lab4 - 实验报告**

本单元地址布局

每个进程都有相同的虚拟地址划分方式，并按照 mmu.h 文件中所示进行排布，也就是说每个进程都具有一张这样的表，同时每个进程的 kseg0、kseg1 段也都存放（或者说映射）着内核相关的数据结构，存在于所有进程的虚拟空间中，相当于被所有进程**只读共享**，所以为了方便使用，它们被整体映射到物理地址的固定区域；对于 kuseg 段，用户的页表和虚拟地址相结合，会指向物理内存中的某些空间，不同进程可能对物理空间进行共享，执行系统调用，**汇编层面上**就是从 kuseg 段的汇编指令跳转至 kseg0 段，（进入内核态）并执行特定序列（系统调用函数），最后返回用户态 EPC **C语言层面上**就是用户态函数和内核态的系统调用函数之间的调用跳转

KSTACKTOP：内核栈栈顶

KERNBASE：内核代码起始地址，内核的代码段、数据段、入口函数都从这里开始加载

ULIM：内核空间与用户空间的分界线，用户程序不能访问其以上空间

UVPT：映射页表本身

UPAGES：映射所有物理页信息 pages[]，用户可以只读访问

UENVS：映射所有进程信息 envs[]，用户可以只读访问

UXSTACKTOP：用户异常栈栈顶（父子进程复制的地址**终点**）

USTACKTOP：用户正常栈栈顶，其下空间是普通用户代码与数据段

UTEXT：用户代码起始地址（父子进程复制的地址**起点**）

UCOW：系统保留页，用于实现 COW 的临时映射

UTEMP：系统保留页，作为临时页映射区，系统调用或内核辅助用户操作内存时用于暂存的一页

0x003fe000 以下：无效地址或保留区域，不能访问，用于错误检验

函数汇总

```
// 系统宏函数

PAGESIZE : 当 4096 用就行，表示一个页大小为 4KB;
PDMAP : 一个页目录映射的虚拟内存大小，值为 4MB;
PGSHIFT : vpn << PGSHIFT → 得到虚拟地址，va >> PGSHIFT → 得到虚拟页号，等价于(VPN);
PDSHIFT : 将地址 va 右移 22 位可得到页目录索引，等价于(PDX);
ROUND(a, n) : a 向上对齐到 n 的整数倍;
ROUNDDOWN(a, n) : a 向下对齐到 n 的整数倍;
ENVX() : 给定一个 envid，提取出它在 envs 数组中的索引值;
vpd[] : 虚拟页目录，页目录虚拟地址的映射，用户态能看到的页目录内容，通过 PDX 索引访问;
vpt[] : 虚拟页表，页表虚拟地址空间的映射,用户态能看到的页表内容，通过 VPN 索引访问;
PDX(va) : 获取虚拟地址 va 所在页目录项的索引;
PTX(va) : 获取虚拟地址 va 所在页表项的索引;
VPN(va) : 获取虚拟地址 va 的虚拟页号，也就是高 20 位左移的结果;
PPN(va) : 获取虚拟地址 va 的虚拟页号，也就是高 20 位左移的结果;
PTE_ADDR(pte) : 从页表项中提取物理页基地址，屏蔽低 12 位标志位;
PTE_FLAGS(vpt[VPN(va)]) : 提取一个页表项中的权限位，也就是低 12 位;

// 底层功能函数

void sys_putchar(int c) : 向控制台输出字符 c;

int sys_print_cons(const void *s, u_int num) : 向控制台输出字符串 s 中长度为 num 的部分;

int sys_cgetc(void) : 从控制台获取一个字符;

void sys_panic(char *msg) : 接收到 msg 信息的时候，触发系统崩溃;

// 其他系统调用函数

u_int sys_getenvid(void) : 获取运行中进程 curenv 的进程 id，和返回 0 其实没区别;

void __attribute__((noreturn)) sys_yield(void) : 运行中进程 curenv 主动放弃处理机，执行 schedule(1)，注意这个函数是没有返回值的;

int sys_env_destroy(u_int envid) : 销毁掉 envid 对应的那个进程;
```

```
int sys_set_tlb_mod_entry(u_int envid, u_int func) : 设置 tlb_mod 入口函数，把 envud 对应进程的 tlb_mod 的入口函数设置为 func 对应的那个函数，在本单元其实就是设成了 cow_entry;

static inline int is_illegal_va(u_long va) : 判断 va 这个虚拟地址是否不合法;

static inline int is_illegal_va_range(u_long va, u_int len) : 判断 [va, va + len) 这个虚拟地址范围是否不合法;

int envid2env(u_int envid, struct Env **penv, int checkperm) : 根据 envid 获取其对应的进程控制块 env，需要特殊规定如果是要获取当前运行中进程，就传入 envid == 0，用 0 表示 curenv 的 env_id，其中 checkperm 表示权限验证标志，为 0 的时候无特殊要求，为 1 的时候需要验证当前进程能否操作 envid 对应的进程，也就是看 envid 对应的进程是否为当前进程本身或者它的子进程;

// 重要系统调用函数

int sys_mem_alloc(u_int envid, u_int va, u_int perm) : 为进程建立一个虚拟地址映射，在 envid 对应的进程环境下，为虚拟地址 va 建立一个与之映射的物理页，并设置页表权限位为 perm;

int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm) : 为两个建成建立一个共同的虚拟地址映射，首先在 srcid 对应的进程环境中，找到虚拟地址 srcva 所映射到的物理页，然后在 dstid 对应的进程环境中，把 dstva 这个虚拟地址也映射到同一个物理页，并设置页表权限位为 perm;

int sys_mem_unmap(u_int envid, u_int va) : 为进程解除一个虚拟地址映射，在 envid 对应的进程环境下，取消 va 这个虚拟地址和它的物理页之间的映射关系;

int sys_exofork(void) : 为运行中进程 curenv 创建一个子进程;

int sys_set_env_status(u_int envid, u_int status) : 修改进程的状态，把 envid 对应的进程的状态参数修改为 status，并据此更新调度队列;

int sys_set_trapframe(u_int envid, struct Trapframe *tf) : 修改进程的上下文，把 envid 对应的进程上下文修改为 tf;

// 进程间通信(IPC)

int sys_ipc_recv(u_int dstva) : 运行中进程 curenv 等待接收信息 dstva，当前进程会被阻塞，让出处理机，直到再次被唤醒;

int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm) : 进程尝试发出信息，envid 对应的进程会发出一个值为 value 的信息，同时会发出一个待映射的虚拟地址 srcva，和映射页表的有效位 perm;

// 硬件读写有关

static inline int is_illegal_dev_range(u_long pa, u_long len) : 寻找地址合法的硬件设备，遍历所有硬件设备，返回找到的第一个物理地址在 [pa, pa + len) 范围内的合法的硬件，如果找不到会返回相应错误码;

int sys_write_dev(u_int va, u_int pa, u_int len) : 向硬件设备写入，把 [va, va + len) 范围内的信息写入 [pa, pa + len) 范围内的硬件设备中;

int sys_read_dev(u_int va, u_int pa, u_int len) : 从硬件设备读出，把硬件设备中物理地址在 [pa, pa + len) 范围的信息读出到虚拟地址 [va, va + len) 的空间内;

