一、Lab4 前言

Lab4 主要实现了系统调用，并通过系统调用实现了进程的创建和通信等操作。按照提示编写代码的难度应该不大（除非你的 Lab3 `schedule` 函数有 bug，很可惜我就是这样 `:()` ，所以本次的笔记更多的讨论了一些和实验无关的代码。希望不会显得太啰嗦。

二、系统调用

（1）从一个用户程序引入

在之前的几篇文章中，我们大致循着内核初始化的过程进行分析。可是在这 Lab4 中这一思路就不适用了。因为在本次实验中我们所要实现的，不过是一些由内核提供的，可供用户程序调用的接口而已。这种调用被称为系统调用。

但是为了保持文章行文的一致性，我们还是希望确定一个入口开始讲解。正好在 `mips_init` 中有这样的语句，那我们就从被创建的这个用户程序开始。

```
    // lab4:
    // ENV_CREATE(user_tltest);
```

这里需要插一嘴，在 Lab3 中我们就已经使用 `ENV_CREATE` 完成了一些程序的加载，可你有没有仔细看过被加载的程序的源代码是什么样的？代码在 user/bare 路径下。我们查看其中的 put_a.c 程序

```
void _start() {
    for (unsigned i = 0;; ++i) {
        if ((i & ((1 << 16) - 1)) == 0) {
            // Requires `e->env_tf.cp0_status &= ~STATUS_KUp;` in kernel to work
            *(volatile char *)0xb0000000 = 'a';
            *(volatile char *)0xb0000000 = ' ';
        }
    }
}
```

你会发现这些所谓的程序并没有 `main` 函数，而是 `_start`。实际上和内核一样，我们同样在用于用户程序编译的链接器脚本中将程序入口设定为 `_start`。该脚本为 user/user.lds，其中同样有

```
ENTRY(_start)
```

但是如果你查看一下我们将要查看的 user/tltest.c 程序，就会发现其中又是以 `main` 函数为入口了。这是为什么呢?

这是因为我们在 user/lib/entry.S 中定义了统一的 `_start` 函数。

```
.text
EXPORT(_start)
    lw      a0, 0(sp)
    lw      a1, 4(sp)
    jal     libmain
```

值得注意跳转前的两条指令。这两条指令加载了 argc 和 argv。你可能还记得 Lab3 中 env_alloc 函数的这条语句，它将栈底的 8 个字节留给 argc 和 argv。

```
    // Keep space for 'argc' and 'argv'.
    e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **);
```

在 _start 函数的最后，跳转到 libmain。这个函数定义在 user/lib/libos.c 中。在这个函数中，我们通过一个函数 syscall_getenvid 获取了当前进程的 envid。此函数是一个系统调用，我们留到之后讲解。之后使用宏 ENVX 根据 envid 获取到该 envid 对应的进程控制块相对于进程控制块数组的索引值，并取得当前进程的进程控制块。

```
void libmain(int argc, char **argv) {
    // set env to point at our env structure in envs[].
    env = &envs[ENVX(syscall_getenvid())];
```

你可能没找到 env 和 envs 的定义，它们都在 user/include/lib.h 中。你可能还记得 Lab3 中我们将页控制块数组和进程控制块数组映射到用户虚拟地址空间的某一位置，这里 envs 就是映射到的用户虚拟地址，同理还有 pages。注意不要与内核空间中的 envs 和 pages 搞混。虽然它们确实表示同一物理地址的相同数据，但是用户程序无法访问内核地址空间中的 envs 和 pages，如果访问会产生异常。

另外 env 用于表示本用户程序的进程控制块。我们可以通过访问 env 获取当前进程的信息。

```
#define envs ((volatile struct Env *)UENVS)
#define pages ((volatile struct Page *)UPAGES)

extern volatile struct Env *env;
```

之后，libmain 函数便调用了 main 函数

```
    // call user main routine
    main(argc, argv);
```

最后，当 main 函数返回后，我们调用 exit 函数结束进程

```
    // exit gracefully
    exit();
}
```

exit 函数同样定义在 user/lib/libos.c 中。

```
void exit(void) {
    // After fs is ready (lab5), all our open files should be closed before dying.
#if !defined(LAB) || LAB >= 5
    close_all();
#endif

    syscall_env_destroy(0);
    user_panic("unreachable code");
}
```

我们不考虑 Lab5 之后才用到的内容。exit 函数只调用了一个 syscall_env_destroy，该函数也是一个系统调用，用于销毁进程，释放进程资源。这里传入了一个参数 asid = 0，表示销毁的是当前进程。在之后还会多次出现 asid = 0 表示当前进程（调用函数的进程）的用法。

值得注意，在 syscall_env_destroy 之后还多出一条语句 user_panic。此函数类似于 panic，不过用于表示用户程序出现了难以恢复的错误。在调用该函数打印错误信息后，就会结束该进程。

```
// user/include/lib.h
void _user_panic(const char *, int, const char *, ...) __attribute__((noreturn));
#define user_panic(...) _user_panic(__FILE__, __LINE__, __VA_ARGS__)

// user/lib/debug.c
void _user_panic(const char *file, int line, const char *fmt, ...) {
    debugf("panic at %s:%d: ", file, line);
    va_list ap;
    va_start(ap, fmt);
    vdebugf(fmt, ap);
    va_end(ap);
    debugf("\n");
    exit();
}
```

在 exit 函数中出现的 user_panic 很明显的指出 syscall_env_destroy 是一个不会返回的函数。

这样，用户程序的入口的故事我们就讲完了。你现在应该就可以理解 Lab3 中使用的用户程序位于 bare 路径下的原因了。因为他们没有被 libmain 包裹，是赤裸裸地暴露在外运行的。

之后还是让我们看一下 user/tltest.c 的程序吧

```
int main() {
    debugf("Smashing some kernel codes...\n"
           "If your implementation is correct, you may see unknown exception
here:\n");
    *(int *)KERNBASE = 0;
    debugf("My mission completed!\n");
    return 0;
}
```

这里用户程序试图向内核空间写入数据，于是就会产生异常。运行的结果如下。`do_reserved` 函数用于处理除了定义过异常处理函数的其他异常。5 号异常表示地址错误。

```
Smashing some kernel codes...
If your implementation is correct, you may see unknown exception here:
...
panic at traps.c:24 (do_reserved): Unknown ExcCode 5
```

