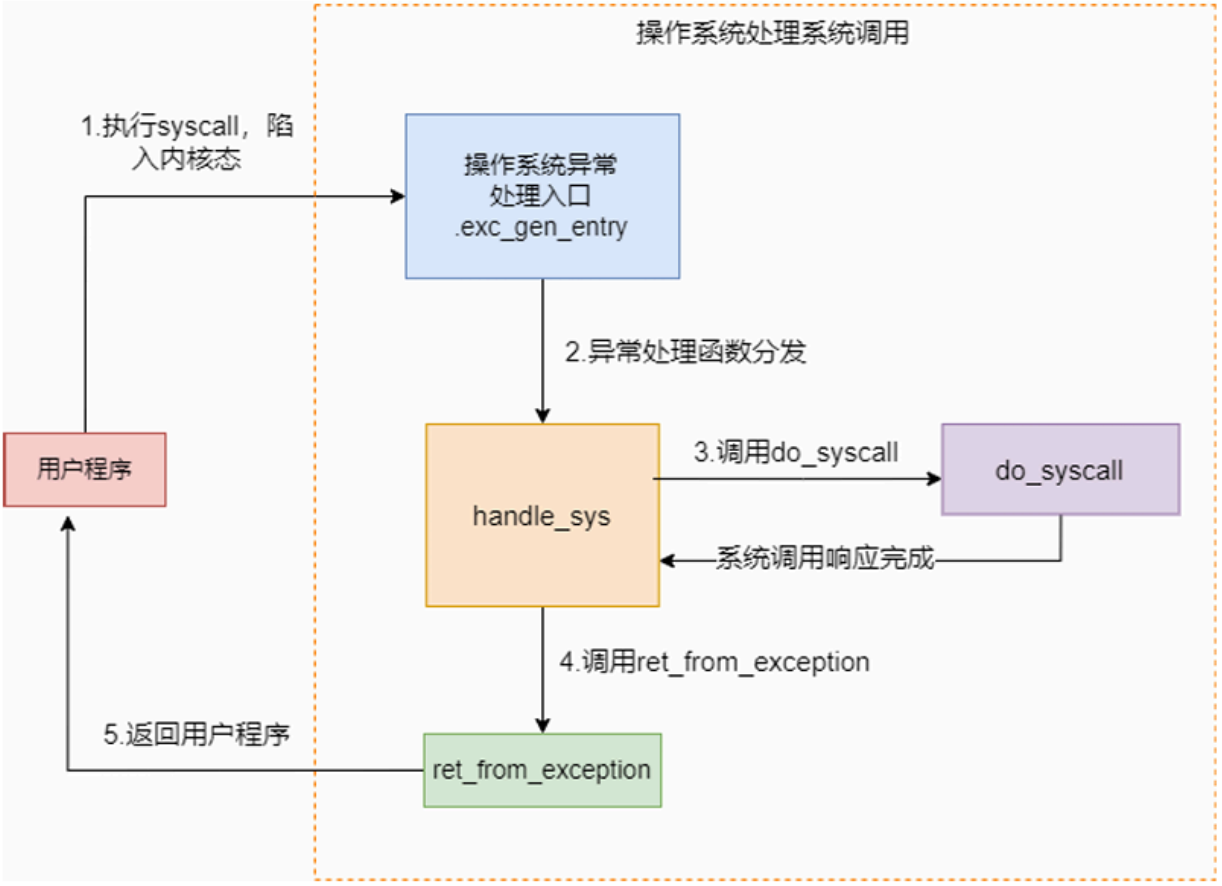
> **Thinking 4.9** 请思考并回答以下几个问题：
>
> - 为什么需要将 `syscall_set_tlb_mod_entry` 的调用放置在 `syscall_exofork` 之前？
> - 如果放置在写时复制保护机制完成之后会有怎样的效果？
>
> ■

- `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。

2. 难点分析

2.1 系统调用

2.1.1 系统调用流程图



2.1.2 从代码角度看流程

1. 首先在`user/lib/syscall_lib.c`中执行`syscall_putchar`函数（以此为例）该函数调用`msyscall`，`msyscall`定义在`user/lib/syscall_wrap.S`里，它在调用`syscall`后会进入异常处理函数`kern/entry.S`，也就是在`lab3`中提到的`exc_gen_entry`，在`SAVE_ALL`后回去查异常中断表`exception_handlers(t0)`，后者定义在`trap.c`中，[8]对应`handle_sys`，而后者又是定义在`genex.S`中的`BUILD_HANDLER sys do_syscall`而`do_syscall`又是定义在`kern/syscall_all.c`中。最后返回用户程序继续运行。

2.1.3 流程图的代码细节

- `user/lib/syscall_lib.c`

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

- `user/lib/syscall_wrap.S`

```
#include <asm/asm.h>

LEAF(msyscall)
    // Just use 'syscall' instruction and return.
    /* Exercise 4.1: Your code here. */
    syscall // 让CPU陷入内核态，触发8号异常/sys异常
    jr  ra

END(msyscall)
```

- `kern/entry.S`

```
#include <asm/asm.h>
#include <stackframe.h>

.section .text.tlb_miss_entry
tlb_miss_entry:
    j       exc_gen_entry

.section .text.exc_gen_entry
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) // t0 为 8
    jr  t0
```

- `kern/trap.c`

```
#include <env.h>
#include <pmap.h>
#include <printk.h>
#include <trap.h>

extern void handle_int(void);
extern void handle_tlb(void);
extern void handle_sys(void);
extern void handle_mod(void);
extern void handle_reserved(void);

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
```

```
};

void do_reserved(struct Trapframe *tf) {
    print_tf(tf);
    panic("Unknown ExcCode %2d", (tf->cp0_cause >> 2) & 0x1f);
}
```

- `kern/genex.S`

```
#include <asm/asm.h>
#include <stackframe.h>

.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 // 从异常回退
END(handle_\exception)
.endm

.text

FEXPORT(ret_from_exception)
    RESTORE_ALL
    eret

NESTED(handle_int, TF_SIZE, zero)
    mfc0    t0, CP0_CAUSE
    mfc0    t2, CP0_STATUS
    and     t0, t2
    andi    t1, t0, STATUS_IM7
    bnez    t1, timer_irq
timer_irq:
    li      a0, 0
    j       schedule
END(handle_int)

BUILD_HANDLER tlb do_tlb_refill

#if !defined(LAB) || LAB >= 4
BUILD_HANDLER mod do_tlb_mod
BUILD_HANDLER sys do_syscall // 定义在这里
#endif

BUILD_HANDLER reserved do_reserved
```

- `kern/syscall_all.c`

```
void do_syscall(struct Trapframe *tf) {
    int (*func)(u_int, u_int, u_int, u_int, u_int);  // 声明一个函数指针
```

```
    int sysno = tf->regs[4]; // 从寄存器4($a0)获取系统调用号
    if (sysno < 0 || sysno >= MAX_SYSNO) {
        tf->regs[2] = -E_NO_SYS;
        return;
    }

    /* Step 1: Add the EPC in 'tf' by a word (size of an instruction). */
    /* Exercise 4.2: Your code here. (1/4) */
    tf->cp0_epc += 4;  // 将异常程序计数器(EPC)增加4（一条指令的长度）
    /* Step 2: Use 'sysno' to get 'func' from 'syscall_table'. */
    /* Exercise 4.2: Your code here. (2/4) */
    func = syscall_table[sysno];  // 从系统调用表中获取对应的函数
    /* Step 3: First 3 args are stored in $a1, $a2, $a3. */
    u_int arg1 = tf->regs[5]; // $a1
    u_int arg2 = tf->regs[6]; // $a2
    u_int arg3 = tf->regs[7]; // $a3

    /* Step 4: Last 2 args are stored in stack at [$sp + 16 bytes], [$sp + 20 bytes]. */
    u_int arg4, arg5;
    /* Exercise 4.2: Your code here. (3/4) */
    arg4 = *((u_int *)tf->regs[29] + 4); // 从栈上获取
    arg5 = *((u_int *)tf->regs[29] + 5);/* Step 5: Invoke 'func' with retrieved arguments
and store its return value to $v0 in 'tf'.
     */
    /* Exercise 4.2: Your code here. (4/4) */
    tf->regs[2] = func(arg1, arg2, arg3, arg4, arg5); // 把返回值存到$v0
}
```