```

```
// 执行系统调用

void do_syscall(struct Trapframe *tf) : 进行系统调用，根据 tf 的 regs[4] 中存储的 sysno 值，调用相应编号的系统函数，需要传入 6 个参数，并接收 1 个返回值;

// COW 机制有关


static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) : 写异常发生时复制被写的页面，将其设置为可写后分配给尝试写该页面的进程;

static void duppage(u_int envid, u_int vpn) : 让 envid 对应的进程也能够访问 vpn 这个虚拟地址索引对应的虚拟页，根据 vpn 索引到的页表权限位，考虑是子进程直接映射，还是设置为 PTE_COW 再同时修改父子映射;

int fork(void) : 用户级别进程的创建，创建一个子进程并且复制父进程的地址空间，设置父子进程的 TLB 异常修改入口为 cow_entry 函数的地址;
```

常见宏变换

```
env = envs + ENVX(syscall_getenvid());
syscall_getenvid() : 获得当前进程的 envid;
envs + ENVX(envid) : 由 envid 获得 env 进程控制块，存入 env 这个 Env * 类型的指针中;

vpd 是页目录首地址，以 vpd 为基地址，加上页目录项偏移数即可指向 va 对应页目录项的虚拟地址，即 vpd + (va >> 22)，而对这个地址解引用，即可得到这个页目录项所指向页表始址的物理地址，即 *(vpd + (va >> 22)) 或者 vpd[va >> 22];

二级页表的物理地址 : vpd[va >> 22] & (~0xfff)，这里注意要屏蔽掉低 12 位标志位;

提前判断有效位: (vpd[va >> 22] & PTE_V) 或 (vpd[VPN(va) >> 10] & PTE_V)，后者是先得到虚拟页号 VPN 之后再左移 10 位，二者是等价的;

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)，而对这个地址解引用，即可得到这个页表项所指向的物理页面的物理地址，即 *(vpt + va >> 12) 或 vpt[va >> 12];
这里格外需要注意的是，vpt 是以所有页表项为整体索引的，而不是以每个页目录所对应的页表页分别索引的，所以 va 对应的页表项索引就是 VPN(va)，而不是 TDX(va)，这里很容易错;

物理页面地址：vpt[va >> 12] & (~0xfff)，这里要注意屏蔽掉低 12 位标志位;

提前判断有效位: (vpt[va >> 12] & PTE_V) 或 (vpt[VPN(va)] & PTE_V);

vpn = VPN(va) = va >> 12 (虚拟页号)，也就是 va 高 20 位左移的结果;

这里常考的还是 i j 两重循环，分别遍历页目录项和页表项的形式，可以参考 lab2 课上的实现，也可以参考 fork 函数的实现，只是这里我们索引页目录和页表的方式，因为是在用户态下访问，变成了借助 vpd 和 vpt 访问，而且这里遍历 vpd 和 vpt 的形式不尽相同，需要注意:
  for (i = 0; i < 页目录范围上界; i++) {
    if (vpd[i] & PTE_V) {
      for (u_int j = 0; j < 1024; j++) {
        u_long va = (i * 10 + j) << 12;
        if (va >= 页目录范围上界对应的虚拟地址上界) {
          break;
        }
        if (vpt[VPN(va)] & PTE_V) {
```

```
          do_something...
        }
      }
    }
  }

操作内核栈 : *((struct Trapframe *)KSTACKTOP - 1) = *tf; // 全局赋值
或者 : ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0; // 单点赋值
```

课上 exam

根据往年题，能考的地方有三点：

1. 考察系统调用的流程，从用户态到内核态一路修改下来
2. 考察一个具体的内核系统调用函数的实现，可能涉及课下的宏或者其余系统调用函数
3. 对 ipc 两个系统调用函数的扩展