这里出现了一个新函数 `debugf`，该函数在用户程序的地位相当于内核中的 `printk`。它定义在 user/lib/debugf.c 中。为了避免这篇文章过程，不详细解释该函数，只是提一下 `debugf` 和相关函数的调用关系。这里有 `debugf -> vdebugf -> vprintfmt -> debug_output -> debug_flush -> syscall_print_cons`。最终，为了向屏幕输出字符，我们的用户程序还是使用了系统调用 `syscall_print_cons`。

```
static void debug_flush(struct debug_ctx *ctx) {
    if (ctx->pos == 0) {
        return;
    }
    int r;
    if ((r = syscall_print_cons(ctx->buf, ctx->pos)) != 0) {
        user_panic("syscall_print_cons: %d", r);
    }
    ctx->pos = 0;
}

static void debug_output(void *data, const char *s, size_t l) {
    struct debug_ctx *ctx = (struct debug_ctx *)data;

    while (ctx->pos + l > BUF_LEN) {
        size_t n = BUF_LEN - ctx->pos;
        memcpy(ctx->buf + ctx->pos, s, n);
        s += n;
        l -= n;
        ctx->pos = BUF_LEN;
        debug_flush(ctx);
    }
    memcpy(ctx->buf + ctx->pos, s, l);
    ctx->pos += l;
```

```
}

static void vdebugf(const char *fmt, va_list ap) {
    struct debug_ctx ctx;
    ctx.pos = 0;
    vprintfmt(debug_output, &ctx, fmt, ap);
    debug_flush(&ctx);
}

void debugf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    vdebugf(fmt, ap);
    va_end(ap);
}
```

你可能会想，系统调用到底有什么用？为什么用户程序的许多操作都需要由内核经手？在我看来，系统调用，乃至其他操作系统的许多其他功能，目的都是安全性。这体现在两个方面。一是系统的安全，保证计算机不会被非法篡改；二是操作的安全，保证用户的大部分操作都不会破坏计算机系统本身。为了做到这两点，我们就需要将一些更加重要的数据结构和算法隐藏在内核中，与用户隔离。从而避免有意或无意的操作对数据造成无法挽回的破坏。

（2）系统调用的实现

在上一小节中我们见识了一些系统调用函数，在 MOS 的提供给用户程序的库中，所有的系统调用均为 syscall_* 的形式。所以，这些函数就是内核提供给用户程序的接口吗？

答案是否定的。所有的系统调用都定义在 user/lib/syscall_lib.c 中。我们取之前看到过的系统调用函数为例

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

u_int syscall_getenvid(void) {
    return msyscall(SYS_getenvid);
}

int syscall_env_destroy(u_int envid) {
    return msyscall(SYS_env_destroy, envid);
}
```

可以看到，这些函数（其实是所有的系统调用函数）都调用了 `msyscall` 一个函数。为其传入了不同数量、类型的参数。你可能已经想到了，这是一个拥有变长参数的函数。我们可以在 user/include/lib.h 中找到该函数的声明。

```
/// syscalls
extern int msyscall(int, ...);
```

你可能会想，“原来内核提供给用户程序的接口其实只有 msyscall 函数一个。第一个参数表示系统调用的类型，剩下的参数根据系统调用不同也有所区别。内核只对用户程序暴露这一个函数，最大程度限制了用户程序对内核的访问。” 事实是这样吗?

其实也不是。还是让我们看一看 msyscall 的定义吧。

```
LEAF(msyscall)
    // Just use 'syscall' instruction and return.

    /* Exercise 4.1: Your code here. */
    syscall
    jr  ra
END(msyscall)
```

该函数只有两条指令，第二条指令 jr ra 只是从该函数返回。真正关键的只有第一条 syscall。这条指令用于让程序自行产生一个异常，该异常被称为系统异常。这样进行异常处理，才进入了内核态。实际上经过了几个实验，应该明白一点，**内核态提供给用户程序的接口，只有异常。**

插一句嘴，虽然 msyscall 是一个拥有可变参数的函数，可我们并没有使用可变参数宏来进行处理。由此可见，所谓的可变参数，不过是编译器的特殊处理罢了。

接下来我们就要处理系统异常。还记得我们在 Lab3 中略过不讲的两个异常处理函数吗？这两个函数将在本次实验中起到大用处。这里我们先看 do_syscall

```
#if !defined(LAB) || LAB >= 4
BUILD_HANDLER mod do_tlb_mod
BUILD_HANDLER sys do_syscall
#endif
```

回忆一下 Lab3 中从进入异常到进入特定异常的处理函数中间的过程。在这中间我们使用 SAVE_ALL 宏保存了发生异常时的现场。对于系统异常来说，需要着重强调，我们保存了调用 msyscall 函数时的参数信息。对于这一点，请回忆 mips 函数调用的相关知识。

do_syscall 函数位于 kern/syscall_all.c 中。该函数用于实现系统调用的分发和运行。

首先，我们取出用户程序 trap frame 中的 a0 寄存器的值。该值即调用 msyscall 函数时的第一个参数，确实用于表示系统调用的类型。之后我们判断 sysno 是否处于范围内，如果不是则 “返回”。需要注意这里返回值的设定方法。我们为 trap frame 中的 v0 寄存器赋值。在 mips 函数调用中，该寄存器存储返回值。

```
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;
    }
```

之后，我们将 epc 寄存器的地址加 4。如果还记得 Lab3，应该知道 epc 寄存器存储了发生异常的指令地址。当完成异常处理，调用 ret_from_exception 从异常中返回时，也是回到 epc 指向的指令再次执行。这里将 epc 寄存器的地址加 4，意味着从异常返回后并不重新执行 syscall 指令，而是其下一条指令。

```
    /* 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;
```

然后，我们根据 sysno 取得对应的系统调用函数。

```
    /* Step 2: Use 'sysno' to get 'func' from 'syscall_table'. */
    /* Exercise 4.2: Your code here. (2/4) */
    func = syscall_table[sysno];
```

其中 syscall_table 中存储了所有的系统调用函数