2.1.4 基础系统调用函数

- `kern/env.c/envid2env`函数：实现通过一个进程的id获取该进程控制块的功能。

```
int envid2env(u_int envid, struct Env **penv, int checkperm) {
    struct Env *e;
    /* Exercise 4.3: Your code here. (1/2) */
    if (envid == 0) {
        *penv = curenv; // envid为0，直接返回当前环境
        return 0;
    }
    e = &envs[ENVX(envid)]; // 使用ENVX宏从envid中提取索引，获取对应的环境结构体

    if (e->env_status == ENV_FREE || e->env_id != envid) {
        return -E_BAD_ENV;
    }
    /* Exercise 4.3: Your code here. (2/2) */
    // 确保当前进程只能访问自身或其子进程的元数据，防止越权访问其他无关进程
    if (checkperm && e != curenv && e->env_parent_id != curenv->env_id) {
        return -E_BAD_ENV;
    }
    *penv = e; // 把e赋值给返回用的指针
    return 0;
}
```

- `sys_mem_alloc`函数：为`envid`所对应进程的虚拟地址空间`va`分配实际的物理页。

```
int sys_mem_alloc(u_int envid, u_int va, u_int perm) {
    struct Env *env;
    struct Page *pp;
    /* Exercise 4.4: Your code here. (1/3) */
    if (is_illegal_va(va)) { // 检查地址合法性
        return -E_INVAL;
    }
    /* Exercise 4.4: Your code here. (2/3) */
    try(envid2env(envid, &env, 1)); // 获取进程控制块
    /* Step 3: Allocate a physical page using 'page_alloc'. */
    /* Exercise 4.4: Your code here. (3/3) */
    try(page_alloc(&pp)); // 分配物理页
    return page_insert(env->env_pgdir, env->env_asid, pp, va, perm);
}
```

- `sys_mem_map`函数：将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去。实现方式是将这两块地址映射到同一块物理页面。

```
int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm) {
    struct Env *srcenv;
    struct Env *dstenv;
    struct Page *pp;

    /* Exercise 4.5: Your code here. (1/4) */
    if (is_illegal_va(srcva) || is_illegal_va(dstva)) {
        return -E_INVAL; // 判断两虚拟地址是否合法
    }
    /* Exercise 4.5: Your code here. (2/4) */
    try(envid2env(srcid, &srcenv, 1)); // 获取进程控制块
    /* Exercise 4.5: Your code here. (3/4) */
    try(envid2env(dstid, &dstenv, 1)); // 获取进程控制块
    /* Exercise 4.5: Your code here. (4/4) */
    pp = page_lookup(srcenv->env_pgdir, srcva, NULL);
    if (pp == NULL) {
        return -E_INVAL;
    }
    return page_insert(dstenv->env_pgdir, dstenv->env_asid, pp, dstva, perm);
}
```

- `sys_mem_numap`：是解除某个进程地址空间虚拟内存和物理内存之间的映射关系。

```
int sys_mem_unmap(u_int envid, u_int va) {
    struct Env *e;
    /* Exercise 4.6: Your code here. (1/2) */
    if (is_illegal_va(va)) {
        return -E_INVAL; // 检查虚拟地址合法性
    }
    /* Exercise 4.6: Your code here. (2/2) */
    try(envid2env(envid, &e, 1)); // 获得进程控制块
    /* Step 3: 解除 envid 地址空间中虚拟地址 va 处的物理页映射 */
    page_remove(e->env_pgdir, e->env_asid, va); // 移除映射
    return 0;
}
```

- `sys_yield`：实现用户进程主动让出执行权。调用该系统调用会切换到其他进程。

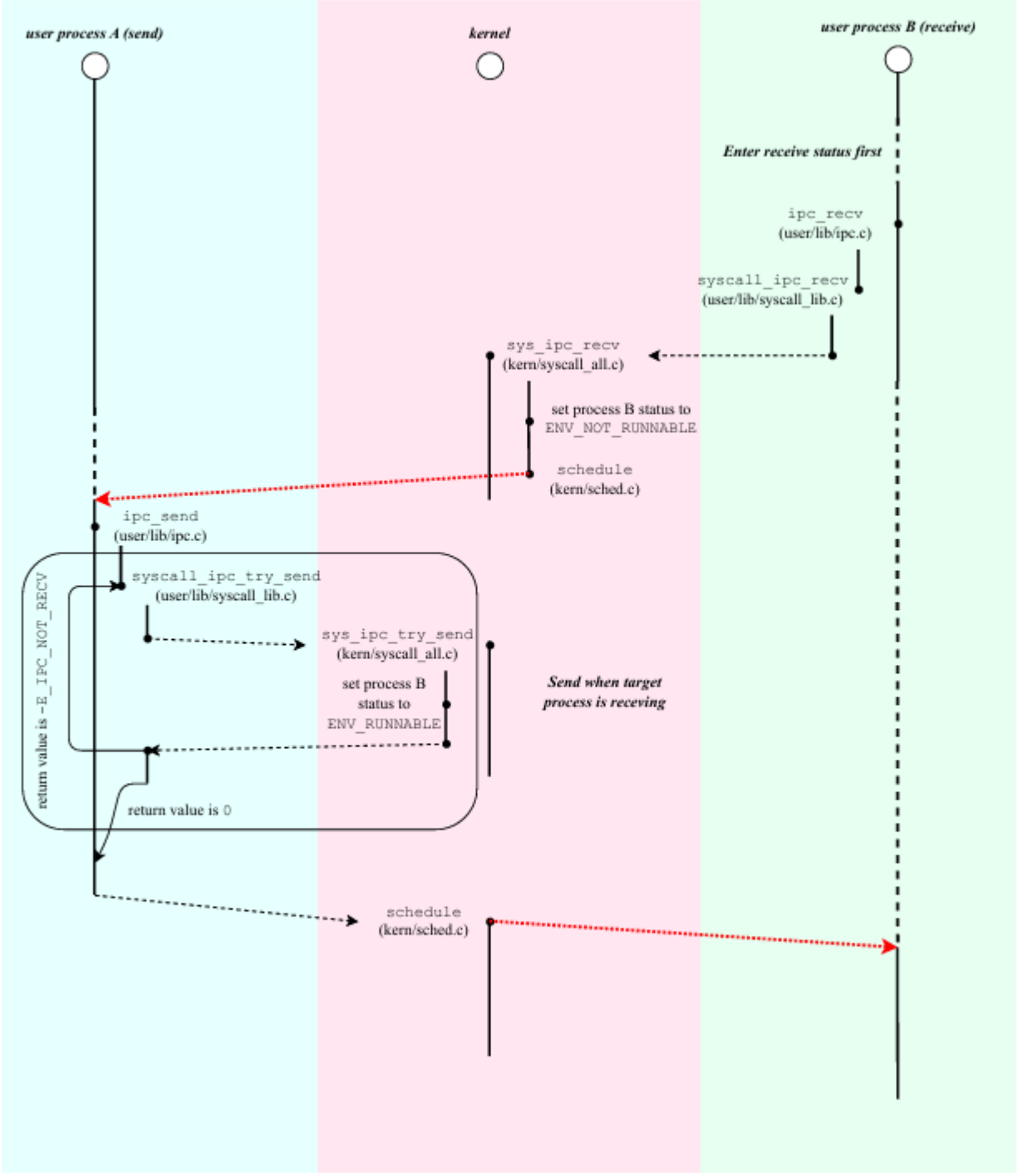
```
void __attribute__((noreturn)) sys_yield(void) {
    // __attribute__((noreturn)) 是一个GCC特性，表示该函数不会返回给调用者
    /* Exercise 4.7: Your code here. */
    schedule(1); // 将 yield 置 1
}
```

