Exercise 4.1

填写user/lib/syscall_wrap.S 中的msyscall 函数，使得用户部分的系统调用机制可以正常工作

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

    /* Exercise 4.1: Your code here. */
    syscall    #陷入内核态
    jr      ra

END(msyscall)
```

只需执行自陷指令syscall 来陷入内核态并在处理结束后正常返回即可。请注意，syscall 指令是不允许在延迟槽中使用的。

Exercise 4.2

根据kern/syscall_all.c 中的提示，完成do_syscall 函数，使得内核部分的系统调用机制可以正常工作。

```
/* Overview:
 *   Call the function in 'syscall_table' indexed at 'sysno' with arguments from user context and
 * stack.
 *
 * Hint:
 *   Use sysno from $a0 to dispatch the syscall.
 *   The possible arguments are stored at $a1, $a2, $a3, [$sp + 16 bytes], [$sp + 20 bytes] in
 *   order.
 *   Number of arguments cannot exceed 5.
 */
void do_syscall(struct Trapframe *tf) {
    int (*func)(u_int, u_int, u_int, u_int, u_int);  //函数指针声明，指向一个接受5个无符号整数参数（u_int）并返回整型（int）的函数.
    int sysno = tf->regs[4];
    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;

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

    /* Step 3: First 3 args are stored in $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);      //取地址指向的内容
    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);
}
```

Exercise 4.3

实现kern/env.c 中的envid2env 函数。实现通过一个进程的id 获取该进程控制块的功能。提示：可以利用include/env.h中的宏函数ENVX()

```
/* Overview:
 *   Convert an existing 'envid' to an 'struct Env *'.
 *   If 'envid' is 0, set '*penv = curenv', otherwise set '*penv =
&envs[ENVX(envid)]'.
 *   In addition, if 'checkperm' is non-zero, the requested env must be either
'curenv' or its
 *   immediate child.
 *
 * Pre-Condition:
 *   'penv' points to a valid 'struct Env *'.
 *
 * Post-Condition:
 *   return 0 on success, and set '*penv' to the env.
 *   return -E_BAD_ENV on error (invalid 'envid' or 'checkperm' violated).
 */
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;  //直接返回当前进程块
        return 0;
    } else {
        e = &envs[ENVX(envid)];  //ENVX：env index； 对envid作处理
    }

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

    /* 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 != 0) {
        if(e->env_id != curenv->env_id && e->env_parent_id != curenv->env_id) {
            return -E_BAD_ENV;   //不满足'e' is either 'curenv' or its 子进程.
        }
    }

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

注：**当`envid`的值是0时，函数会返回指向当前进程控制块的指针**（通过形参`penv`返回）。当某些系统调用函数需要访问当前进程的进程控制块时，可以直接通过向`envid2env`传0来会获得指向当前进程控制块的指针，然后通过指针对进程控制块进行访问。

Exercise 4.4

实现kern/syscall_all.c 中的int sys_mem_alloc(u_int envid,u_int va, u_int perm) 函数。

sys_mem_alloc。这个函数的主要功能是分配内存，简单的说，用户程序可以通过这个系统调用给该程序所允许的虚拟内存空间显式地分配实际的物理内存。

```
/* Overview:
 *   Allocate a physical page and map 'va' to it with 'perm' in the address
space of 'envid'.
 *   If 'va' is already mapped, that original page is sliently unmapped.
 *   'envid2env' should be used with 'checkperm' set, like in most syscalls, to
ensure the target is
 * either the caller or its child.
 *
 * Post-Condition:
 *   Return 0 on success.
 *   Return -E_BAD_ENV: 'checkperm' of 'envid2env' fails for 'envid'.
 *   Return -E_INVAL:   'va' is illegal (should be checked using
'is_illegal_va').
```

```
 *   Return the original error: underlying calls fail (you can use 'try' macro).
 *
 * Hint:
 *   You may want to use the following functions:
 *   'envid2env', 'page_alloc', 'page_insert', 'try' (macro)
 */
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)) {   //虚拟地址是否合法
        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) */
    if(envid2env(envid, &env, 1)) {  // 1表示取的envid对应的env要是当前调用进程块或子
进程
        return -E_BAD_ENV;
    }

    /* 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);
}
```