```
void *syscall_table[MAX_SYSNO] = {
    [SYS_putchar] = sys_putchar,
    [SYS_print_cons] = sys_print_cons,
    [SYS_getenvid] = sys_getenvid,
    [SYS_yield] = sys_yield,
    [SYS_env_destroy] = sys_env_destroy,
    [SYS_set_tlb_mod_entry] = sys_set_tlb_mod_entry,
    [SYS_mem_alloc] = sys_mem_alloc,
    [SYS_mem_map] = sys_mem_map,
    [SYS_mem_unmap] = sys_mem_unmap,
    [SYS_exofork] = sys_exofork,
    [SYS_set_env_status] = sys_set_env_status,
    [SYS_set_trapframe] = sys_set_trapframe,
    [SYS_panic] = sys_panic,
    [SYS_ipc_try_send] = sys_ipc_try_send,
    [SYS_ipc_recv] = sys_ipc_recv,
    [SYS_cgetc] = sys_cgetc,
    [SYS_write_dev] = sys_write_dev,
    [SYS_read_dev] = sys_read_dev,
};
```

最后，我们从用户程序的 trap frame 中取出调用 msyscall 时传入的参数。按照 mips 函数调用规范，函数调用的前四个参数存储在 a0-a3 寄存器中。更多的参数被存储在 sp 寄存器的对应地址中。因为我们的系统调用最多有 5 个参数，因此需要取得 arg1 到 arg5 的值。最后我们调用根据 sysno 取得的系统调用函数，调用该函数，将其返回值存储在 v0 寄存器中。

```
    /* Step 3: First 3 args are stored at $a1, $a2, $a3. */
    u_int arg1 = tf->regs[5];
    u_int arg2 = tf->regs[6];
    u_int arg3 = tf->regs[7];

    /* 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]+16);
    arg5 = *(u_int *)(tf->regs[29]+20);

    /* 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);
}
```

实际上我们不难看出，在 do_syscall 函数中我们修改了用户程序 trap frame 的值，其目的就是模拟函数调用的效果。这样在恢复现场，用户程序继续运行时，就会感觉和平常的函数调用没有不同之处。异常的处理过程，就仿佛变成了内核暴露给用户程序的接口了。可是我们还是要明确这一点，系统调用的本质，就是异常处理。

（3）补充 env_destroy

Lab3 的笔记中没有讲解 env_destroy，主要是因为我们确实没有用到该函数。现在我们将其封装成系统调用，作为 main 函数之后的资源回收函数，就是时候考虑该函数了。

我们使用的系统调用为 sys_env_destroy，可以看出只是封装了一下 env_destroy。

```
int sys_env_destroy(u_int envid) {
    struct Env *e;
    try(envid2env(envid, &e, 1));

    printk("[%08x] destroying %08x\n", curenv->env_id, e->env_id);
    env_destroy(e);
    return 0;
}
```

那么 env_destroy 呢？该函数主要作用是调用了 env_free 函数。另外对于是当前函数被销毁的情况，我们就需要进行进程调度。

```
void env_destroy(struct Env *e) {
    /* Hint: free e. */
    env_free(e);

    /* Hint: schedule to run a new environment. */
    if (curenv == e) {
        curenv = NULL;
        printk("i am killed ... \n");
        schedule(1);
    }
}
```

对于 env_free 函数。首先遍历所有页表项，使用 page_remove 删除虚拟地址到物理页的映射；另外使用 page_decref 释放页表和页目录本身。其中还使用 asid_free 释放了 asid。对页表项进行修改后，使用了 tlb_invalidate 将对应的项无效化。

```
void env_free(struct Env *e) {
    Pte *pt;
    u_int pdeno, pteno, pa;

    /* Hint: Note the environment's demise.*/
    printk("[%08x] free env %08x\n", curenv ? curenv->env_id : 0, e->env_id);

    /* Hint: Flush all mapped pages in the user portion of the address space */
    for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {
        /* Hint: only look at mapped page tables. */
        if (!(e->env_pgdir[pdeno] & PTE_V)) {
            continue;
        }
        /* Hint: find the pa and va of the page table. */
        pa = PTE_ADDR(e->env_pgdir[pdeno]);
        pt = (Pte *)KADDR(pa);
        /* Hint: Unmap all PTEs in this page table. */
        for (pteno = 0; pteno <= PTX(~0); pteno++) {
            if (pt[pteno] & PTE_V) {
                page_remove(e->env_pgdir, e->env_asid,
                        (pdeno << PDSHIFT) | (pteno << PGSHIFT));
            }
        }
        /* Hint: free the page table itself. */
        e->env_pgdir[pdeno] = 0;
        page_decref(pa2page(pa));
        /* Hint: invalidate page table in TLB */
        tlb_invalidate(e->env_asid, UVPT + (pdeno << PGSHIFT));
    }
    /* Hint: free the page directory. */
    page_decref(pa2page(PADDR(e->env_pgdir)));
    /* Hint: free the ASID */
    asid_free(e->env_asid);
    /* Hint: invalidate page directory in TLB */
```

```
    tlb_invalidate(e->env_asid, UVPT + (PDX(UVPT) << PGSHIFT));
```

最后修改进程控制块的状态为 ENV_FREE，将该控制块从调度队列中删除，重新放回空闲列表中。

```
    /* Hint: return the environment to the free list. */
    e->env_status = ENV_FREE;
    LIST_INSERT_HEAD((&env_free_list), (e), env_link);
    TAILQ_REMOVE(&env_sched_list, (e), env_sched_link);
}
```

二、fork 的实现

（1）用户态异常处理的实现

fork 是创建进程的基本方法。某一进程调用该函数后，就会创建一个该进程的复制作为调用进程的子进程。子进程也会从 fork 后的时刻开始执行。但会具有和父进程不同的返回值。父进程的返回值为子进程的 进程标识符 envid，而子进程的返回值为 0。当然之后我们就会知道，envid = 0 可以表示当前进程。这样的话我们也可以理解成不管在父进程还是子进程，返回值都为指示子进程的 envid。

上面的内容可能有些难以理解，指导书里给出了一段样例代码用于帮助理解，这里也贴出来

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int var = 1;
    long pid;
    printf("Before fork, var = %d.\n", var);
    pid = fork();
    printf("After fork, var = %d.\n", var);
    if (pid == 0) {
        var = 2;
        sleep(3);
        printf("child got %ld, var = %d", pid, var);
    } else {
        sleep(2);
        printf("parent got %ld, var = %d", pid, var);
    }
    printf(", pid: %ld\n", (long) getpid());
    return 0;
}
```