因为往年是 lab4 分两次分开考的，一次一个考点，今年可能会把这三部分合起来一起考在同一个题目里

共享页设计

题目要求：实现一个函数，使得虚拟地址 `va` 对应的页面成为**共享页面**，可在多个进程之间共享。也就是说：

1. 如果该虚拟地址还没有被分配内存，先给当前进程分配一个物理页
2. 如果该页是有效的、且是可写的普通页（非共享页），将它重新标记为共享页（`PTE_LIBRARY`）
3. 让这个页变成**写时共享或读写共享的内存页**，使多个进程可以映射到它，并共享访问

code：

```
int make_shared(void *va) {
// 初始化一个页的权限，表示既可写又有效
    u_int perm = PTE_D | PTE_V;
// 判断当前地址 va 对应的页目录项和页表项都有效，如果有一个无效，就在当前进程下为 va 分配一个物理页
    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，尝试分配一个物理页，将传入的 envid 设为 0，表示默认 curenv，如果既不存在又分配失败，就只能直接返回 -1
        if (syscall_mem_alloc(0, ROUNDDOWN(va, BY2PG), perm) != 0) {
            return -1;
      }
    }
// 获得 va 的 perm
    perm = vpt[VPN(va)] & 0xfff;
// 检查是否非法或不能共享，非法是指当前 va 超过用户空间最大值 UTOP，不能共享是指不能写，如果不满足其中之一，直接返回
    if (va >= (void *)UTOP ||
        ((vpd[va >> 22] & PTE_V) && (vpt[va >> 12] & PTE_V) && !(perm & PTE_D))) {
        return -1;
    }
// 为当前页的权限加上可共享权限，也就是 PTE_LIBRARY
    perm = perm | PTE_LIBRARY;
// 恢复出虚拟地址 addr = va 的虚拟页号 << 12
    u_int addr = VPN(va) << 12;
// 把这个页重新映射到自己，此时的权限中已经带有共享标志 PTE_LIBRARY，这是操作系统重修改权限的一种手段，因为系统调用不能直接修改页表，只能通过重新建立自身映射的方式更新页表项中的权限位
```

```
    if (syscall_mem_map(0, (void *)addr, 0, (void *)addr, perm) != 0) {
      return -1;
  }
// 返回对应的物理地址，也就是页表项中的内容去掉低 12 位的值，注意一个物理页大小是 4KB，也就是 4096 的大小
    return ROUNDDOWN(vpt[VPN(va)] & (~0xfff), BY2PG);
}
```

本例重点考察两个系统调用函数：

1. sys_mem_alloc：在某个进程下为某个虚拟地址分配一个物理页
2. sys_mem_map：这里用到它的特殊作用，修改某个进程下某个虚拟地址的映射权限（**重要**）

```
syscall_mem_map(0, (void *)addr, 0, (void *)addr, perm);// 其中 perm 是修改后的权限
```

3. 还有一些小知识点：比如如何利用 vpd 和 vpt 索引，如何设置权限位等，参考注释即可

通过本例，我们还注意到一个地方，就是 sys_mem_alloc 和 sys_mem_map 这两个函数要求传入的虚拟地址 va 都必须是低 12 位对齐的地址，我们有以下三种方式把一个 void *va 虚拟地址对齐：

1. ROUNDDOWN(va, BY2PG); 这个宏返回值就是低 12 位对齐的
2. u_long addr = VPV(va) << 12; addr 就是低 12 位对齐的
3. va & (~0xfff); 得到的结果也是低 12 位对齐的

如何添加一个系统调用

题目要求：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，要求熟悉整个系统调用的主流程，一步一步修改每一个对应的函数

1. 在 user / include / lib.h 中添加我们需要的函数的两个声明，一个是函数**本身**在用户态下的定义，也就是 user_lib_func，另一个是**用户态**下启动系统调用的函数定义，也就是 syscall_func，内核态不需要声明，但是用户态是需要的：

```
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 中添加用户态函数 syscall_func 的函数体，它负责调用 msyscall 函数，这里我们只需要根据外层的 syscall_func 的参数，设定好 msyscall 函数的入参，注意第 msyscall 的第一个入参是我们规定的系统调用号 sysno，具体的 msyscall 函数体不需要修改，需要根据 syscall_func 是否有返回值进行区分，如果有返回值，直接 return msyscall 函数的返回值：

```
void syscall_func(u_int envid, u_int value, const void *srcva, u_int perm) {
    msyscall(SYS_func, envid, value, srcva, perm);// if msyscall has return, just return
}
```

3. 在 user / lib 中建立一个新的文件，命名为 func.c，这个文件是用来存放我们用户顶层函数的地方，在其中实现 user_lib_func 这个函数本身，该函数内部又会调用 syscall_func 函数：

```c
void user_lib_func(u_int whom, u_int val, const void *srcva, u_int perm)
{
  //...
  syscall_func(envid, value, srcva, perm);
  //...
}
```

4. 在 include / syscall.h 文件中的 enum 类的 MAX_SYSNO 前面加上 SYS_func，也就是实现 func 这个内核函数系统调用号的定义

```c
#ifndef SYSCALL_H
#define SYSCALL_H
#ifndef __ASSEMBLER__
enum {
      // add new sysno here
        SYS_func,
        MAX_SYSNO,
};
#endif
#endif
```

5. 在 kern / syscall_all.c 文件的调用函数表中增加一条表项，建立从 sysno 到 sys_* 的映射关系，具体来讲就是 SYS_func 指向 sys_func 这个函数的索引，注意后面要加个逗号

```c
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,
    // add new index here
    [SYS_func] = sys_func,
};
```

6. 在 kern / syscall_all.c 文件中具体实现该系统调用函数的功能，这里就需要根据题目描述的具体过程来实现了：

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

实现屏障同步机制

题目要求：当多个环境（`env`，即用户进程）调用 `sys_barrier_wait` 时，它们会在此同步点等待，直到满足某个人数（`barrier_num`），一旦有 `barrier_num` 个进程调用该函数，所有等待进程将同时通过屏障，恢复为可运行状态

code：