Exercise 4.5

实现kern/syscall_all.c 中的int sys_mem_map(u_int srcid,u_int srcva, u_int dstid, u_int dstva, u_int perm)函数。

sys_mem_map。意义很直接：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。换句话说，此时两者共享一页物理内存。首先找到需要操作的两个进程，其次获取源进程的虚拟页面对应的实际物理页面，最后将该物理页面与目标进程的相应地址完成映射即可。

```
/* Overview:
 *   Find the physical page mapped at 'srcva' in the address space of env
'srcid', and map 'dstid''s
 *   'dstva' to it with 'perm'.
 *
 * Post-Condition:
 *   Return 0 on success.
```

```c
 *   Return -E_BAD_ENV: 'checkperm' of 'envid2env' fails for 'srcid' or 'dstid'.
 *   Return -E_INVAL: 'srcva' or 'dstva' is illegal, or 'srcva' is unmapped in 'srcid'.
 *   Return the original error: underlying calls fail.
 *
 * Hint:
 *   You may want to use the following functions:
 *   'envid2env', 'page_lookup', 'page_insert'
 */
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)) {  //虚拟地址是否合法
        return -E_INVAL;
    }

    /* Step 2: Convert the 'srcid' to its corresponding 'struct Env *' using 'envid2env'. */
    /* Exercise 4.5: Your code here. (2/4) */
    if(envid2env(srcid, &srcenv, 1)) {  //取得srcid对应进程块
        return -E_BAD_ENV;
    }

    /* Step 3: Convert the 'dstid' to its corresponding 'struct Env *' using 'envid2env'. */
    /* Exercise 4.5: Your code here. (3/4) */
    if(envid2env(dstid, &dstenv, 1)) {
        return -E_BAD_ENV;
    }

    /* 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) */
    Pte* pte = NULL;
    pp = page_lookup(srcenv->env_pgdir, srcva, &pte); //查询srcva映射的物理页框的页控制块（二级页表）
    if(pp == NULL) {
        return -E_INVAL;
    }

    /* 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); //该物理页面与目标进程的相应地址完成映射
}
```

Exercise 4.6

实现kern/syscall_all.c 中的int sys_mem_unmap(u_int envid, u_intva) 函数。

sys_mem_unmap，这个系统调用的功能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。

```c
/* Overview:
 *   Unmap the physical page mapped at 'va' in the address space of 'envid'.
 *   If no physical page is mapped there, this function silently succeeds.
 *
 * Post-Condition:
 *   Return 0 on success.
 *   Return -E_BAD_ENV: 'checkperm' of 'envid2env' fails for 'envid'.
 *   Return -E_INVAL:   'va' is illegal.
 *   Return the original error when underlying calls fail.
 */
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) */
    if(envid2env(envid, &e, 1)) {  //取得envid对应进程块e
        return -E_BAD_ENV;
    }

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

Exercise 4.7

实现kern/syscall_all.c 中的void sys_yield(void) 函数。

sys_yield。这个函数的功能是实现用户进程对CPU 的放弃，从而调度其他的进程。

```c
/* Overview:
 *   Give up remaining CPU time slice for 'curenv'.
 *
 * Post-Condition:
 *   Another env is scheduled.
 *
 * Hint:
 *   This function will never return.
 */
// void sys_yield(void);
void __attribute__((noreturn)) sys_yield(void) {
    // Hint: Just use 'schedule' with 'yield' set.
    /* Exercise 4.7: Your code here. */
    schedule(1);   //调度其他（1是指函数中yield参数不为0，即启动调度）
}
```

Exercise 4.8

实现kern/syscall_all.c 中的int sys_ipc_recv(u_int dstva) 函数和int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm) 函数。请注意在修改进程控制块的状态后，应同步维护调度队列。