我们还是查看一下 fork 函数的代码吧，它位于 user/lib/fork.c 中。fork 本身不是系统调用，但该函数使用了许多系统调用来完成子进程的创建。

首先我们就遇到了一个系统调用 env_user_tlb_mod_entry，这个系统调用的作用是设置一个 TLB Mod 异常的处理函数。TLB Mod 异常即页写入（Modify）异常，会在程序试图写入不可写入（对应页表项无 PTE_D）的物理页面时产生。

```
int fork(void) {
    u_int child;
    u_int i;
    extern volatile struct Env *env;

    /* Step 1: Set our TLB Mod user exception entry to 'cow_entry' if not done
yet. */
    if (env->env_user_tlb_mod_entry != (u_int)cow_entry) {
        try(syscall_set_tlb_mod_entry(0, cow_entry));
    }
```

该系统调用对应的函数为 sys_set_tlb_mod_entry，这个函数很简单，只是设置了进程控制块的 env_user_tlb_mod_entry 参数。

```
int sys_set_tlb_mod_entry(u_int envid, u_int func) {
    struct Env *env;

    /* Step 1: Convert the envid to its corresponding 'struct Env *' using
'envid2env'. */
    /* Exercise 4.12: Your code here. (1/2) */
    try(envid2env(envid, &env, 1));

    /* Step 2: Set its 'env_user_tlb_mod_entry' to 'func'. */
    /* Exercise 4.12: Your code here. (2/2) */
    env->env_user_tlb_mod_entry = func;

    return 0;
}
```

你可能会想，为什么要在用户程序中设置异常处理函数呢？这不是应该交由内核处理吗？设置了之后又要如何使用？我们需要查看一下内核中 TLB Mod 异常的处理函数 do_tlb_mod，位于 kern/tlbex.c 中。

在该函数中，我们首先将 sp 寄存器设置到 UXSTACKTOP 的位置。UXSTACKTOP 和 KSTACKTOP 类似，都是在处理异常时使用的调用栈，因为我们要将异常处理交由用户态执行，因此需要在用户的地址空间中分配。另外这里添加判断语句是考虑到在异常处理的过程中再次出现异常的情况，我们不希望之前的异常处理过程的信息丢失，而希望在处理了后一个异常后再次继续处理前一个异常，于是对于 sp 已经在异常栈中的情况，就不再从异常栈顶开始。

```
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;
    }
```

随后我们在用户异常栈底分配一块空间，用于存储 trap frame。这里和在 `KSTACKTOP` 底使用 `SAVE_ALL` 类似。

```
    tf->regs[29] -= sizeof(struct Trapframe);
    *(struct Trapframe *)tf->regs[29] = tmp_tf;
```

最后我们从当前进程的进程控制块的 `env_user_tlb_mod_entry` 取出由 `syscall_set_tlb_mod_entry` 设定的当前进程的 TLB Mod 异常处理函数，“调用”该处理函数。当然，因为处于异常处理过程中，所以这里我们采用和一般系统调用类似的处理方法。首先我们设定 `a0` 寄存器的值为 trap frame 所在的地址，接着 `sp` 寄存器自减，留出第一个参数的空间。最后设定 `epc` 寄存器的值为用户的 TLB Mod 函数的地址，使得恢复现场后跳转到该函数的位置继续执行。

```
    if (curenv->env_user_tlb_mod_entry) {
        tf->regs[4] = tf->regs[29];
        tf->regs[29] -= sizeof(tf->regs[4]);
        // Hint: Set 'cp0_epc' in the context 'tf' to 'curenv-
>env_user_tlb_mod_entry'.
        /* Exercise 4.11: Your code here. */
        tf->cp0_epc = curenv->env_user_tlb_mod_entry;
    } else {
        panic("TLB Mod but no user handler registered");
    }
}
```

应当注意，作为用户态异常处理函数的参数的 `TrapFrame * tf` 和当前进程的 `tf` 是不同的。前者依旧是产生 TLB Mod 异常时的现场；而后者则经过了修改，以便在从异常处理返回时跳转到用户态异常处理函数而非产生异常的位置。在用户态异常处理函数完成异常处理后，我们会通过前者返回产生异常的位置。

最后我们还需要注意 `sys_set_tlb_mod_entry` 中使用了 `envid2env` 函数来根据进程标识符获取对应的进程。这里出现了 envid = 0 时直接返回当前进程的实现。正是因为以 envid 作为参数的函数都会使用本函数获取对应的进程控制块，所以才会有 envid = 0 表示当前进程的说法。另外，因为 envid = 0 有特殊含义，所以生成进程标识符的函数 `mkenvid` 永远不可能生成为 0 的 envid。对于 envid 不为 0 的情况，我们通过 `ENVX` 宏获取 envid 对应的进程控制块相对于进程控制块数组的索引。`mkenvid` 的算法中，envid 的低十位就是进程控制块的索引，因此我们才可以通过 envid 获取对应的进程控制块。

```
int envid2env(u_int envid, struct Env **penv, int checkperm) {
    struct Env *e;

    /* Step 1: Assign value to 'e' using 'envid'. */
    /* Hint:
     *   If envid is zero, set 'penv' to 'curenv'.
     *   You may want to use 'ENVX'.
     */
```

```
    /* Exercise 4.3: Your code here. (1/2) */
    if (envid == 0) {
        *penv = curenv;
        return 0;
    } else {
        e = envs + ENVX(envid);
    }
```

在 envid2env 的后半部分，我们进行了一系列进程控制块的有效性检查。注意这里的 e->env_id != envid，按说我们是通过 envid 获取的进程控制块，怎么可能进程控制块的 env_id 不相同呢？这实际上考虑了这样一种情况，某一进程完成运行，资源被回收，这时其对应的进程控制块会插入回 env_free_list 中。当我们需要再次创建内存时，就可能重新取得该进程控制块，并为其赋予不同的 envid。这时，已销毁进程的 envid 和新创建进程的 envid 都能通过 ENVX 宏取得相同的值，对应了同一个进程控制块。可是已销毁进程的 envid 却不应当再次出现，e->env_id != envid 就处理了 envid 属于已销毁进程的情况。

```
    if (e->env_status == ENV_FREE || e->env_id != envid) {
        return -E_BAD_ENV;
    }
```