2.2 IPC

2.2.1 进程间的通信机制

- 进程间通信即不同进程间的数据传递。实现数据传递的方式有两种：
    - 借助内核空间
    - 共享内存

2.2.2 进程通信的整体流程



- 要点：
  - 接收进程设置状态并阻塞
  - 发送进程写入数据或设置共享页面后将 接收进程设置为可运行状态
- 理解进程控制块中 IPC 相关属性

```
struct Env {
    // lab 4 IPC
    u_int env_ipc_value; // data value sent to us
    u_int env_ipc_from; // envid of the sender
    u_int env_ipc_recving; // env is blocked receiving
    u_int env_ipc_dstva; // va at which to map received page
    u_int env_ipc_perm; // perm of page mapping received
 };
```

- env_ipc_value 表示进程传递的数值。当进程间需要传递一个数值时，由发送进程通过系统调用将这个 值写入接收进程控制块
- env_ipc_value ，接收进程访问自身控制块即可得到该值
- env_ipc_from 表示发送方的进程 ID
- env_ipc_recving 表示进程的接收状态。其值为 0 或 1 。值为 1 时，表示等待接受数据中；值为 0 时，表 示不可接受数据。 env_ipc_dstva 表示进程接收到的物理页面需要与自身的哪个虚拟页面完成映射
- env_ipc_perm 表示传递的物理页面的权限位设置

2.2.3 系统调用消息传递

- 系统调用：接收消息

```
int sys_ipc_recv(u_int dstva) {
    if (dstva != 0 && is_illegal_va(dstva)) { // 如果进程需要通过共享内存接收数据，即`dstva`不为0
        return -E_INVAL; // 检查地址合法性
    }
    /* Exercise 4.8: Your code here. (1/8) */
    curenv->env_ipc_recving = 1; // 将当前进程的env_ipc_recving标志设为1，表示该进程正在等待接收消息
    /* Exercise 4.8: Your code here. (2/8) */
    curenv->env_ipc_dstva = dstva; // 保存接收消息的目标地址到当前进程的env_ipc_dstva字段
    /* Exercise 4.8: Your code here. (3/8) */ // 修改进程状态
    curenv->env_status = ENV_NOT_RUNNABLE;
    TAILQ_REMOVE(&env_sched_list, curenv, env_sched_link);
    /* Step 5: Give up the CPU and block until a message is received. */
    ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0; // 设置陷阱帧(Trapframe)中的返回值即$v0寄存器设置为0，表示成功接收消息
    schedule(1);
}
```

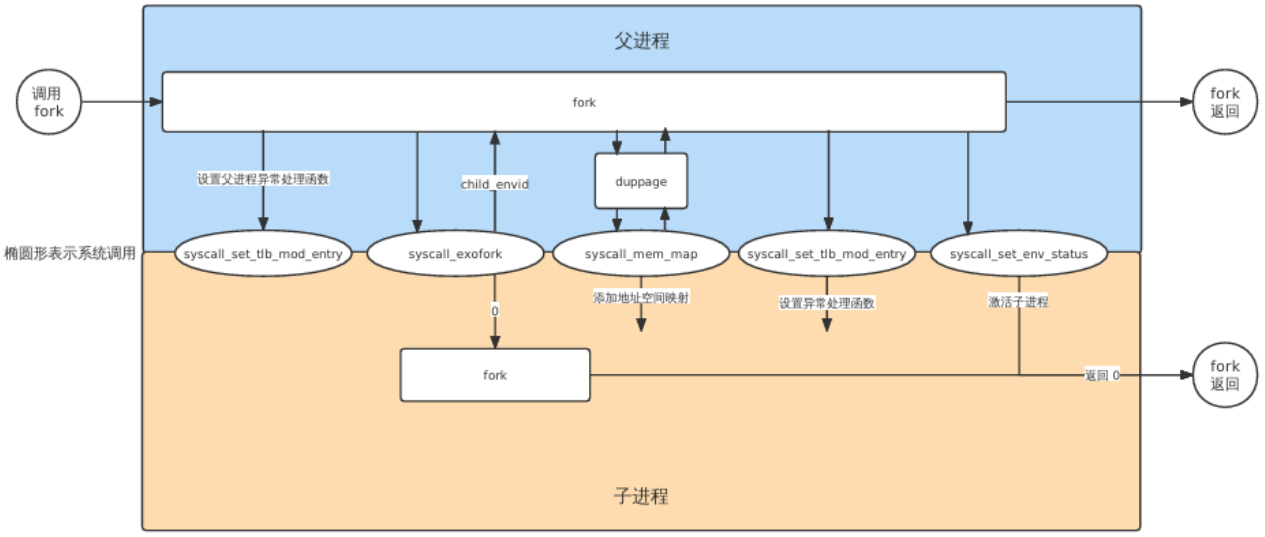
- 系统调用：发送消息（若srcva为0，则只传递value不传递物理页）

```
int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm) {
    struct Env *e;
    struct Page *p;
    /* Exercise 4.8: Your code here. (4/8) */
    if (srcva != 0 && is_illegal_va(srcva)) { // 如果进程需要通过共享内存接收数据，即`srcva`不为0
        return -E_INVAL; // 检查地址合法性
    }
    /* Exercise 4.8: Your code here. (5/8) */
    try(envid2env(envid, &e, 0)); // 获得接收方的进程控制块
    /* Exercise 4.8: Your code here. (6/8) */
```

```
    if (e->env_ipc_recving == 0) {
        return -E_IPC_NOT_RECV; // 检查目标进程是否在等待接收
    }
    // 设置目标进程的IPC字段，设置接收到的值、发送方ID、页面权限、清除接收标志。
    e->env_ipc_value = value;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_perm = PTE_V | perm;
    e->env_ipc_recving = 0;
    /* Exercise 4.8: Your code here. (7/8) */
    e->env_status = ENV_RUNNABLE; // 唤醒目标进程
    TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link); // 将接收进程加入调度队列
    if (srcva != 0) { // 如果进程需要通过共享内存接收数据，即`srcva`不为0
        /* Exercise 4.8: Your code here. (8/8) */
        p = page_lookup(curenv->env_pgdir, srcva, NULL); // 查找源地址对应的物理页
        if (p == NULL) {
            return -E_INVAL; // 源地址对应的物理页是否存在
        }
        try(page_insert(e->env_pgdir, e->env_asid, p, e->env_ipc_dstva, perm)); // 将物理页映射到目标进程的接收地址
    }
    return 0;
}
```