```
/* Overview:
 *   Wait for a message (a value, together with a page if 'dstva' is not 0) from other envs.
 *   'curenv' is blocked until a message is sent.
 *
 * Post-Condition:
 *   Return 0 on success.
 *   Return -E_INVAL: 'dstva' is neither 0 nor a legal address.
 */
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;
    }

    /* Step 2: Set 'curenv->env_ipc_recving' to 1. */
    /* Exercise 4.8: Your code here. (1/8) */
    curenv->env_ipc_recving = 1;  //将自身的env_ipc_recving 设置为1，表明该进程准备接受发送方的消息。

    /* Step 3: Set the value of 'curenv->env_ipc_dstva'. */
    /* Exercise 4.8: Your code here. (2/8) */
    curenv->env_ipc_dstva = dstva;  //给env_ipc_dstva 赋值，表明自己要将接受到的页面与dstva完成映射。

    /* 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;  //阻塞当前进程，即把当前进程的状态置为不可运行
    TAILQ_REMOVE(&env_sched_list, curenv, env_sched_link);

```

```c
    /* Step 5: Give up the CPU and block until a message is received. */
    ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0;  //放弃CPU（调用相关函数重新进行调度），等待发送方将数据发送过来
    schedule(1);
}
```

```c
/* Overview:
 *   Try to send a 'value' (together with a page if 'srcva' is not 0) to the target env 'envid'.
 *
 * Post-Condition:
 *   Return 0 on success, and the target env is updated as follows:
 *   - 'env_ipc_recving' is set to 0 to block future sends.
 *   - 'env_ipc_from' is set to the sender's envid.
 *   - 'env_ipc_value' is set to the 'value'.
 *   - 'env_status' is set to 'ENV_RUNNABLE' again to recover from 'ipc_recv'.
 *   - if 'srcva' is not NULL, map 'env_ipc_dstva' to the same page mapped at 'srcva' in 'curenv'
 *     with 'perm'.
 *
 *   Return -E_IPC_NOT_RECV if the target has not been waiting for an IPC message with
 *   'sys_ipc_recv'.
 *   Return the original error when underlying calls fail.
 */
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)) {   //'srcva' 要么为0要么合法
        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) */
    envid2env(envid, &e, 0);  //根据envid 找到相应进程。 【唯一checkperm为0】

    /* Step 3: Check if the target is waiting for a message. */
    /* Exercise 4.8: Your code here. (6/8) */
    if(e->env_ipc_recving == 0) {  //指定进程是否为可接收状态。
        return -E_IPC_NOT_RECV;
    }

    /* Step 4: Set the target's ipc fields. */
    e->env_ipc_value = value;   //清除接收进程的接收状态，将相应数据填入进程控制块，传递物理页面的映射关系
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_perm = PTE_V | perm;  //PTE_V 有效位,若某页表项的有效位为1,则该页表项中高20位就是对应物理页号
    e->env_ipc_recving = 0;

    /* 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;   //修改进程控制块中的进程状态，使接受数据的进程可继续运行。
    TAILQ_INSERT_TAIL(&env_sched_list, (e), env_sched_link);

    /* 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) {     //srcva 不为0时，才建立两个进程的页面映射关系
        /* Exercise 4.8: Your code here. (8/8) */
        Pte* tmp_pte = NULL;
        p = page_lookup(curenv->env_pgdir, srcva, &tmp_pte);  //查找srcva(发射进程)对应物理页框
        if(p == NULL) {
            return -E_INVAL;
        }
        int ret = page_insert(e->env_pgdir, e->env_asid, p, e->env_ipc_dstva, e->env_ipc_perm); //让recv(接受进程)映射对应srcva的物理页框。[将两级页表结构中虚拟地址va映射到页控制块p对应的物理页面，并将页表项权限为设置为perm]
        if(ret != 0) {
            return ret;
        }
    }
    return 0;
}
```

注：当srcva 不为0 时，我们才建立两个进程的页面映射关系。

4_1 ↑ && 4_2 ↓

【地址分布】