对于设置了 checkperm 的情况，我们还需要额外检查传入的 envid 是否与当前进程具有直接亲缘关系。由此也可以得知，对于一些操作，只有父进程对子进程具有权限。

```
    /* Step 2: Check when 'checkperm' is non-zero. */
    /* Hints:
     *   Check whether the calling env has sufficient permissions to manipulate
the
     *   specified env, i.e. 'e' is either 'curenv' or its immediate child.
     *   If violated, return '-E_BAD_ENV'.
     */
    /* Exercise 4.3: Your code here. (2/2) */
    if (checkperm && e->env_id != curenv->env_id  &&e->env_parent_id != curenv-
>env_id) {
        return -E_BAD_ENV;
    }
```

最后返回取得的进程控制块。

```
    /* Step 3: Assign 'e' to '*penv'. */
    *penv = e;
    return 0;
}
```

（2）写时复制技术（Copy on Write, COW）

所以说，我们为什么需要设置 TLB Mod 的异常处理函数呢？在这里我们的目的是实现进程的写时复制。我们知道，fork 会根据复制调用进程来创建一个新进程。可如果每创建一个新的进程就要在内存中复制一份相同的数据，开销就太大了。所以我们可以在创建子进程时只是让子进程映射到和父进程相同的物理页。这样如果父进程和子进程只是读取其中的内容，就可以共享同一片物理空间。那么如果有进程想要修改该空间内的数据要怎么办？这时我们才需要将这块物理空间复制一份，让想要修改的进程只修改属于自己的数据。

在 MOS 中，对于可以共享的页，我们会去掉其写入（PTE_D）权限，为其赋予写时复制标记（PTE_COW），这样当共享该页面的某一个进程尝试修改该页的数据时，因为不具有 PTE_D 权限，就会产生 TLB Mod 异常。转到异常处理函数。根据 fork 中的语句可知，我们的异常处理函数为 cow_entry，同样位于 user/lib/fork.c 中。

首先，我们取得发生异常时尝试写入的虚拟地址位置，使用 VPN 宏获取该虚拟地址对应的页表项索引。之后使用 vpt 获取页表项的内容，通过位操作取出该页表项的权限。发生 TLB Mod 的必然是被共享的页面，因此如果不具有 PTE_COW 则产生一个 user_panic。

```
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
    u_int va = tf->cp0_badvaddr;
    u_int perm;

    /* Step 1: Find the 'perm' in which the faulting address 'va' is mapped. */
    /* Hint: Use 'vpt' and 'VPN' to find the page table entry. If the 'perm'
doesn't have
     * 'PTE_COW', launch a 'user_panic'. */
    /* Exercise 4.13: Your code here. (1/6) */
    perm = vpt[VPN(va)] & 0xfff;
    if (!(perm & PTE_COW)) {
        user_panic("perm doesn't have PTE_COW");
    }
```

值得注意的是 vpt。从表现上来看，vpt 是一个 Pte 类型的数组，其中按虚拟地址的顺序存储了所有的页表项。从本质上看，vpt 是用户空间中的地址，并且正是页表自映射时设置的基地址。这样我们就不难理解为什么可以通过这取得所有列表项了。类似的还有 vpd，是存储页目录项的数组。

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

如果你忘了页表自映射，可以回头看看 Lab2 和 Lab3。在 MOS 中实现页表自映射的语句位于 kern/env.c 的 env_setup_vm 中。

```
    /* Step 3: Map its own page table at 'UVPT' with readonly permission.
     * As a result, user programs can read its page table through 'UVPT' */
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
```

让我们回到 cow_entry，接下来重新设置页的权限，再去掉 PTE_COW，加上 PTE_D。

```
    /* Step 2: Remove 'PTE_COW' from the 'perm', and add 'PTE_D' to it. */
    /* Exercise 4.13: Your code here. (2/6) */
    perm = (perm & ~PTE_COW) | PTE_D;
```

接着再申请一页新的物理页，该物理页对应的虚拟地址为 UCOW。因为处于用户程序中，所以不能使用 page_alloc，而需要使用系统调用。需要注意，因为 asid = 0，所以新物理页是属于当前调用进程的。

```
    /* Step 3: Allocate a new page at 'UCOW'. */
    /* Exercise 4.13: Your code here. (3/6) */
    syscall_mem_alloc(0, (void *)UCOW, perm);
```

还有一点值得思考，这里我们申请的物理页不在发生异常的 va 的位置，而是特定的 UCOW。之后又通过一系列映射操作将该物理页映射到 va。为什么要这样做呢？很简单，如果我们一开始就映射到 va，那么如果你还记得 page_insert 内容的话，就会知道这样会使原本的映射丢失，我们就不能访问原本 va 映射到的物理页的内容了。这样的话我们又要如何复制呢？所以要先映射到一个完全无关的地址 UCOW。该地址之上 BY2PG 大小的空间专门用于写时复制时申请新的物理页。

syscall_mem_alloc 系统调用较为简单，只是单纯的申请物理页，并插入到对应进程中。

```
int sys_mem_alloc(u_int envid, u_int va, u_int perm) {
    struct Env *env;
    struct Page *pp;

    /* Step 1: Check if 'va' is a legal user virtual address using
'is_illegal_va'. */
    /* Exercise 4.4: Your code here. (1/3) */
    if (is_illegal_va(va)) {
        return -E_INVAL;
    }

    /* Step 2: Convert the envid to its corresponding 'struct Env *' using
'envid2env'. */
    /* Hint: **Always** validate the permission in syscalls! */
    /* Exercise 4.4: Your code here. (2/3) */
    try(envid2env(envid, &env, 1));

    /* Step 3: Allocate a physical page using 'page_alloc'. */
    /* Exercise 4.4: Your code here. (3/3) */
    try(page_alloc(&pp));

    /* Step 4: Map the :allocated page at 'va' with permission 'perm' using
'page_insert'. */
    return page_insert(env->env_pgdir, env->env_asid, pp, va, perm);
}
```

`is_illegal_va` 函数用于判断虚拟地址是否位于用户空间中

```
static inline int is_illegal_va(u_long va) {
    return va < UTEMP || va >= UTOP;
}
```