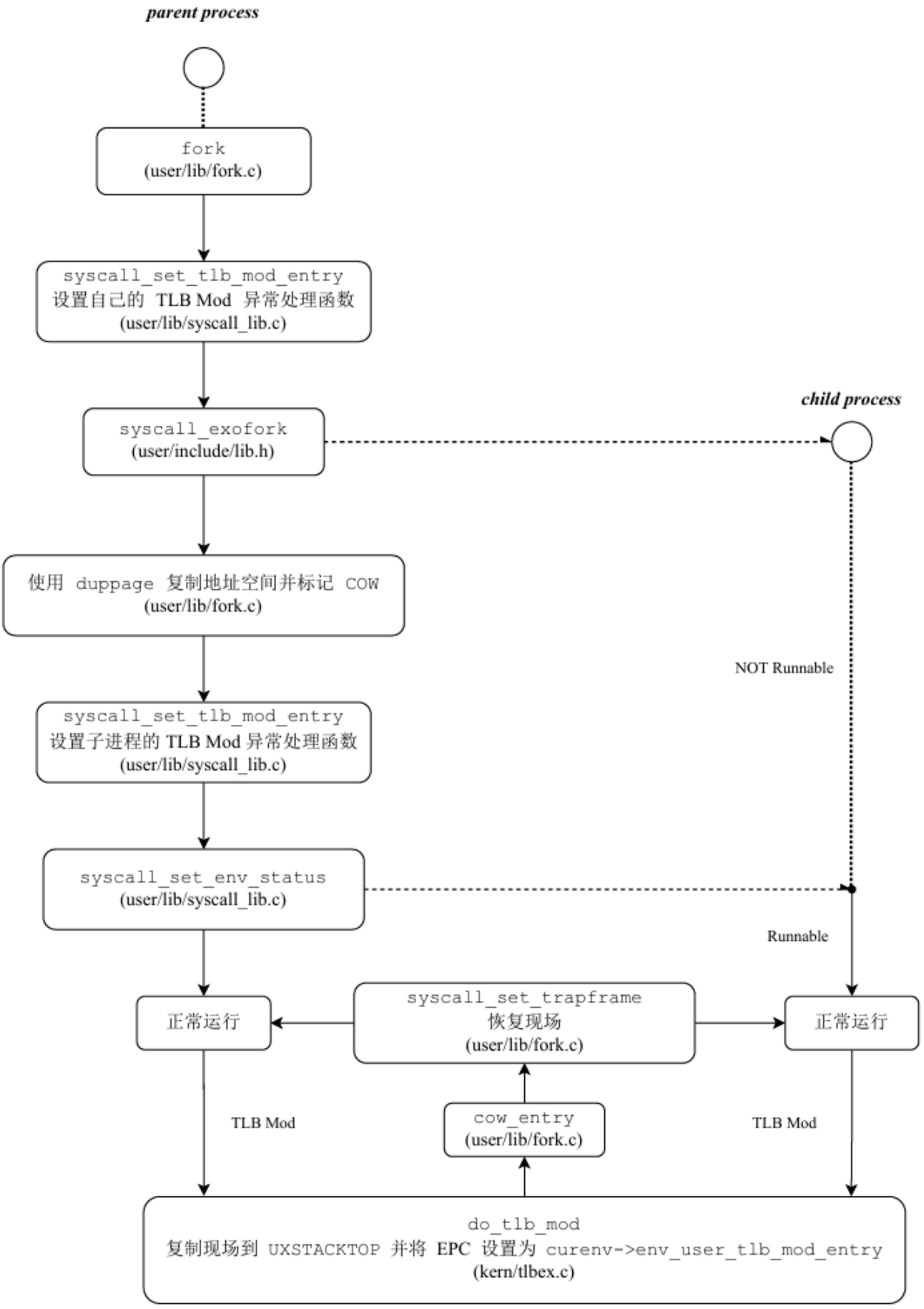
2.3 fork

2.3.1 fork整体流程



- 父进程调用 `fork` 函数:
  - 调用 `syscall_set_tlb_mod_entry` 设置当前进程 `TLB Mod` 异常处理函数
  - 调用 `syscall_exofork` 函数为子进程分配进程控制块
  - 将当前进程低于 `USTACKTOP` 的虚拟页标记为写时复制映射到子进程虚拟地址空间
  - 调用 `syscall_set_tlb_mod_entry` 设置子进程 `TLB Mod` 异常处理函数

    - 调用 `syscall_set_env_status` 进程使子进程处于就绪状态，并将子进程加入调度队列 `env_sched_list` 中
- 子进程通过 `env_run` 函数启动运行
    - 最终通过调用 `ret_from_exception` 从内核栈的 `Trapframe` 中恢复上下文
    - 子进程从 `fork` 函数中的 `syscall_exofork` 调用处返回，返回值为 0 子进程设置 `env` 变量，从 `fork` 中返回，返回值为 0
- fork 流程图


parent process
fork
(user/lib/fork.c)
syscall_set_tlb_mod_entry
设置自己的 TLB Mod 异常处理函数
(user/lib/syscall_lib.c)
syscall_exofork
(user/include/lib.h)
child process
使用 duppage 复制地址空间并标记 COW
(user/lib/fork.c)
NOT Runnable
syscall_set_tlb_mod_entry
设置子进程的 TLB Mod 异常处理函数
(user/lib/syscall_lib.c)
syscall_set_env_status
(user/lib/syscall_lib.c)
Runnable
正常运行
syscall_set_trapframe
恢复现场
(user/lib/fork.c)
正常运行
TLB Mod
cow_entry
(user/lib/fork.c)
TLB Mod
do_tlb_mod
复制现场到 UXSTACKTOP 并将 EPC 设置为 curenv->env_user_tlb_mod_entry
(kern/tlbex.c)

2.3.2 写时复制机制

- 通过 fork 函数创建的子进程是对父进程的复制，如果这时将父进程地址空间中的内容全部复制到新的物理页，将会消耗大量的物理内存。可以选择让父子进程暂时共享内存，当重新写入数据时才将该段内存分离。这 就是**写时复制** 。
- 在MIPS中，当程序尝试写入的虚 拟页对应的`TLB`项没有`PTE_D`标志时，会触发`TLBMod`异常，使系统陷入到内核中。由于对 应的异常处理函数是由内核设置的，因此我们可以使用它来实现写时复制机制。
- 我们可以将写时复制界面的`PTE_D`标志置为0。当进程读这个页面时，不会出现问题。但当 进程尝试写这个页面时，由于`PTE_D` 为 0，所以会触发`TLB Mod` 异常。此时，我们就可以在 异常处理函数中，将虚拟页映射到一个新的物理页，然后将旧的物理页的内容复制到新的物理页 中。这样，两个进程的虚拟页就会各自映射到不同的物理页了。之后进程再对这个虚拟页进行修 改，就不会有任何问题了。
- 同时，为了区分真正的“只读”页面和“写时复制”页面，我们需要利用`TLB`项中的软件 保留位，引入新的标志位`PTE_COW`。在`TLBMod`的异常处理函数中，如果触发该异常的页面的 `PTE_COW` 为 1，我们就需要进行上述的写时复制处理，即分配一页新的物理页，将写时复制页面 的内容拷贝到这一物理页，然后映射给尝试写入该页面的进程

2.3.3 创建进程代码细节

- `sys_exofork` 函数：实现了子进程的创建。该系统调用是父子进程正真“分支”的地方。实现了“一次调用，两次返回”。

```
int sys_exofork(void) {
    struct Env *e;
    /* Exercise 4.9: Your code here. (1/4) */
    try(env_alloc(&e, curenv->env_id)); // 分配一个新的进程控制块
    /* Exercise 4.9: Your code here. (2/4) */
    e->env_tf = *((struct Trapframe *)KSTACKTOP - 1); // 将父进程的上下文复制到子进程中
    /* Exercise 4.9: Your code here. (3/4) */
    e->env_tf.regs[2] = 0; // 将子进程上下文的返回值设为0
    /* Exercise 4.9: Your code here. (4/4) */
    e->env_status = ENV_NOT_RUNNABLE; // 避免立刻执行
    e->env_pri = curenv->env_pri; // 继承优先级
    return e->env_id; // 父进程获得的返回值
}
```