```
/*
        4G -----------> +----------------------------+------------0x100000000
 o                      |       ...                  |  kseg2
 o      KSEG2    -----> +----------------------------+------------0xc000 0000
 o                      |          Devices           |  kseg1
 o      KSEG1    -----> +----------------------------+------------0xa000 0000
 o                      |      Invalid Memory        |   /|\
 o                      +----------------------------+----|-------Physical Memory Max
 o                      |       ...                  |  kseg0
 o      KSTACKTOP-----> +----------------------------+----|-------0x8040 0000-------end
 o                      |       Kernel Stack         |    | KSTKSIZE            /|\
 o                      +----------------------------+----|------              |
 o                      |       Kernel Text          |    |                 PDMAP
 o      KERNBASE -----> +----------------------------+----|-------0x8001 0000    |
 o                      |      Exception Entry       |   \|/                    \|/
 o      ULIM     -----> +----------------------------+------------0x8000 0000-------
 o                      |         User VPT           |     PDMAP                /|\
 o      UVPT     -----> +----------------------------+------------0x7fc0 0000    |
 o                      |           pages            |     PDMAP                 |
 o      UPAGES   -----> +----------------------------+------------0x7f80 0000    |
 o                      |           envs             |     PDMAP                 |
 o  UTOP,UENVS   -----> +----------------------------+------------0x7f40 0000    |
 o  UXSTACKTOP -/       |     user exception stack   |     BY2PG                 |
 o                      +----------------------------+------------0x7f3f f000    |
 o                      |                            |     BY2PG                 |
 o      USTACKTOP ----> +----------------------------+------------0x7f3f e000    |
 o                      |     normal user stack      |     BY2PG                 |
 o                      +----------------------------+------------0x7f3f d000    |
 a                      |                            |                           |
 a                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                           |
 a                      .                            .                           |
 a                      .                            .                         kuseg
 a                      .                            .                           |
```

```
 a                        |~~~~~~~~~~~~~~~~~~~~~~~~~~~~|                          |
 a                        |                            |                          |
 o       UTEXT    -----> +----------------------------+------------0x0040 0000    |
 o                        |      reserved for COW      |     BY2PG                |
 o       UCOW     -----> +----------------------------+------------0x003f f000    |
 o                        |   reversed for temporary   |     BY2PG                |
 o       UTEMP    -----> +----------------------------+------------0x003f e000    |
 o                        |       invalid memory       |                         \|/
 a     0 ------------>  +----------------------------+ -------------------------------
 o
*/
```

Exercise 4.9

请根据上述步骤以及代码中的注释提示，填写kern/syscall_all.c 中的sys_exofork 函数。

> 新进程的创建——sys_exofork (系统调用)

```c
/* Overview:
 *   Allocate a new env as a child of 'curenv'.
 *
 * Post-Condition:
 *   Returns the child's envid on success, and
 *   - The new env's 'env_tf' is copied from the kernel stack, except for $v0 set to 0 to indicate
 *     the return value in child.
 *   - The new env's 'env_status' is set to 'ENV_NOT_RUNNABLE'.
 *   - The new env's 'env_pri' is copied from 'curenv'.
 *   Returns the original error if underlying calls fail.
 *
 * Hint:
 *   This syscall works as an essential step in user-space 'fork' and 'spawn'.
 */
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));    //创建子进程 （后一个id是parent_id）

    /* Step 2: Copy the current Trapframe below 'KSTACKTOP' to the new env's 'env_tf'. */
```

```c
    /* Exercise 4.9: Your code here. (2/4) */
    e->env_tf = *((struct Trapframe*) KSTACKTOP - 1);  //复制一份当前进程的运行现场（进程上下文）Trapframe 到子进程的进程控制块中

    /* 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;         //子进程的$v0（返回值）修改为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;       //优先级复制

    return e->env_id;
}
```

Exercise 4.10

结合代码注释以及上述提示，填写user/lib/fork.c 中的duppage 函数。

> 父进程还需要将地址空间中需要与子进程共享的页面映射给子进程，这需要我们遍历父进程的大部分用户空间页，并使用将要实现的duppage 函数来完成这一过程。duppage 时，对于可以写入的页面的页表项，在父进程和子进程都需要加上PTE_COW 标志位，同时取消PTE_D标志位，以实现写时复制保护。