```
u_int sys_barrier_wait(u_int* p_barrier_num, u_int* p_barrier_useful) {
// 注意以下 4 个变量都是 static 的
  static u_int env_not[100]; // 记录哪些进程已经被屏障并挂起
  static u_int N = 0;        // 期望同步的总人数
  static u_int num = 0;      // 剩余的未达到屏障的人数
  static u_int useful = 0;   // 屏障是否被激活，为 1 表示屏障在起作用

// 判断是否需要更新屏障参数，如果传入的屏障人数 *p_barrier_num 大于当前记录的 N，说明是一次新的屏障调用或新的回合开始，此时需要更新屏障人数 N，等待人数 num，是否激活 useful 标志
  if ((*p_barrier_num) > N) {
    N = (*p_barrier_num);
    num = N;
    useful = (*p_barrier_useful);
  }
// 判断剩余人数为 0，达到屏障人数，进程可以直接运行，先关闭屏障，再返回 ENV_RUNNABLE 标识
  if (num == 0) {
      useful = 0;
      return ENV_RUNNABLE;
  }
// 如果屏障依然有效，则说明全部环境还没有同步，当前进程需要等待
  if (useful == 1) {
    // 先检查当前进程是否已经被阻塞过，如果有，直接返回 NOT_RUNNABLE 标识
    for (u_int i = 0; i < N - num; i++) {
      if (env_not[i] == curenv->env_id) {
          return ENV_NOT_RUNNABLE;
      }
    }
    // 如果还没被阻塞过，则把他加入到阻塞队列中，同时剩余等待的进程数 -1，然后再返回 NOT_RUNNABLE 标识
    env_not[N - num] = curenv->env_id;
    num--;
    return ENV_NOT_RUNNABLE;
  }
  // 此时说明屏障未生效，直接返回 RUNNABLE 标识即可
  return ENV_RUNNABLE;
}
```

这种类型的上机题不是难在对课下的掌握，而是难在对课上指导书的理解，要敢于**多设变量**（局部变量或者 static 静态变量），**多写分支**，不要担心打破原有函数框架，基于这样的大思路，下面给出两份等价的代码实现：

```
// version-1
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) {
    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) { //first version
        useful = 0;
        return ENV_RUNNABLE;
    }
    return ENV_NOT_RUNNABLE;
  }
  return ENV_RUNNABLE;
}
// version-2
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) {
    N = (*p_barrier_num);
    num = N;
    useful = (*p_barrier_useful);
  }
  if (useful == 1) {
    if (num == 0) { //second version
        useful = 0;
        return ENV_RUNNABLE;
    }
    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--;
    return ENV_NOT_RUNNABLE;
  }
  return ENV_RUNNABLE;
}
```

进程组 ipc 通信

题目要求：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);
```

同时添加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，增加以下新的错误码，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. 系统调用主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），需要将其复制到 `user/lib/ipc.c` 的位置处
5. 在 include / syscall.h 中，定义两个新的系统调用号，注意新增系统调用号的位置，应该位于 MAX_SYSNO 之前
6. 在 user / lib / syscall_lib.c 中，实现两个用户级的系统调用函数，发起用户级的系统调用
7. 在 kern / syscall_all.c 中，添加两个系统调用在内核中的实现函数，需要保证个函数的定义位于系统调用函数表之前
8. 在 kern / syscall_all.c 中，在系统调用函数表中，添加新增的两个系统调用号对应的内核函数指针
9. 在 kern / syscall_all.c 中，具体实现这两个系统调用函数

内核函数具体实现：

code：

```
// 函数功能 ： 该系统调用尝试向同一组(Group)中的另一个进程(环境)发送消息(包括整数值和可选页面)，只有当对方正在接收消息且属于相同分组时才成功，与普通 sys_ipc_try_send() 唯一不同是加了 e->env_gid != curenv->env_gid 检查，即仅允许向同一组的进程发送消息
int sys_ipc_try_group_send(u_int envid, u_int value, u_int srcva, u_int perm) {
        struct Env *e;
        struct Page *p;

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

        try(envid2env(envid, &e, 0));

```

```
        if (e->env_ipc_recving == 0) {
                return -E_IPC_NOT_RECV;
        }
// 增加一条特判 : 本例和课下唯一的不同，就是增加了对 gid 的判断，只有两个进程的 gid 相同时，才能进行同组内的通信
        if (e->env_git != curenv->gid) {
                return -E_IPC_NOT_GROUP;
        }

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

// 函数功能 : 设置当前进程的组标识符(GID)，用于实现分组通信
void sys_set_gid(u_int gid) { // 简单的赋值函数
  curenv->env_gid = gid;
  return;
}
```

广播 IPC

题目要求：实现一个广播 IPC 的机制，即将一个消息从当前环境（进程）光报给他所有的**子孙**进程（其实这是个 extra 的题目，不知道为什么连续两年考了同一道题）