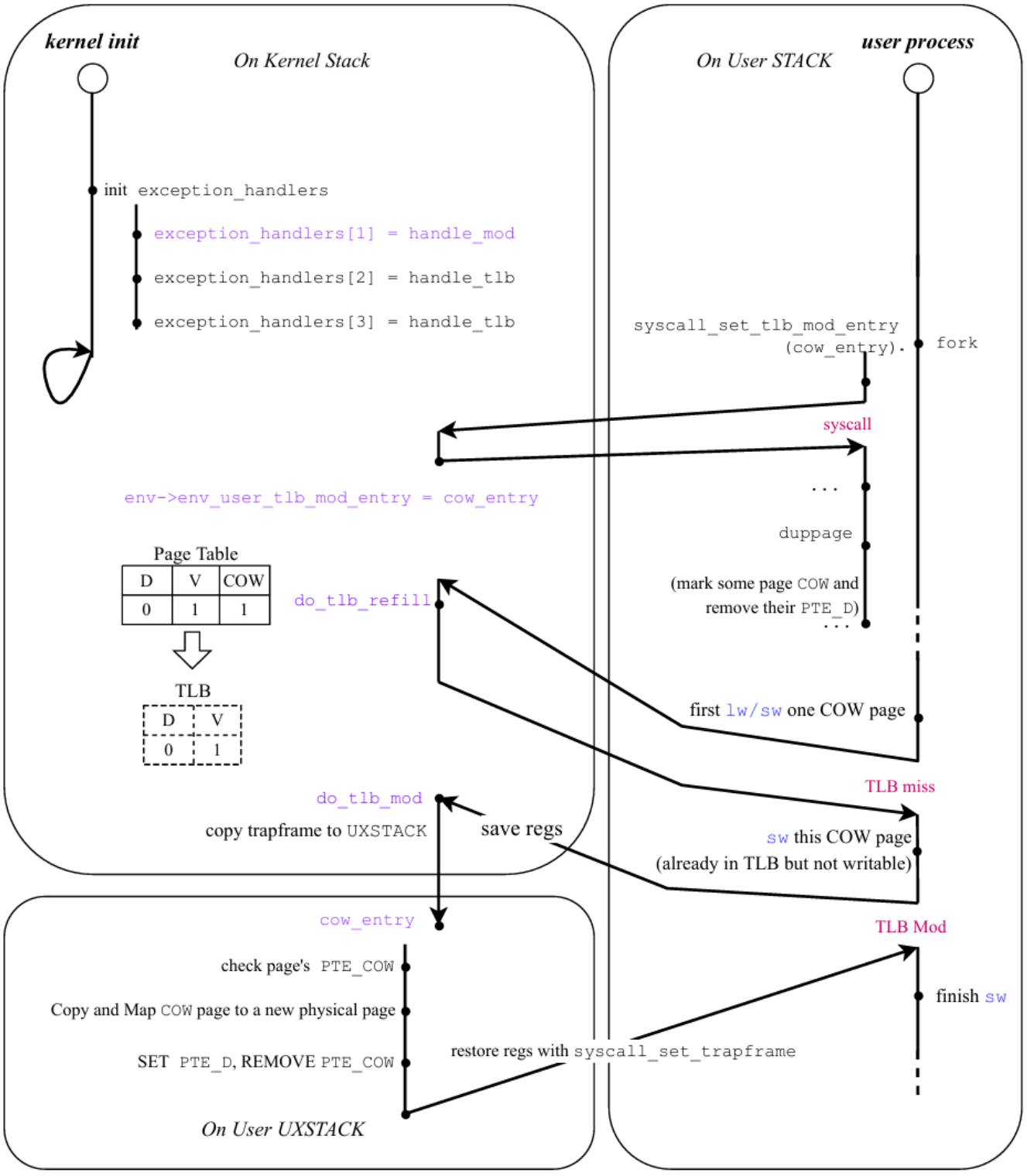
- `duppage`函数`user/fork.c`：将父进程地址空间中需要与子进程共享的页面映射给子进程。
  - 情况1：只读页面或共享库页面或已经是COW页面
  - 情况2：可写且非共享的页面
- 在这两种情况下， `PTE_D` 标志均为 0 。因此，我们需要添加一个标志位 `PTE_COW` ，用来区分这两种情况。
- 写时复制机制的处理流程
  - 首先在用户态进程中因为写入 `COW` 页面引发页写入异常
  - 然后陷入内核态处理，最终分发到 `do_tlb_mod` 中
  - `do_tlb_mod` 保存现场后，将 `epc` 设置为用户的页写入异常处理程序并向其中传入现场，然后 退出内核。
  - 在 " 恢复现场 " 时恢复到了异常处理程序中，此时已经转到用户态，
  - 完成处理后，进行 `syscall_set_trapframe` 又进内核态处理，希望回到原位。

    - 最终是恢复到了触发异常的那条 `epc` 。
- 注意：其中 `(*vpt) + (va >> 12)` 或 `vpt[va >> 12]` 即 `vpt[VPN(va)]`

```
static void duppage(u_int envid, u_int vpn) {
    int r;
    u_int addr;
    u_int perm;
    /* Exercise 4.10: Your code here. (1/2) */
    addr = vpn << PGSHIFT; // 将虚拟页号vpn转换为虚拟地址addr
    perm = vpt[vpn] & ((1 << PGSHIFT) - 1); // 从页表项vpt[vpn]中提取页面权限位perm
    /* Exercise 4.10: Your code here. (2/2) */
    // 对于不可写(PTE_D未设置)或是共享库(PTE_LIBRARY)或已经是COW(PTE_COW)页面
    if ((perm & PTE_D) == 0 || (perm & PTE_LIBRARY) || (perm & PTE_COW)) {
        if ((r = syscall_mem_map(0, (void *)addr, envid, (void *)addr, perm)) < 0) {
            // 直接将页面映射到子进程，保持原权限不变
            user_panic("user panic mem map error: %d", r);
        }
    } else { // 对于可写非共享且不是写时复制的页面
        // 先映射到子进程，防止此时触发写时复制
        if ((r = syscall_mem_map(0, (void *)addr, envid, (void *)addr, (perm & ~PTE_D) |
PTE_COW)) < 0) { // 更新权限，取消可写位，加上COW位
            user_panic("user panic mem map error: %d", r);
        }
        // 然后修改父进程的映射，也设置为只读+COW
        if ((r = syscall_mem_map(0, (void *)addr, 0, (void *)addr, (perm & ~PTE_D) |
PTE_COW)) < 0) {
            user_panic("user panic mem map error: %d", r);
        }
    }
}
```

2.3.4 页面写入异常

- 页写入异常处理流程图：



- `do_tlb_mod` 函数：在用户空间处理 TLB 修改异常时设置适当的堆栈和跳转到用户注册的处理程序。

```
void do_tlb_mod(struct Trapframe *tf) {
    struct Trapframe tmp_tf = *tf; // 首先复制当前的陷阱帧（Trapframe），保存所有寄存器和状态信息。
    if (tf->regs[29] < USTACKTOP || tf->regs[29] >= UXSTACKTOP) { // 检查当前栈指针（寄存器29，通常是SP）是否在用户栈范围内。如果不在，则将其设置为异常栈顶（UXSTACKTOP）
        tf->regs[29] = UXSTACKTOP;
    }
    tf->regs[29] -= sizeof(struct Trapframe);
    *(struct Trapframe *)tf->regs[29] = tmp_tf; // 在用户栈上预留空间并保存原始的陷阱帧
    Pte *pte;
```

```
    page_lookup(cur_pgdir, tf->cp0_badvaddr, &pte); // 查找引发异常的虚拟地址对应的页表项（PTE）
    if (curenv->env_user_tlb_mod_entry) { // 如果注册了用户处理函数，设置返回地址并跳转
        tf->regs[4] = tf->regs[29];  // 保存栈指针到 $a0（MIPS 的 $a0 是第一个参数寄存器）
        tf->regs[29] -= sizeof(tf->regs[4]); // 调整栈指针（预留空间）
        /* Exercise 4.11: Your code here. */
        tf->cp0_epc = curenv->env_user_tlb_mod_entry;  // 设置返回地址为用户处理程序
    } else {
        panic("TLB Mod but no user handler registered");
    }
}
```

- `sys_se t_tlb_mod_entry` 函数：设置 TLB Mod 异常处理函数。

```
int sys_set_tlb_mod_entry(u_int envid, u_int func) {
    struct Env *env;
    /* Exercise 4.12: Your code here. (1/2) */
    try(envid2env(envid, &env, 1)); // 通过 envid 获取进程控制块
    /* Exercise 4.12: Your code here. (2/2) */
    env->env_user_tlb_mod_entry = func; // 在进程控制块中设置TLB Mod异常处理函数为func
    return 0; // 成功返回0
}
```