```c
/* Overview:
 *   Grant our child 'envid' access to the virtual page 'vpn' (with address 'vpn' * 'BY2PG') in our
 *   (current env's) address space.
 *   'PTE_COW' should be used to isolate the modifications on unshared memory from a parent and its
 *   children.
 *
 * Post-Condition:
 *   If the virtual page 'vpn' has 'PTE_D' and doesn't has 'PTE_LIBRARY', both our original virtual
 *   page and 'envid''s newly-mapped virtual page should be marked 'PTE_COW' and without 'PTE_D',
 *   while the other permission bits are kept.
 *
 *   If not, the newly-mapped virtual page in 'envid' should have the exact same permission as our
 *   original virtual page.
 *
 * Hint:
 *   - 'PTE_LIBRARY' indicates that the page should be shared among a parent and its children.
 *   - A page with 'PTE_LIBRARY' may have 'PTE_D' at the same time, you should handle it correctly.
 *   - You can pass '0' as an 'envid' in arguments of 'syscall_*' to indicate current env because
 *     kernel 'envid2env' converts '0' to 'curenv').
 *   - You should use 'syscall_mem_map', the user space wrapper around 'msyscall' to invoke
 *     'sys_mem_map' in kernel.
 */
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) */
  //perm = *(vpt + vpn) & 0xfff;
    perm = vpt[vpn] & 0xFFF;   //vpn即virtual page number;   perm对应二级页表内容的低12位;
                             //  #define vpt ((volatile Pte *)UVPT);  vpt是定义在用户态的include/
    addr = vpn << PGSHIFT;  // 等价于 addr = vpn * BY2PG;  [PGSHIFT=12; BY2PG=4096]

    /* vpt: virtual page table
     * vpd: vitrual page dictionary
     * vpd = vpt + (vpt << PGSHIFT)
    */


    /* 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) */
    if(!(perm &  PTE_D) || (perm & PTE_LIBRARY)) {
        syscall_mem_map(0, addr, envid, addr, perm);  //直接映射给子进程（dstid是envid[子id]）
    } else {
        perm = (perm & ~PTE_D) | PTE_COW;     //修改权限
        syscall_mem_map(0, addr, envid, addr, perm);   //先映射给子
        syscall_mem_map(0, addr, 0, addr, perm);       //再重映射给父
    }

}
```

Exercise 4.11

根据上述提示以及代码注释，完成kern/tlbex.c 中的do_tlb_mod 函数，设置好保存的现场中EPC 寄存器的值。

> do_tlb_mod 函数负责将当前现场保存在异常处理栈中，并设置a0 和EPC 寄存器的值，使得从异常恢复后能够以异常处理栈中保存的现场（Trapframe）为参数，跳转到env_user_tlb_mod_entry 域存储的用户异常处理函数的地址。

```c
#if !defined(LAB) || LAB >= 4
/* Overview:
 *   This is the TLB Mod exception handler in kernel.
 *   Our kernel allows user programs to handle TLB Mod exception in user mode, so we copy its
 *   context 'tf' into UXSTACK and modify the EPC to the registered user exception entry.
 *
 * Hints:
 *   'env_user_tlb_mod_entry' is the user space entry registered using 'sys_set_user_tlb_mod_entry'.
 *
 *   The user entry should handle this TLB Mod exception and restore the context.
 */
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;
    }
    tf->regs[29] -= sizeof(struct Trapframe);
    *(struct Trapframe *)tf->regs[29] = tmp_tf;

    if (curenv->env_user_tlb_mod_entry) {
        tf->regs[4] = tf->regs[29];       // $a0寄存器的值被设置为 $sp[帧指针]的值
        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;   //设置epc寄存器的值