code：

```
//kern/syscall_all.c
extern struct Env envs[NENV]; // envs 的定义是在 env.c 文件中，这里不需要重新定义，只是引用即可，故而需要加上外部变量标识 extern
// 入参含义 : 传递的消息内容 value，需要映射的页面信息(srcva==0表示不需要页面映射)，映射页面的权限
int sys_ipc_try_broadcast(u_int value, u_int srcva, u_int perm) {
  struct Env *e;
  struct Page *p;

// 仿照 sys_ipc_try_send 检查 va 合法性
  if (srcva != 0 && is_illegal_va(srcva)) {
    return -E_IPC_NOT_RECV;
  }

// 找到当前 curenv 的所有子孙进程，其中 NENV 是最大进程数，这里不仅要找 curenv 的直接子进程，它的子进程的子进程，类似于并查集中 find(x) == root 的所有 x 节点，在一个联通块内的都算做是 curenv 的子进程
  int signal[NENV];
```

```
15
16    // 首先先找到 curenv 的所有直接子进程，直接线性扫描一遍 signal 数组即可
17    for (u_int i = 0; i < NENV; i++) {
18      if (curenv->env_id == envs[i].env_parent_id) {
19        signal[i] = 1;
20      } else {
21        signal[i] = 0;
22      }
23    }
24
25    // 接着遍历 curenv 的所有子进程的子进程(也就是所有 signal 标记为 1 的那些进程)，对于某个子进程 i 满足 siganl[i]==1，我们再扫描整个 envs 数组寻找它的子进程，只要在一轮循环中，还有新的 siganl[j]=1 被加入，就不能结束循环，直到所有的 signal[j]==1 都被找到
26    int flag = 0;
27    while(flag == 0) {
28      flag = 1;
29      for (u_int i = 0; i < NENV; i++) {
30        if (signal[i] == 1) {
31              for (u_int j = 0; j < NENV; j++) {
32            if (signal[j] == 0 && envs[i].env_id == envs[j].env_parent_id) {
33              signal[j] = 1;
34              flag = 0;// 控制循环继续的标识
35            }
36          }
37        }
38      }
39    }
40  // 其实到这里关键的步骤也就是找到 curenv 的所有联通子进程，这一步已经完成了，那么问题就变成了 curenv 要向这些已知的子进程传播信息，直接调用实现好的 sys_ipc_try_send 函数即可
41    for (int i = 0; i < NENV; i++) {
42      if (signal[i]) {
43        // 这里 signal 数组中记录的是 envs 数组的索引，根据索引可以找到 env 控制块，有了控制块就可以得到进程的一切信息了
44        sys_ipc_try_send(env[i].env_id, val, srcva, perm);
45      }
46    }
47    return 0;
48  }
```

本例的实现重点是如何找到 curenv 进程的所有子孙进程，尤其是 e->parent_id->parent_id->.....->parent_id == curenv->env_id 这样的非直接子进程，示例代码中把递归展开成循环写，可能是为了防止爆栈的风险，这里也可以借助广搜来实现，因为是 C 所以需要自己手动实现一个**队列**（普通队列即可）：

```
1   void bfs() {
2     int signal[NENV + 5];
3     int q[NENV + 5];
4     int hh = 0, tt = 0;
5     q[tt++] = 0;
6     signal[0] = 1;
7     while(hh < tt) {
8       int u = q[hh++];
9       for (int v = 0; v < NENV; v++) {
10        if (!signal[v] && envs[v]->parent_id == envs[u]->env_id) {
11          signal[v] = 1;
12          q[tt++] = v;
13        }
```

```
    }
  }
  return;
}
```

这里再给出另一位学长实现的代码，他的实现形式是直接找一个 child 数组存 curenv 的所有后代进程的 **envid**，而不是在整个 envs 数组上建立数组索引的标记，这样的好处就是可以通过仅遍历 child 数组就快速且直接地拿到我们所需要的子进程 envid：

```
int sys_ipc_broadcast(u_int val, void *srcva, u_int perm) {
  u_int childs[NENV];
  for (int i = 0; i < NENV; 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 */
  for (int i = 0; i < NENV; i++) {
    if (envs[i].env_parent_id == curenv->env_id) {
      for (int j = 0; j < NENV; 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 < NENV; 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++) {
      sys_ipc_try_send(childs[i], val, srcva, perm);
  }

```

```
  return 0;
}
```

添加自旋锁

考察的还是添加系统调用的流程，最后要我们实现的就是获取锁和释放锁的过程，因为内核本身就保证了原子性，因此我们仅做最简单的实现即可

code：

```
// 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;
}
```

```
48
49 int sys_release_console(void) {
50   if (lock == curenv->env_id) {
51     lock == -1;
52     return 0;
53   }
54   return -1;
55 }
```

系统调用

系统调用的本质

系统调用：为什么会有系统调用?

因为用户进程本不能访问系统的内核空间，但用户进程在某些特定场景下需要执行一些只能由内核完成的操作（比如操作硬件，动态分配内存，进程间通信），实现这部分功能的代码如果让用户提供，我们在内核态执行，这显然是不安全的，因此我们考虑把实现这些功能的代码封装成**内核函数**，用户要进行这些操作的时候，会引发特定异常陷入内核态，由内核调用我们封装好的这些函数，从而安全地实现上述功能，这个机制就称为**系统调用**

我们也可以认为**系统调用**是实现用户进程和内核之间进行通信的重要实现方式，它可以看成是一种特殊的异常或中断，只是先前我们接触的异常大多不能被用户主动触发（如时钟中断，TLB MISS 等），而系统调用则是由**用户主动发起**的异常，可以理解成用户通知内核，然后由内核完成某个功能

用户进程通过执行`syscall`语句，使得进程发生自陷，陷入内核的异常处理程序中，由内核根据系统调用时的上下文执行相应的内核函数，完成相应的功能，并最终返回到 syscall 的后一条指令，换言之，执行具体操作的是内核中的函数，而用户进程只是触发这个函数并且把函数执行需要的有关参数传过去，之后的执行就和用户进程无关了

