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];

```

```c
    /* 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()

```c
/* 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);

    /* 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);
}
```

```
/* 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。