    } else {
        panic("TLB Mod but no user handler registered");
    }
}
#endif
```

Exercise 4.12

完成kern/syscall_all.c 中的sys_set_tlb_mod_entry 函数。

> 使用syscall_set_tlb_mod_entry函数来注册自身的页写入异常处理函数，也就是上文提到的env_user_tlb_mod_entry 域指向的用户处理函数。这里需要通过系统调用告知内核自身的处理程序是cow_entry。
>
> 内核中的系统调用处理函数sys_set_tlb_mod_entry，将进程控制块的env_user_tlb_mod_entry域设为传入的参数。

```c
/* Overview:
 *   Register the entry of user space TLB Mod handler of 'envid'.
 *
 * Post-Condition:
 *   The 'envid''s TLB Mod exception handler entry will be set to 'func'.
 *   Returns 0 on success.
 *   Returns the original error if underlying calls fail.
 */
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));       //先取得envid对应的进程

    /* 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;    //将进程的tlb_mod处理入口设置为自定义处理函数func

    return 0;
}
```

Exercise 4.13

填写user/lib/fork.c 中的cow_entry 函数。

> 1.根据vpt 中va 所在页的页表项，判断其标志位是否包含PTE_COW，是则进行下一步，否则调用user_panic() 报错。
>
> 2.分配一个新的临时物理页到临时地址UCOW，使用memcpy 将va页的数据拷贝到刚刚分配的页中。
>
> 3.将发生页写入异常的地址va映射到临时页面上，注意设定好对应的页面标志位（即去除PTE_COW 并恢复PTE_D），然后解除临时地址UCOW 的内存映射。

```c

/* Overview:
 *   Map the faulting page to a private writable copy.
 *
 * Pre-Condition:
 *  'va' is the address which led to the TLB Mod exception.
 *
 * Post-Condition:
```

```
 *  - Launch a 'user_panic' if 'va' is not a copy-on-write page.
 *  - Otherwise, this handler should map a private writable copy of
 *    the faulting page at the same address.
 */
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
    u_int va = tf->cp0_badvaddr;  // va是CPU设置的发生页写入异常的地址。
    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;   //权限为对应低12位。 （先将va转成VPN[virtual page number]，再在vpt二级页表中找对应）
    if(!(perm & PTE_COW)) {
        user_panic("page of va 0x%x doesn't have PTE_COW", va);
    }

    /* 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;     //修改权限

    /* Step 3: Allocate a new page at 'UCOW'. */
    /* Exercise 4.13: Your code here. (3/6) */
    syscall_mem_alloc(0, UCOW, perm);  //分配一个新的临时物理页到临时地址UCOW

    /* 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) */
    va = ROUNDDOWN(va, BY2PG);   //宏ROUNDDOWN(a, n),它的作用是将a按n向下对齐,要求n必须是2的非负整数次幂。
                                 //将va按页大小向下对齐。
    memcpy((void* )UCOW, (const void* )va, BY2PG);  //使用memcpy 将va页的数据拷贝到刚刚分配的页中。(大小为BY2PG)

    // 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, UCOW, 0, va, perm);   //将临时页面UCOW映射到发生页写入异常的地址va上。（注意新权限perm）

    // Step 6: Unmap the page at 'UCOW'.
    /* Exercise 4.13: Your code here. (6/6) */
    syscall_mem_unmap(0, UCOW);           //解除临时地址UCOW 的内存映射。