系统调用的顺序

1. 用户执行用户态函数`syscall_*`以后，在该函数内部会调用用户态函数`mysyscall`，接着执行`syscall`指令，进入内核态
2. 进入内核态后，处理器把 PC 寄存器指向内核中一个固定的**异常处理入口**，也就是内核态函数`handle_sys`，该函数会实现其内部封装好的`do_syscall`函数，进而调用内核函数`sys_*`，实现具体功能
3. 执行完毕后回到用户态，继续执行`syscall`后面的下一条语句

```
1  用户程序(用户态会有一个入口程序，这个程序里面会调用 syscall_*(...))
2    ↓
3  调用 syscall_*(...) (用户态封装函数)
4    只有一句话 : 调用 msyscall 函数
5    ↓
6  调用 msyscall(...) (封装系统调用的汇编入口)
7    只有两步 : 调用 syscall 函数; jr ra 返回
8    ↓
9  执行 syscall 指令(陷入内核)
10   1.使用 SAVE_ALL 宏：将用户进程的上下文运行环境保存在内核栈中
11   2.取出 CP0_CAUSE 寄存器中的异常码，系统调用对应的异常码为 8
12   3.以异常码为索引在 exception_handlers数组中找到对应异常处理函数 handle_sys
13   4.转跳至 handle_sys 函数处理用户的系统调用请求
14   ↓
15 CPU 切换到内核态，并跳转到内核统一入口点
16   ↓
17 handle_sys(...)(分发系统调用)
```

```
  1.由 SAVE_ALL 作用之后，内核栈顶保存的是 Trapframe 结构体的起始地址，将之存入 a0 寄存器作为 do_syscall 的传入参数
  2.调用 do_syscall 实现处理系统调用
  3.调用 ret_from_exception 从内核态返回用户程序(被阻塞后调度的进程相当于跳过 do_syscall 直接跳到这一步)
  ↓
调用 do_syscall(...)
  1.改 epc 使得由内核态返回用户态之后能够执行 msyscall 函数中的 jr ra 指令
  2.获得参数，通过 func(arg1, arg2, arg3, arg4, arg5) 直接调用内核中相应的系统调用函数， 也就是 sys_* 函数
  ↓
调用 sys_*(...)(具体内核服务函数)
  ↓
返回结果 → 回到用户态
```

具体过程详解

- 用户提出请求，用户态调用`syscall_*()`函数，他们都位于`user/lib/syscall_lib.c`文件中

在 MOS 中，操作系统为用户准备了一系列可以在用户态调用的函数：`syscall_*()`系列函数，他们每个函数都对应一个可以通过系统调用完成的任务，用户通过调用这些函数，向操作系统传达信息，**我要使用系统调用完成某个功能**

- 进入内核态，调用汇编函数`msyscall`，这个函数位于`user/lib/syscall_wrap.S`文件中

在刚提到的`syscall_*`函数中，存在一个`msyscall`函数，在这个汇编写成的`msyscall`中，才出现了真正的`syscall`汇编指令，也就是在这里正式地进入了内核态，准备使用异常处理程序解决`syscall`

- 用户态保存，`exc_gen_entry()`函数，这个函数位于`kern/entry.S`文件中

在异常处理程序中，调用 SAVE_ALL 保存当前现场为内核栈中的 Trapframe ，并将当前使用栈转换为内核栈，完成上下文的保存和栈的转换

- 接收请求并分类，`handle_sys`

通过异常处理程序判断为系统调用后，转入 handler 这个函数创建了一个size = TFSIZE + 8 大小的栈帧，其中 TFSIZE 是 Trapframe 的大小，额外的 8 字节是为了保存 a0 寄存器和函数返回地址 ra，这里之所以要保存 a0 是因为 handler 函数继续调用 do_syscall 函数的时候，要借助 a0 寄存器把 Trapframe 传入，因此需要保存 a0 寄存器修改前的值

而本质上，其实 ra 不保存也没有影响，这里除了 a0 以外，还额外多加了 4 字节，主要是为了遵循 ABI 守则，调用 syscall_handler 函数时满足 MIPS 调用约定的栈空间需要