回到 `cow_entry`，接下来我们将 `va` 所在物理页的内容复制到新申请的页面中。注意 `va` 可能只是随意的一个虚拟地址，不一定以 `BY2PG` 对齐，因此还需要使用 `ROUNDDOWN` 将其对齐。

```
    /* Step 4: Copy the content of the faulting page at 'va' to 'UCOW'. */
    /* Hint: 'va' may not be aligned to a page! */
    /* Exercise 4.13: Your code here. (4/6) */
    memcpy((void *)UCOW, (void *)ROUNDDOWN(va, BY2PG), BY2PG);
```

现在我们只需要取消 `va` 到原物理页的映射，将 `va` 映射到新申请的物理页即可。因此首先，使用系统调用 `syscall_mem_map` 将当前进程的 `UCOW` 所在的物理页，作为当前进程的 `va` 地址所映射的物理页，并设定其权限即可。

```
    // Step 5: Map the page at 'UCOW' to 'va' with the new 'perm'.
    /* Exercise 4.13: Your code here. (5/6) */
    syscall_mem_map(0, (void *)UCOW, 0, (void *)va, perm);
```

该系统调用同样是一个较为简单的函数。首先判断虚拟地址是否位于用户空间

```
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;

    /* Step 1: Check if 'srcva' and 'dstva' are legal user virtual addresses using
     * 'is_illegal_va'. */
    /* Exercise 4.5: Your code here. (1/4) */
    if (is_illegal_va(srcva) || is_illegal_va(dstva)) {
        return -E_INVAL;
    }
```

之后取得 `srcid`，`dstid` 进程标识符所对应的进程控制块

```
    /* Step 2: Convert the 'srcid' to its corresponding 'struct Env *' using
'envid2env'. */
    /* Exercise 4.5: Your code here. (2/4) */
```

```
    try(envid2env(srcid, &srcenv, 1));

    /* Step 3: Convert the 'dstid' to its corresponding 'struct Env *' using
'envid2env'. */
    /* Exercise 4.5: Your code here. (3/4) */
    try(envid2env(dstid, &dstenv, 1));
```

从源进程的页表系统中查找 srcva 对应的物理页

```
    /* Step 4: Find the physical page mapped at 'srcva' in the address space of
'srcid'. */
    /* Return -E_INVAL if 'srcva' is not mapped. */
    /* Exercise 4.5: Your code here. (4/4) */
    if ((pp = page_lookup(srcenv->env_pgdir, srcva, NULL)) == NULL) {
        return -E_INVAL;
    }
```

在目标进程的页表系统中创建 dstva 到该物理页的映射，同时设置权限

```
    /* Step 5: Map the physical page at 'dstva' in the address space of 'dstid'.
*/
    return page_insert(dstenv->env_pgdir, dstenv->env_asid, pp, dstva, perm);
}
```

这里因为 page_insert 会在要创建映射的虚拟地址已存在映射的情况下取消先前的映射，所以我们调用 syscall_mem_map 就同时完成了 “取消 va 到原物理页的映射” 和 “将 va 映射到新申请的物理页” 这两个操作。

UCOW 处还存在到新申请物理页的映射，我们需要取消该映射。调用 syscall_mem_unmap 实现 “取消该进程中 UCOW 与物理页的映射” 操作。

```
    // Step 6: Unmap the page at 'UCOW'.
    /* Exercise 4.13: Your code here. (6/6) */
    syscall_mem_unmap(0, (void *)UCOW);
```

syscall_mem_unmap 只是调用 page_remove 取消了映射而已。

```
int sys_mem_unmap(u_int envid, u_int va) {
    struct Env *e;

    /* Step 1: Check if 'va' is a legal user virtual address using
'is_illegal_va'. */
```

```
    /* Exercise 4.6: Your code here. (1/2) */
    if (is_illegal_va(va)) {
        return -E_INVAL;
    }

    /* Step 2: Convert the envid to its corresponding 'struct Env *' using
'envid2env'. */
    /* Exercise 4.6: Your code here. (2/2) */
    try(envid2env(envid, &e, 1));

    /* Step 3: Unmap the physical page at 'va' in the address space of 'envid'. */
    page_remove(e->env_pgdir, e->env_asid, va);
    return 0;
}
```

最后，我们需要在异常处理完成后恢复现场，可现在位于用户态，去哪里恢复现场呢？我们需要使用系统调用 `syscall_set_trapframe`。也因此 `cow_entry` 成为了一个不返回的函数 (`__attribute__((noreturn))`) 。

```
    // Step 7: Return to the faulting routine.
    int r = syscall_set_trapframe(0, tf);
    user_panic("syscall_set_trapframe returned %d", r);
}
```

`sys_set_trapframe` 将 trap frame 修改为传入的参数 `struct Trapframe *tf` 对应的 trap frame。这样当**从该系统调用返回**时，将返回设置的栈帧中 `epc` 的位置。对于 `cow_entry` 来说，意味着恢复到产生 TLB Mod 时的现场。

```
int sys_set_trapframe(u_int envid, struct Trapframe *tf) {
    if (is_illegal_va_range((u_long)tf, sizeof *tf)) {
        return -E_INVAL;
    }
    struct Env *env;
    try(envid2env(envid, &env, 1));
    if (env == curenv) {
        *((struct Trapframe *)KSTACKTOP - 1) = *tf;
        // return `tf->regs[2]` instead of 0, because return value overrides
regs[2] on
        // current trapframe.
        return tf->regs[2];
    } else {
        env->env_tf = *tf;
        return 0;
    }
}
```

这样，写时复制的主要流程就结束了。

（3）一次调用、两次返回

我们继续看 fork 函数。接下来的步骤是 fork 的关键。我们使用 syscall_exofork 系统调用。该系统调用的作用是复制父进程的信息，创建一个子进程。该系统调用实现了一次调用、两次返回，也是 fork 拥有此能力的原因。

```
    /* Step 2: Create a child env that's not ready to be scheduled. */
    // Hint: 'env' should always point to the current env itself, so we should fix
it to the
    // correct value.
    child = syscall_exofork();
```

在此调用之后的一条语句，我们就通过不同返回值实现了父子进程的不同流程，对于子进程来说，我们直接返回 0。这里我们还设置了 env 的值为当前进程（因为复制后，env 原本还指向父进程）。为了取得当前进程的 envid，我们又使用了系统调用 syscall_getenvid。再次强调，env 和 envs 是位于用户空间的。

```
    if (child == 0) {
        env = envs + ENVX(syscall_getenvid());
        return 0;
    }
```