- `cow_entry` 函数`user/fork.c`：页写入异常处理时会返回到用户空间的cow_entry 函数 。

```
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
    u_int va = tf->cp0_badvaddr; // 获取触发异常的虚拟地址
    u_int perm;
    perm = PTE_FLAGS(vpt[VPN(va)]); // 获取触发页故障的虚拟地址对应的页表项权限
    if ((perm & PTE_COW) == 0) { // 检查权限是否含有 PTE_COW，即检查va是否为写时复制页面
        user_panic("PTE_COW not found, va=%08x, perm=%08x", va, perm);
    }
    /* Exercise 4.13: Your code here. (2/6) */
    perm = (perm & ~PTE_COW) | PTE_D; // 更新页权限，清除 PTE_COW 标志（不再需要写时复制），添加
PTE_D（脏页标志），表示页可写
    /* Exercise 4.13: Your code here. (3/6) */
    syscall_mem_alloc(0, (void *)UCOW, perm); // 系统调用，在用户空间分配物理页，暂时映射到UCOW处
    /* Exercise 4.13: Your code here. (4/6) */
    memcpy((void *)UCOW, (void *)ROUNDDOWN(va, PAGE_SIZE), PAGE_SIZE); // 复制原页内容到新页，
ROUNDDOWN(va, PAGE_SIZE)将 va 对齐到页起始地址
    /* Exercise 4.13: Your code here. (5/6) */
    syscall_mem_map(0, (void *)UCOW, 0, (void *)va, perm); // 映射新页到原故障地址
    /* Exercise 4.13: Your code here. (6/6) */
    syscall_mem_unmap(0, (void *)UCOW); // 解除临时映射
    int r = syscall_set_trapframe(0, tf); // 使用系统调用恢复上下文
    user_panic("syscall_set_trapframe returned %d", r);
}
```

- `sys_set_env_status` 函数：由于 `TLB Mod` 异常由用户态处理，因此需要一个系统调用为当前进程设置其 `TLB Mod` 异常处理函数。该系统 调用即 `sys_set_tlb_mod_entry` 。

```
int sys_set_env_status(u_int envid, u_int status) {
```

```
    struct Env *env;
    /* Exercise 4.14: Your code here. (1/3) */
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE) { // 检查 status 是否合法
        return -E_INVAL;
    }
    /* Exercise 4.14: Your code here. (2/3) */
    try(envid2env(envid, &env, 1)); // 通过 envid 获取进程控制块
    /* Exercise 4.14: Your code here. (3/3) */
    if (status == ENV_RUNNABLE && env->env_status != ENV_RUNNABLE) {
        TAILQ_INSERT_TAIL(&env_sched_list, env, env_sched_link);
    } else if (status == ENV_NOT_RUNNABLE && env->env_status != ENV_NOT_RUNNABLE) {
        TAILQ_REMOVE(&env_sched_list, env, env_sched_link);
    }
    /* Step 4: Set the 'env_status' of 'env'. */
    env->env_status = status; // 更新进程状态
    /* Step 5: Use 'schedule' with 'yield' set if ths 'env' is 'curenv'. */
    if (env == curenv) {
        schedule(1); // 避免当前进程继续占用 CPU 资源
    }
    return 0;
}
```

2.3.5 user/fork 的实现

```
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)); // 设置当前进程的 TLB 异常处理函数
    }

    child = syscall_exofork(); // 为子进程分配进程控制块
    if (child == 0) {
        env = envs + ENVX(syscall_getenvid()); // 可以得到子进程控制块
        return 0; // 子进程的返回
    }
    /* Exercise 4.15: Your code here. (1/2) */ // 复制地址空间（COW 机制核心）
    for (i = 0; i < PDX(UXSTACKTOP); i++) { // UXSTACKTOP一下的部分都调用duppage复制了一遍
        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));
                }
            }
        }
    }
    /* Exercise 4.15: Your code here. (2/2) */
    syscall_set_tlb_mod_entry(child, cow_entry); // 子进程也需要设置相同的 COW 页错误处理能力
```

```
    syscall_set_env_status(child, ENV_RUNNABLE); // 将子进程状态从 ENV_NOT_RUNNABLE 改为
ENV_RUNNABLE
    return child; // 父进程返回子进程的 envid
}
```

2.4 一些常见宏与定义

```
// include/mmu.h
#define NASID 256
#define PAGE_SIZE 4096
#define PTMAP PAGE_SIZE
#define PDMAP (4 * 1024 * 1024) // bytes mapped by a page directory entry
#define PGSHIFT 12
#define PDSHIFT 22 // log2(PDMAP)
#define PDX(va) ((((u_long)(va)) >> PDSHIFT) & 0x03FF) // 获得页目录索引
#define PTX(va) ((((u_long)(va)) >> PGSHIFT) & 0x03FF) // 获得页表项索引
#define PTE_ADDR(pte) (((u_long)(pte)) & ~0xFFF)
#define PTE_FLAGS(pte) (((u_long)(pte)) & 0xFFF) // 获得页表项权限
// Page number field of an address
#define PPN(pa) (((u_long)(pa)) >> PGSHIFT)
#define VPN(va) (((u_long)(va)) >> PGSHIFT) // 获取虚页号
// Page Table/Directory Entry flags
#define PTE_HARDFLAG_SHIFT 6

//user/include/lib.h
#define vpt ((const volatile Pte *)UVPT) // 获取页表项
#define vpd ((const volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT))) // 获取页目录项
#define envs ((const volatile struct Env *)UENVS)
#define pages ((const volatile struct Page *)UPAGES)
```

3. 实验体会

3.1 2022年

Lab4-1-Exam

```
// user/lib.h
int syscall_try_acquire_console(void);
int syscall_release_console(void);

// user/syscall_lib.c
int syscall_try_acquire_console(void) {
    return msyscall(SYS_try_acquire_console);
}

int syscall_release_console(void) {
    return msyscall(SYS_release_console);
}

// include/syscall.h
enum {
    SYS_try_acquire_console,
```

```
    SYS_release_console,
    ...
};

// kern/syscall_all.c
void *syscall_table[MAX_SYSNO] = {
    [SYS_try_acquire_console] = sys_try_acquire_console,
    [SYS_release_console] = sys_release_console,
    ...
}

// kern/syscall_all.c
...

extern struct Env *curenv;
int lock = -1;

void sys_putchar(int c) {
    if (lock != curenv->env_id) {
        printcharc((char)c);
    }
    return;
}

int sys_try_acquire_console(void) {
    if (lock == -1) {
        lock == curenv->env_id;
        return 0;
    }
    return -1;
}

int sys_release_console(void) {
    if (lock == curenv->env_id) {
        lock == -1;
        return 0;
    }
    return -1;
}
```

Lab4-2-Exam

题目：

题目描述

请你在 `user/fork.c` 中实现下面这个用户函数，并在 `user/lib.h` 中添加相应的声明，使用户程序能使用该函数：

```
// user/lib.h
int make_shared(void *va);
```

- 该函数将当前进程中虚拟地址 va 所属的虚拟页标记为**共享页**，并返回其映射到的物理页的物理地址。
- 在该进程后续执行 fork 时，其共享页应与子进程共享，使得两个进程的地址空间中该页映射到同一个物理页。
- fork **不应**对共享页进行 COW 保护。若父进程或子进程修改了共享页中的数据，随后另一进程读取该页时也会读取到修改后的数据。

实现要求

- 若当前进程的页表中不存在该虚拟页，该函数应首先分配一页物理内存，并将该虚拟页映射到新分配的物理页，使当前进程能够**读写**该虚拟页。若无法分配新的物理页，该函数应返回 -1 表示失败。
- 若 va 不在用户空间中（大于或等于 UTOP），或者当前进程的页表中已存在该虚拟页，但进程对其没有写入权限，则该函数应返回 -1 表示失败，不产生任何影响。
- 除了失败的情况，该函数都应返回 va 所在的虚拟页所映射到的物理页的物理地址。
- 若虚拟页 va 已经为共享页，该函数仍应成功，直接返回对应的物理地址。
- 评测保证调用 `make_shared` 之前，虚拟页 va 没有被 COW 保护。

请注意：

- 进程的共享页作为进程的状态，在执行 fork 后创建的子进程中仍应保持。即：若虚拟页 va 是父进程中的共享页，则在子进程中 va 仍然是子进程的共享页。若子进程再执行一次 fork，父进程、子进程、子进程的子进程都能通过 va 共享同一个物理页。
- 作为参数传入 `make_shared` 的 va 不一定是页对齐的，但**返回的物理地址一定是页对齐的。**