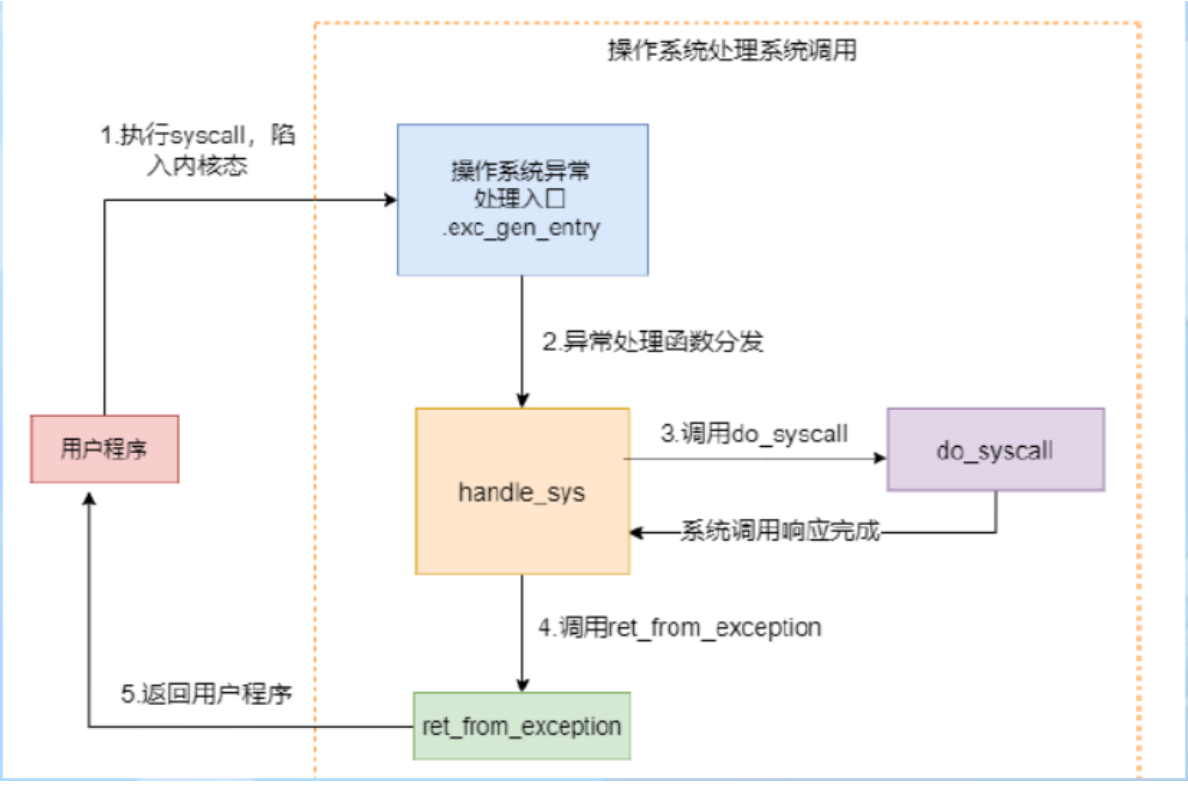
- 参数预处理调用处理函数，`do_syscall`

在这个函数中，我们通过分析用户传入的信息（`syscall_*`的类型和用户现场）来响应系统调用，这里主要是借助传入参数 tf 对应五个参数和一个系统调用号，告诉内核它要执行的是哪个内核系统调用函数，然后跳转到那个函数进行真正的调用

- 响应完毕，返回用户态，`ret_from_exception`

在从`do_syscall`跳出并执行完对应处理函数，并返回至`handle_sys`后，最后会和其他异常一样，执行`ret_from_exception`，还原现场，返回用户态，整个系统调用的过程结束

图示：



SAVE_ALL 宏的作用：在保存用户态现场时 sp 减去了一个 Trapframe 结构体的空间大小，此时我们将用户进程现场保存在内核栈中范围为 [sp, sp + sizeof(TrapFrame)) 的这一空间范围内，说白了就是内核栈的栈顶

系统调用号：用户进程告诉内核具体要执行的是哪一个系统调用的函数，相当于是不同系统调用之间区分的唯一标识，作为用户态函数`mysyscall`执行的**第一个**参数传入

`mysyscall`一共有 6 个入参，这些参数就是系统调用的时候，用户进程传给内核的参数，其中前 4 个参数用寄存器 a0 到 a3 存储，只需要在栈中留出相应空间，不需要真正参数压栈，后 2 个参数需要实际压栈，存在 sp+20 和 sp+16 位置上

区分函数跳转：陷入内核的操作并不是从一个函数跳转到另一个函数，这里的函数跳转指的是函数调用、参数压栈、返回这样的控制流，而陷入内核是通过执行 CPU 的特权指令（如 syscall 指令），完成从用户态到内核态的模式切换与跳转，属于**硬件级别**的控制流转移，实现以下三个转换：

1. 执行`syscall`等指令后，程序使用堆栈自动从用户栈切换到内核栈，切换的同时，内核会把用户进程的运行现场保存到内核空间，通过`SAVE_ALL`函数实现，随后栈指针会指向保存好的`Trapframe`，从而获取用户传来的参数
2. 程序计数器跳转到`handle_sys()`位置，借助上一步设置好的栈指针获取系统调用的参数
3. 发生 CPU 模式切换，硬件主导，从用户态切换进入内核态

具体实现

发起系统调用 - syscall_*

上面提到，可由用户调用、距离内核态最近的用户函数就是这一系列`syscall_*`函数，它们作为用户可调用的函数，位于`user/lib/syscall_lib.c`文件中，现在来看看它们的具体内容：

```
// 为节省空间仅保留了部分函数，反正差不多里面所有函数都长这样()
void syscall_putchar(int ch) {
    msyscall(SYS_putchar, ch);
}

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

int syscall_mem_map(u_int srcid, void *srcva, u_int dstid, void *dstva,
u_int perm) {
    return msyscall(SYS_mem_map, srcid, srcva, dstid, dstva, perm);
}
```

可以看到，它们都只调用了**不同参数**的 `msyscall` 函数，这里还需要注意，`syscall_putchar`、`syscall_yield`、`syscall_panic` 这几个函数没有以"return"的方式调用 `msyscall` ，因为他们不是 `void` 的，就是 `noreturn` 的，其中每个函数的第一个参数是系统调用号，相互之间一定是不同的

转入内核态 - msyscall

这个在用户态执行的最后一个函数，位于 `user/lib/syscall_wrap.S` 中，这个函数很简单：

```
LEAF(msyscall)
        // Just use 'syscall' instruction and return.
        /* Exercise 4.1: Your code here. */
        syscall
        jr ra
END(msyscall)
```

这个函数实际上充当了用户态、内核态的转接口：执行 `syscall` 进入内核态，从调用返回后执行 `jr ra` ，十分简洁，分工明确，我们可能注意到了不同的 `msyscall` 调用可能有不同的参数数量，他们都被保存在堆栈中为函数创造的 stack frame 空间中，与 sp 相邻，下一步，内核就会接收到由硬件产生的 8 号异常（系统调用异常），通过处在 `kern/entry.S` 的异常分发程序 `exc_gen_entry` 跳转到 handler 函数： `do_syscall`