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

Exercise 4.14

填写kern/syscall_all.c 中的sys_set_env_status 函数。

> 父进程还需要使用syscall_set_tlb_mod_entry，设置子进程的页写入异常处理函数为cow_entry。最后，父进程通过系统调用syscall_set_env_status 设置子进程为可以运行的状态。
>
> 内核中实现sys_set_env_status 函数时，不仅需要设置进程控制块的env_status 域，还需要在env_status从ENV_NOT_RUNNABLE 转换为ENV_RUNNABLE 时将控制块加入到可调度进程的链表中，反之则从链表中移除。

```c
/* Overview:
 *   Set 'envid''s 'env_status' to 'status' and update 'env_sched_list'.
 *
 * Post-Condition:
 *   Returns 0 on success.
 *   Returns -E_INVAL if 'status' is neither 'ENV_RUNNABLE' nor 'ENV_NOT_RUNNABLE'.
 *   Returns the original error if underlying calls fail.
 *
 * Hint:
 *   The invariant that 'env_sched_list' contains and only contains all runnable envs should be
 *   maintained.
 */
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) { //先检查要设置的状态是否正确
        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));   //取得envid对应的进程

    /* Step 3: 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 != status) {    //需要设置新状态
        if(status == ENV_RUNNABLE) {
            TAILQ_INSERT_TAIL(&env_sched_list, env, env_sched_link);  //插入空闲链表尾部
        } else {
            TAILQ_REMOVE(&env_sched_list, env, env_sched_link);
        }
    }

    /* Step 4: Set the 'env_status' of 'env'. */
    env->env_status = status;
    return 0;
}
```

Exercise 4.15

填写user/lib/fork.c 中的fork 函数。

> fork （用户态实现）中父进程在syscall_exofork 后还需要做的事情有：
>
> 1. 遍历父进程地址空间，进行duppage（父子地址映射）。
> 2. 设置子进程的异常处理函数，确保页写入异常可以被正常处理。
> 3. 设置子进程的env_status，允许其被调度。最后再将子进程的envid 返回，fork 函数就完成了。

```c
/* Overview:
 *   User-level 'fork'. Create a child and then copy our address space.
 *   Set up ours and its TLB Mod user exception entry to 'cow_entry'.
 *
 * Post-Conditon:
 *   Child's 'env' is properly set.
 *
 * Hint:
 *   Use global symbols 'env', 'vpt' and 'vpd'.
 *   Use 'syscall_set_tlb_mod_entry', 'syscall_getenvid', 'syscall_exofork',
and 'duppage'.
 */
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));
    }

    /* 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();        //创建子进程，查看返回值
    if (child == 0) {    //返回值为0说明现在处在刚创建的子进程下。
        env = envs + ENVX(syscall_getenvid());      //env指针始终指向当前进程。
ENVX 【env index】
        return 0;
    }

    /* 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) */
    /*  another-way-1:
    u_int j;
    for (i = 0; i < USTACKTOP; i += PDMAP) {
        if (*(vpd + PDX(i)) & PTE_V) {
            for (j = 0; (j < PDMAP) && (i + j < USTACKTOP); j += BY2PG) { // 防止未对齐
                if(*(vpt + VPN(i + j)) & PTE_V) {
                    duppage(child, VPN(i + j));
                }
            }
        }
    } */
    //将USTACKTOP地址以下的所有页面映射进子进程的地址空间。 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。
    for(i=0; i < PDX(ROUND(USTACKTOP, PDMAP)); i++) {  //PDX() 【page directory index】
        //     PDMAP=4*1024*1024 【page directory map】 bytes mapped by a page directory entry
        if(vpd[i] & PTE_V) {  //一级页表有效(如果PTE_V为0(V-valid),则任何访问该地址的操作都将引发TLB异常)
            u_int j;
            for(j=0; j<1024; j++) {   //一级页表项对应1024个二级页表项
                u_int vpn = (i << 10) | j;     //vpnumber是 （一级index-10位，二级index-10位）
                if((vpn < VPN(USTACKTOP)) && (vpt[vpn] & PTE_V)) {  //二级页表有效
                    duppage(child, vpn);
                }
            }
        }
    }

    /*   another-way-2:
    for (i = 0; i < VPN(USTACKTOP); i++) {            //#define VPN(va) (((u_long)(va)) >> 12)
        if ((vpd[i >> 10] & PTE_V) && (vpt[i] & PTE_V)) {    //这里i取得高20位
            duppage(child, i);
        }
    }
    */

    /* 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) */
    syscall_set_tlb_mod_entry(child, cow_entry);    //child要么0（父进程）要么子的envid
    syscall_set_env_status(child, ENV_RUNNABLE);

    return child;
}
```