作用：将当前进程的虚拟地址 `va` 对应的物理页标记为共享内存，并返回其物理地址，以便其他进程可以映射同一块物理内存，实现进程间共享数据。

```
int make_shared(void *va) { // 将虚拟地址 va 对应的页标记为共享内存（PTE_LIBRARY）
    u_int perm = PTE_D | PTE_V;
    if (!(vpd[va >> 22] & PTE_V) || !(vpt[va >> 12] & PTE_V)) { //当前进程的页表中不存在该虚拟页
        if (syscall_mem_alloc(0, ROUNDDOWN(va, PAGE_SIZE), perm) != 0) {
            //将envid设为0，表示默认curenv
            return -1;
        }
    }
    perm = vpt[VPN(va)] & 0xfff; //获得va的perm
    if (va >= (void *)UTOP ||
```

```
        ((vpd[va >> 22] & PTE_V) && (vpt[va >> 12] & PTE_V) && !(perm & PTE_D))) { // 检查共享条件
        return -1;
    }
    perm = perm | PTE_LIBRARY; //  设置共享标志 PTE_LIBRARY
    u_int addr = VPN(va) * BY2PG;  // 虚拟页号对应的起始地址（VPN(va) * BY2PG）
    if (syscall_mem_map(0, (void *)addr, 0, (void *)addr, perm) != 0) { // 因为权限改变了，所以重新自己映射回自己
            return -1;
    }
    return ROUNDDOWN(vpt[VPN(va)] & (~0xfff), PAGE_SIZE); // 返回va对应物理页的起始地址
}
```

3.2 Yang

3.2.1 Lab4-1-Exam

主要考察添加一个系统调用的步骤，如下以用户进程调用函数 `user_lib_func(u_int whom, u_int val, const void \*srcva, u_int perm)` 过程中，会使用到系统调用 `syscall_func` 为例归纳步骤：

1. 在 `user/include/lib.h` 中添加：

   `void user_lib_func(u_int whom, u_int val, const void *srcva, u_int perm);`

   `void syscall_func(u_int envid, u_int value, const void *srcva, u_int perm);`

2. 在 `user/lib/syscall_lib.c` 中添加：

   ```
   void syscall_func(u_int envid, u_int value, const void *srcva, u_int perm) {
       msyscall(SYS_func, envid, value, srcva, perm);
   }
   ```

3. 在 `user/lib` 中的使用 `user_lib_func` 函数的目标文件中编写实现该函数（注意在该函数过程中会调用 `syscall_func` 函数）

4. 在 `include/syscall.h` 中的 `enum` 的 `MAX_SYSNO` 前面加上 `SYS_func,`

5. 在 `kern/syscall_all.c` 的 `void *syscall_table[MAX_SYSNO]` 的最后加上 `[SYS_func] = sys_func,`（注意最后有逗号）

6. 在 `kern/syscall_all.c` 的 `void *syscall_table[MAX_SYSNO]` 的前面具体编写实现函数

   ```
   int sys_func(u_int envid, u_int value, u_int srcva, u_int perm) {
       //......
   }
   ```

3.2.2 Lab4-1-Extra

- Lab4-1-extra需要实现一种广播通讯机制 `ipc_broadcast` 函数。这段代码实现了一个名为 `sys_ipc_try_broadcast` 的系统调用，其功能是向当前进程的所有后代进程（即直接或间接由当前进程创建的子进程、孙子进程等）广播一条 IPC（进程间通信）消息。
- 主要在于引入全局变量 `envs` 数组，然后遍历判断后代进程。

```
c
```

```c
//kern/syscall_all.c
extern struct Env envs[NENV]; //注意 extern! !
int sys_ipc_try_broadcast(u_int value, u_int srcva, u_int perm) {
    struct Env *e;
    struct Page *p;

    /* Step 1: Check if 'srcva' is either zero or a legal address. */
    /* 抄的sys_ipc_try_send */
    if (srcva != 0 && is_illegal_va(srcva)) {
        return -E_IPC_NOT_RECV;
    }

    /* 函数核心：遍历envs找后代进程 */
    int signal[NENV];
    for (u_int i = 0; i < NENV; i++) {
        if (curenv->env_id == envs[i].env_parent_id) {
            signal[i] = 1;
        } else {
            signal[i] = 0;
        }
    }
    int flag = 0;
    while(flag == 0) {
        flag = 1;
        for (u_int i = 0; i < NENV; i++) {
            if (signal[i] == 1) {
                    for (u_int j = 0; j < NENV; j++) {
                    if (signal[j] == 0 && envs[i].env_id == envs[j].env_parent_id) {
                        signal[j] = 1;
                        flag = 0;
                    }
                }
            }
        }
    }

    /* Step 3: Check if the target is waiting for a message. */
    /* 基于sys_ipc_try_send修改 */
    for (u_int i = 0; i < NENV; i++) {
        if(signal[i] == 1) {
            e = &(envs[i]);
            /* 以下都是抄的sys_ipc_try_send */
            if (e->env_ipc_recving == 0) {
                return -E_IPC_NOT_RECV;
            }
            e->env_ipc_value = value;
            e->env_ipc_from = curenv->env_id;
            e->env_ipc_perm = PTE_V | perm;
            e->env_ipc_recving = 0;
            e->env_status = ENV_RUNNABLE;
            TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
            if (srcva != 0) {
                p = page_lookup(curenv->env_pgdir, srcva, NULL);
```

```
                if(p == NULL) return -E_INVAL;
                if (page_insert(e->env_pgdir, e->env_asid, p, e->env_ipc_dstva, perm)
!= 0) {
                    return -E_INVAL;
                }
            }
        }
    }
    return 0;
}
```

- 有一个 `env = envs + ENVX(envid)`；可以由 envid得到 env。

3.2.3 Lab4-2-Exam

- 考察：系统调用+fork+ipc
- 这段代码实现了一个屏障（Barrier）同步机制，用于协调多个进程（或线程）在某个执行点等待，直到所有参与的进程都到达该点后才能继续执行。

```
u_int sys_barrier_wait(u_int* p_barrier_num, u_int* p_barrier_useful) {
    static u_int env_not[100];
    static u_int N = 0;
    static u_int num = 0;
    static u_int useful = 0;
    if ((*p_barrier_num) > N) { // 若满足条件，更新外部传入值
        N = (*p_barrier_num);
        num = N;
        useful = (*p_barrier_useful);
    }
    if (useful == 1) {
        for (u_int i = 0; i < N - num; i++) {
            if (env_not[i] == curenv->env_id) {
                    return ENV_NOT_RUNNABLE;
            }
        }
        env_not[N - num] = curenv->env_id;
        num--;
        if (num == 0) { // 当 num == 0 时，useful 被置 0，屏障失效，后续调用直接返回
                useful = 0;
                return ENV_RUNNABLE;
        }
        return ENV_NOT_RUNNABLE;
    }
    return ENV_RUNNABLE;
}
```

- lab4主要需要掌握：系统调用，IPC通信机制，fork进程创建，页面写入异常处理。 在本次实验中，脑子里一定要清楚现在是在改内核还是改用户，因为之前写的都是内核，而内核函数是不能在用户空间调用的，这一点要注意区分。

3.3 Sin

3.3.1 Lab4-1-Exam

- 题干

> 在Linux中，进程理论上所拥有的权限与执行它的用户的权限相同。进程运行时能够访问哪些资源或文件，不取决于进程文件的属主属组，而是取决于运行该命令的用户身份的 uid/gid，以该身份获取各种系统资源。

所以我们需要完成同一个进程组ID的不同进程的通信。具体而言，需要做到：

1. 在 Env 结构体中添加 `u_int env_gid` 字段代表进程所在的进程组，初始值为 0。
2. 实现一个修改 `gid` 字段的**用户态函数**：`void set_gid(u_int gid);`
3. 实现一个**仅能**向同组块发送消息的**用户态函数**：`int ipc_group_send(u_int whom, u_int val, const void *srcva, u_int perm);`
4. 实现 2、3 两点中对应的**系统调用函数**和调用接口

教程组已经把两个用户态函数实现了，我们只需要考虑系统调用函数 `syscall_set_gid` 和 `syscall_ipc_try_group` 即可

具体要求:

1. 在内核中为每个进程维护进程组ID，并保证每个进程创建时的的组ID为0。
2. 在`user/include/lib.h`中：
   - 添加以下两个用户函数的声明：

```
void set_gid(u_int gid);
int ipc_group_send(u_int whom, u_int val, const void *srcva, u_int perm);
```