接下来我们就分析一下 syscall_exofork。首先，为了创建新的进程，我们需要申请一个进程控制块。需要注意，这里 env_alloc 的 parent 参数为当前进程的 envid。

```
int sys_exofork(void) {
    struct Env *e;

    /* Step 1: Allocate a new env using 'env_alloc'. */
    /* Exercise 4.9: Your code here. (1/4) */
    try(env_alloc(&e, curenv->env_id));
```

接着我们复制父进程调用系统操作时的现场。

```
    /* Step 2: Copy the current Trapframe below 'KSTACKTOP' to the new env's
'env_tf'. */
    /* Exercise 4.9: Your code here. (2/4) */
    e->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
```

需要注意这里不能使用 curenv->env_tf。因为 curenv->env_tf 存储的是进程调度，切换为其他进程之前的 trap frame。而父进程调用 syscall_exofork 时保存的现场，并不一定等同于curenv->tf 中的信息。如果你还记得 Lab3，应该明白只有一处进行了 env_tf 的赋值。就是在 env_run 函数中。

```
    if (curenv) {
        curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
    }
```

最后 syscall_exofork 将子进程的返回值设置为 0，同时设置进程的状态和优先级，最后返回新创建进程的 envid。

```
    /* Step 3: Set the new env's 'env_tf.regs[2]' to 0 to indicate the return
value in child. */
    /* Exercise 4.9: Your code here. (3/4) */
    e->env_tf.regs[2] = 0;

    /* Step 4: Set up the new env's 'env_status' and 'env_pri'.  */
    /* Exercise 4.9: Your code here. (4/4) */
    e->env_status = ENV_NOT_RUNNABLE;
    e->env_pri = curenv->env_pri;

    return e->env_id;
}
```

到这里你应该能理解何为 “一次调用、两次返回” 了。一次调用指的是只有父进程调用了 syscall_exofork，两次返回分别是父进程调用 syscall_exofork 得到的返回值和被创建的子进程中设定了 v0 寄存器的值为 0 作为返回值。这样当子进程开始运行时，就会拥有一个和父进程不同的返回值。

（4）子进程运行前的设置

我们已经创建了子进程，但是子进程现在还没有加入调度队列，同时父子进程虽然共享了页表，但页表项还没有设置 PTE_COW 位。这些我们都要进行处理。

设置 PTE_COW 位需要通过 duppage 函数。如果当前页表项具有 PTE_D 权限（且不是共享页面 PTE_LIBRARY），则需要重新设置页表项的权限。duppage 会对每一个页表项进行操作，因此我们需要在 fork 中遍历所有的页表项。相比于在内核态中，在用户态中遍历页表项更为方便。

这里我们只遍历 USTACKTOP 之下的地址空间，因为其上的空间总是会被共享。在调用 duppage 之前，我们判断页目录项和页表项是否有效。如果不判断则会在 duppage 函数中发生异常（最终是由 page_lookup 产生的）。需要注意取页目录项的方法，vpd 是页目录项数组，i 相当于地址的高 20 位，我们需要取得地址的高 10 位作为页目录的索引，因此有 vpd[i >> 10]

```
    /* Step 3: Map all mapped pages below 'USTACKTOP' into the child's address
space. */
    // Hint: You should use 'duppage'.
    /* Exercise 4.15: Your code here. (1/2) */
    for (i = 0; i < VPN(USTACKTOP); i++) {
        if ((vpd[i >> 10] & PTE_V) && (vpt[i] & PTE_V)) {
            duppage(child, i);
        }
    }
```

接下来让我们看一下 duppage 函数。首先取得页表项对应的虚拟地址和权限。

```
static void duppage(u_int envid, u_int vpn) {
    int r;
    u_int addr;
    u_int perm;

    /* Step 1: Get the permission of the page. */
    /* Hint: Use 'vpt' to find the page table entry. */
    /* Exercise 4.10: Your code here. (1/2) */
    addr = vpn << PGSHIFT;
    perm = vpt[vpn] & 0xfff;
```

接着，对所有有效的页，我们都需要通过系统调用 syscall_mem_map 实现父进程与子进程页面的共享。特别的，对于可写的，且不是共享的页，我们还需要更新页表项的权限。在用户态，我们不能直接修改页表项，因此需要通过系统调用来实现修改。我们同样使用的是 syscall_mem_map。虽然之前我们使用此系统调用来进行页的共享和复制，但由于该系统调用具有新的映射会覆盖旧的映射的特点，因此可以对本来就有的关系采取重新映射，只改变权限位的设置，就可以实现权限位的修改。

```
    /* Step 2: If the page is writable, and not shared with children, and not
marked as COW yet,
     * then map it as copy-on-write, both in the parent (0) and the child (envid).
*/
    /* Hint: The page should be first mapped to the child before remapped in the
parent. (Why?)
     */
    /* Exercise 4.10: Your code here. (2/2) */
    int flag = 0;
    if ((perm & PTE_D) && !(perm & PTE_LIBRARY)) {
        perm = (perm & ~ PTE_D) | PTE_COW;
        flag = 1;
    }

    syscall_mem_map(0, addr, envid, addr, perm);

    if (flag) {
        syscall_mem_map(0, addr, 0, addr, perm);
    }
}
```

这里需要注意，父进程将页映射到子进程应该先于对自己权限的修改。如果先修改自己的权限位，则该页表就不再可写，这样的话就会发生 TLB Mod 异常，而不能实现父进程将页映射到子进程。

之后在 fork 中，我们同样设置子进程的 TLB Mod 异常处理函数为 cow_entry。

```
    /* Step 4: Set up the child's tlb mod handler and set child's 'env_status' to
     * 'ENV_RUNNABLE'. */
    /* Hint:
     *   You may use 'syscall_set_tlb_mod_entry' and 'syscall_set_env_status'
     *   Child's TLB Mod user exception entry should handle COW, so set it to
'cow_entry'
     */
    /* Exercise 4.15: Your code here. (2/2) */
    try(syscall_set_tlb_mod_entry(child, cow_entry));
```