   - 添加以下两个系统调用函数的声明：

```
void syscall_set_gid(u_int gid);
int syscall_ipc_try_group_send(u_int whom, u_int val, const void *srcva, u_int perm);
```

3. 在`include/error.h`中，增加以下新错误码`E_IPC_NOT_GROUP`，表示组间通信时通信双方进程的组ID不匹配。

```
#define E_IPC_NOT_GROUP 14
```

4. 两个用户态函数的实现已经给出（请参看实验提供代码部分），你需要将其复制到`user/lib/ipc.c`，具体代码的解释在提示部分给出。
5. 在`include/syscall.h`中：定义两个新的系统调用号。请注意新增系统调用号的位置，应当位于`MAX_SYSNO`之前。
6. 在`user/lib/syscall_lib.c`中：实现上述两个系统调用函数，发起系统调用。
7. 在`kern/syscall_all.c`中：添加两个系统调用在内核中的实现函数。请保证两个函数的定义位于系统调用函数表`void *syscall_table[MAX_SYSNO]`之前。
8. 在`kern/syscall_all.c`中的`void *syscall_table[MAX_SYSNO]`系统调用函数表中，为你定义的系统调用号添加对应的内核函数指针。

9. 编写 `syscall_ipc_try_group_send` 系统调用**在内核中的实现函数**时，判断 `-E_IPC_NOT_RECV` 错误的**优先级**高于 `-E_IPC_NOT_GROUP`

- **实验提供代码**

```
/* copy to user/lib/ipc.c */

void set_gid(u_int gid) {
    // 你需要实现此 syscall_set_gid 系统调用
    syscall_set_gid(gid);
}

int ipc_group_send(u_int whom, u_int val, const void *srcva, u_int perm) {
    int r;
    // 你需要实现此 syscall_ipc_try_group_send 系统调用
    while ((r = syscall_ipc_try_group_send(whom, val, srcva, perm)) != 0) {
        // 接受方进程尚未准备好接受消息，进程切换，后续继续轮询尝试发送请求
        if (r == -E_IPC_NOT_RECV) syscall_yield();
        // 接收方进程准备好接收消息，但非同组通信，消息发送失败，停止轮询，返回错误码 -E_IPC_NOT_GROUP
        if (r == -E_IPC_NOT_GROUP) return -E_IPC_NOT_GROUP;
    }
    // 函数返回0，告知用户成功发送消息
    return 0;
}
```

answer:

```
int sys_ipc_try_group_send(u_int whom, u_int val, const void *srcva, u_int perm) {
  struct Env *e;
  struct Page *p;

  if (srcva != 0 && is_illegal_va((u_int)srcva)) {
    return -E_INVAL;
  }

  if (0 != envid2env(whom, &e, 0)) {
    return -E_BAD_ENV;
  }

  if (e->env_status != ENV_NOT_RUNNABLE) {
    return -E_BAD_ENV;
  }

  if (e->env_ipc_recving == 0) {
    return -E_IPC_NOT_RECV;
  }

  if (e->env_gid != curenv->env_gid) { // 唯一与 ipc_send 不同的地方就在这了
    return -E_IPC_NOT_GROUP;
  }

  e->env_ipc_value = val;
```

```
  e->env_ipc_from = curenv->env_id;
  e->env_ipc_perm = PTE_V | perm;
  e->env_ipc_recving = 0;
  e->env_status = ENV_RUNNABLE;
  TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);

  if (srcva != 0) {
    p = page_lookup(curenv->env_pgdir, (u_int)srcva, NULL);
    if (NULL == p) {
      return -E_INVAL;
    } else {
      try(page_insert(e->env_pgdir, e->env_asid, p, e->env_ipc_dstva, perm));
    }
  }
  return 0;
}

void sys_set_gid(u_int gid) { // 简单赋值
  curenv->env_gid = gid;
  return;
}
```

思路：

- 按照题目给的思路顺下来其实很容易就能写完，顺便也可以用这个题回顾一下系统调用：怎样添加一个新的、可用的系统调用？
  - 首先从内核态出发，编写一个能够完成功能的函数，看一下它都需要什么参数 - `kern/syscall_all.c`
  - 然后在 `void *syscall_table[MAX_SYSNO]` 把函数添加进去，使得 `do_syscall` 函数能跳到这个新函数里：这需要随便写一个字符串当成**系统调用号**，无所谓了 - `kern/syscall_all.c`
  - 找到刚才那个系统调用号的枚举类，把定义加上 - `include/syscall.h`
  - `do_syscall` 不需要变化，然后再上一层是 `msyscall`，它需要一个 `syscall_` 开头的函数进行调用。到这里我们就完成了内核态中所需要做的所有改动
  - 回到用户态，编写一个用户态的 `syscall_new` 函数调用 `msyscall`，同时需要注意参数的第一个参数需要是刚才加进去的调用号 - `users/lib/syscall_lib.c`
  - 最后编写顶层的用户态函数，其中调用 `syscall_new` 函数，用户态工作也就完成了 - `users/某个文件`
  - 最最后别忘了**加上函数声明**：内核态不需要，用户态加在 `users/include/lib.h` 即可

3.3.2 Lab4-1-Extra

题干：

课下我们在 MOS 系统中实现了进程间通信。

现在你需要仿照 `ipc_send` 函数在 `user/lib/ipc.c` 中实现 `ipc_broadcast` 函数，使得调用 `ipc_broadcast` 可以使当前进程向其后代进程（也即当前广播进程的子进程、子进程的子进程、子进程的子进程的子进程...以此类推）发起广播消息，当后代进程进入 recv 后进行发送。

具体要求：

`ipc_broadcast`

需要在 `user/lib/ipc.c` 新增：

```
c
void ipc_broadcast(u_int val, void * srcva, u_int perm);
```

参数：

- `val` ： 进程广播传递的具体数值, 与 `ipc_send` 函数中的定义相同。
- `srcva` ： 进程广播发送页的对应用户虚地址，与 `ipc_send` 函数中的定义相同。
- `perm` : 传递的页面的权限位设置，与 `ipc_send` 函数中的定义相同。

注意点：

- 你可以实现 `syscall_ipc_try_broadcast` 系统调用，使其行为类似于 `syscall_ipc_try_send`，但尝试发送给当前进程的所有后代进程。
- 你也可以尝试在用户空间利用 `envs` 实现相关行为。
- 发送广播消息时，你可以先等待所有后代进程进入接受状态，再统一进行实际传输，也可以依次等待每个后代进程，一旦其处于接受状态，当即对其进行实际传输。

answer：

```c
int sys_ipc_broadcast(u_int val, void *srcva, u_int perm) {
  u_int childs[20];
  for (int i = 0; i < 20; i++) {
    childs[i] = 0;
  }
  struct Env *e;
  struct Page *p;

  // printk("childs ready!\n");

  if (srcva != 0 && is_illegal_va((u_int)srcva)) {
    return -E_INVAL;
  }

  /* Step1: 找到直系的子进程 */ // 难点之一，遍历查找子进程
  for (int i = 0; i < NENV; i++) {
    if (envs[i].env_parent_id == curenv->env_id) {
      for (int j = 0; j < 20; j++) {
        if (childs[j] == 0) {
          childs[j] = envs[i].env_id;
          break;
        }
      }
    }
  }

  /* Step2: 通过 bfs 找到所有子进程的子进程 */
  for (int i = 0; childs[i] != 0; i++) {
    for (int j = 0; j < NENV; j++) {
      if (envs[j].env_parent_id == childs[i]) {
```

```
        for (int k = 0; k < 20; k++) {
          if (childs[k] == envs[j].env_id) {
            break;
          }
          if (childs[k] == 0) {
            childs[k] = envs[j].env_id;
            break;
          }
        }
      }
    }
  }

  /* Step3: 对所有待发送的进程进行发送 */
  for (int i = 0; childs[i] != 0; i++) {
    // printk("%d: %x\n", i, childs[i]);
      sys_ipc_try_send(childs[i], val, srcva, perm);
  }

  return 0;
}
```