最后，我们调用 `syscall_set_env_status` 将子进程状态设定为 `RUNNABLE` 并将其加入调度队列中。返回子进程的 envid。作为父进程 `fork` 的返回值。

```
    try(syscall_set_env_status(child, ENV_RUNNABLE));
    return child;
}
```

`syscall_set_env_status` 较为简单，只是根据设定的状态将进程加入或移除调度队列而已。

```
int sys_set_env_status(u_int envid, u_int status) {
    struct Env *env;
    /* Step 1: Check if 'status' is valid. */
    /* Exercise 4.14: Your code here. (1/3) */
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE) {
        return -E_INVAL;
    }

    /* Step 2: Convert the envid to its corresponding 'struct Env *' using
'envid2env'. */
    /* Exercise 4.14: Your code here. (2/3) */
    try(envid2env(envid, &env, 1));

    /* Step 4: Update 'env_sched_list' if the 'env_status' of 'env' is being
changed. */
    /* Exercise 4.14: Your code here. (3/3) */
    if (env->env_status != ENV_NOT_RUNNABLE && status == ENV_NOT_RUNNABLE) {
        TAILQ_REMOVE(&env_sched_list, env, env_sched_link);
    }
    else if (env->env_status != ENV_RUNNABLE && status == ENV_RUNNABLE) {
        TAILQ_INSERT_TAIL(&env_sched_list, env, env_sched_link);
    }

    /* Step 5: Set the 'env_status' of 'env'. */
    env->env_status = status;
    return 0;
}
```

这样，我们就通过 `fork` 完成了子进程的创建。

三、进程间通信

（1）信息接收

在 Lab4 中，我们还需要实现进程间通信。这需要实现两个系统调用 `syscall_ipc_try_send` 和 `syscall_ipc_recv`。

调用 `syscall_ipc_recv` 后会阻塞当前进程，直到收到信息。

该调用的参数为要接收信息的虚拟地址。首先我们要检查虚拟地址是否处于用户空间。另外当 `dstva` 为 0 时表示不需要传输额外信息。

```
int sys_ipc_recv(u_int dstva) {
    /* Step 1: Check if 'dstva' is either zero or a legal address. */
    if (dstva != 0 && is_illegal_va(dstva)) {
        return -E_INVAL;
    }
```

接着我们设置进程控制块的字段，`env_ipc_recving` 表示进程是否正在接收信息；`env_ipc_dstva` 存储要接收信息的地址。

```

    /* Step 2: Set 'curenv->env_ipc_recving' to 1. */
    /* Exercise 4.8: Your code here. (1/8) */
    curenv->env_ipc_recving = 1;

    /* Step 3: Set the value of 'curenv->env_ipc_dstva'. */
    /* Exercise 4.8: Your code here. (2/8) */
    curenv->env_ipc_dstva = dstva;
```

接着我们要阻塞当前进程，将该进程从调度队列中移出

```

    /* Step 4: Set the status of 'curenv' to 'ENV_NOT_RUNNABLE' and remove it from
     * 'env_sched_list'. */
    /* Exercise 4.8: Your code here. (3/8) */
    curenv->env_status = ENV_NOT_RUNNABLE;
    TAILQ_REMOVE(&env_sched_list, curenv, env_sched_link);
```

最后我们将返回值设置为 0，调用 `schedule` 函数进行进程切换。

```

    /* Step 5: Give up the CPU and block until a message is received. */
    ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0;
    schedule(1);
}
```

你可能会想，`schedule` 函数不是没有返回的吗？那这里设置的返回值保存在哪里？另外如果没有返回，该系统调用又要如何返回到用户程序中？关键在于 `env_run` 函数中。在进程切换之前，会将 trap frame 存储到 `env->env_tf` 中。当重新轮到该进程运行的时候，就会从 `env_tf` 存储的位置恢复现场。这里也重新强调了 `sys_exofork` 中为什么不能使用 `env_tf`。

（2）信息发送

在 `sys_ipc_try_send` 中我们同样判断地址是否正确。并通过 envid 获取进程控制块。需要注意这里 `envid2env` 的 `checkperm` 参数为 0，而此前所有的参数值均为 1。这是因为进程通信不一定只在父子进程之间。

```
int sys_ipc_try_send(u_int envid, 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. */
    /* Exercise 4.8: Your code here. (4/8) */
    if (srcva != 0 && is_illegal_va(srcva)) {
        return -E_INVAL;
    }

    /* Step 2: Convert 'envid' to 'struct Env *e'. */
    /* This is the only syscall where the 'envid2env' should be used with
'checkperm' UNSET,
     * because the target env is not restricted to 'curenv''s children. */
    /* Exercise 4.8: Your code here. (5/8) */
    try(envid2env(envid, &e, 0));
```

然后我们检查 `env_ipc_recving`，这一字段在信息接收时设置。

```
    /* Step 3: Check if the target is waiting for a message. */
    /* Exercise 4.8: Your code here. (6/8) */
    if (!e->env_ipc_recving) {
        return -E_IPC_NOT_RECV;
    }
```

接下来我们传输一些信息，并将 `env_ipc_recving` 重新置 0，表示接收进程已经接收到信息。

```
    /* Step 4: Set the target's ipc fields. */
    e->env_ipc_value = value;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_perm = PTE_V | perm;
```

```
    e->env_ipc_recving = 0;
```

既然接收到了信息，那么我们就要取消接收进程的阻塞状态。

```
    /* Step 5: Set the target's status to 'ENV_RUNNABLE' again and insert it to
the tail of
     * 'env_sched_list'. */
    /* Exercise 4.8: Your code here. (7/8) */
    e->env_status = ENV_RUNNABLE;
    TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
```

最后，我们还需要将当前进程的一个页面共享到接收进程。只有这样，接收进程才能通过该页面获得发送进程发送的一些信息。

```
    /* Step 6: If 'srcva' is not zero, map the page at 'srcva' in 'curenv' to 'e-
>env_ipc_dstva'
     * in 'e'. */
    /* Return -E_INVAL if 'srcva' is not zero and not mapped in 'curenv'. */
    if (srcva != 0) {
        /* Exercise 4.8: Your code here. (8/8) */
        p = page_lookup(curenv->env_pgdir, srcva, NULL);
        if (p == NULL) {
            return -E_INVAL;
        }

        try(page_insert(e->env_pgdir, e->env_asid, p, e->env_ipc_dstva, perm));
    }
    return 0;
}
```