分发系统调用 - do_syscall

首先需要注意，在跳转至 `do_syscall` 前，我们在异常分发程序中向内核栈（SAVE_ALL）压入了用户态 trapframe 的信息，随后又通过 move 指令把 a0 寄存器复制成了 trapframe 的地址（SAVE_ALL 中 move sp 的值） 所以 `do_syscall` 在调用时就会自动地带有一个参数，它就是存放在 a0 寄存器中的用户态 `trackframe` **指针**（为什么是指针？因为传入的 sp 的值实际上指向了存放 tf 的地址）

```
/* Overview:
  函数功能 : 调用 syscall_table 中索引为 sysno 的函数(sysno  存在 tf 上下文对应的
regs[4] 中)，并从内核栈(Trapframe)获取用户进程上下文作为参数传给它
  除去第一个参数作为系统调用号以外，其余参数依次存储在寄存器 $a1, $a2, $a3，以及栈中
[$sp + 16B] 和 [$sp + 20B]，最多有 5 个参数
 */
void do_syscall(struct Trapframe *tf) {
// 定义一个函数指针 func，指向一个函数，这个指针最多接收五个 u_int 类型的参数，并且返回一
个 int 类型的结果
        int (*func)(u_int, u_int, u_int, u_int, u_int);
// 从 Trapframe 中的 regs[4] 中取出系统调用信号 sysno，注意 : 在 MIPS 架构里，
$a0(即 regs[4])寄存器里通常传递第一个参数，这里规定用它传递系统调用号
        int sysno = tf->regs[4];
```

```
        if (sysno < 0 || sysno >= MAX_SYSNO) {
                tf->regs[2] = -E_NO_SYS;// $v0 <=> regs[2]，记录错误返回值
                return;
        }

// 更新 EPC，异常程序计数器，当前 epc 指向 syscall 指令的地址，在执行完系统调用后，应该跳过这一条 syscall，因此要对其 +4 操作
        /* Exercise 4.2: Your code here. (1/4) */
        tf->cp0_epc += 4;

// 取出函数指针，以 sysno 为索引，从 syscall_table 中拿到对应的系统调用函数，赋值给 func
        /* Exercise 4.2: Your code here. (2/4) */
        func = syscall_table[sysno];

// 取前三个参数 : MIPS 规定，第1~第3个参数分别放在 $a1($5)、$a2($6)、$a3($7)，依次取出，存入 arg1、arg2、arg3
        u_int arg1 = tf->regs[5];
        u_int arg2 = tf->regs[6];
        u_int arg3 = tf->regs[7];

// 声明后面要用到的第 4 5 个参数，并且从栈上获取，MIPS调用约定 : 如果参数超过3个，剩下的从栈上取,栈指针是 $sp，即 regs[29]，$sp + 16 : 第4个参数; $sp + 20 : 第5个参数
        u_int arg4, arg5;
        /* Exercise 4.2: Your code here. (3/4) */
        arg4 = *((u_int *)tf->regs[29] + 4);
        arg5 = *((u_int *)tf->regs[29] + 5);

// 调用系统调用函数，并且返回结果，把返回值写入 $v0，也就是 regs[2] 号寄存器中
        /* Exercise 4.2: Your code here. (4/4) */
        tf->regs[2] = func(arg1, arg2, arg3, arg4, arg5);
}
```

do_syscall 函数（这个函数本质是由 handle_sys 跳转得到的）

这里需要注意，用户态在 msyscall 中通过内核栈传给 handle_sys 函数的确实是 6 个参数，但是第一个参数是 sysno 系统调用号，通过它索引到正确的 sys_* 函数，这个函数本身需要的其实只有后 5 个参数，系统调用号不需要再接着传下去

因为内核中实现的函数都有**返回值**，且这个返回值是要返回给用户态的，因此我们要把这个返回值存入 Trapframe 结构体中的 v0 寄存器，在恢复用户进程上下文环境以后，用户程序从该寄存器读出 msyscall 的返回值

用户栈与内核栈

- 这里 sysno 取自 a0 寄存器，那前面 handler 中 tf 地址也是保存在 a0 寄存器，二者不会互相覆盖吗?
- 答案是不会，因为存 sysno 的 a0 是用户栈的 a0 寄存器，但是存 tf 的 a0 是内核栈的 a0 寄存器

首先，当发生系统调用的时候，用户态寄存器会通过`SAVE ALL`宏保存到 Trapframe 结构体中，Trapframe 结构体存在内核栈上，此时，用户态传给系统调用函数的 sysno 本来是存在用户栈的 a0 上，此时就会随之保存到 Trapframe 的 regs [4] 中

在发生系统调用的时候，CPU 会把用户栈切换成内核栈，也就是说此时程序使用的是内核栈，Trapframe 就存储在内核栈顶，然后内核调用`do_syscall`函数，把 Trapframe 作为参数传递，此时使用的是内核栈的 a0 寄存器，所以本质上用的是两个栈上的 a0 寄存器，不会有影响，一个是**内核栈上的**`$a0`，用于给`do_syscall(tf)`传参数；一个是**用户态中**`$a0`**寄存器的旧值**，它是 syscall 号（sysno），被保存在`trapframe->regs[4]`中（因为`$a0`是寄存器 4）

换言之，在内核态的情况下，如果直接调用 a0 ~ a4 或者 sp，所取得的都是内核栈对应的寄存器，如果此时我们还想获取用户栈对应的寄存器，只能通过 tf->regs 进行获取，其中用户栈 sp 存在 29 号

系统调用函数表，通过系统调用号，索引到不同的系统调用函数

```
/*
   系统调用表 : 函数指针数组，存放的是各个系统调用函数的指针
*/
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,
};
```

这里用到陷阱帧 tf 中的寄存器与通用寄存器之间的对应关系：

```
struct Trapframe {
    uint32_t regs[32];   // 通用寄存器 r0 ~ r31
    uint32_t hi, lo;     // 乘除法寄存器
    uint32_t cp0_status;
    uint32_t cp0_epc;    // 异常发生时的 PC
    ...
};
```

寄存器号	名称	用途
`$0`	`$zero`	常量 0（只读）
`$1`	`$at`	汇编器保留
`$2`	`$v0`	返回值1 / 系统调用号
`$3`	`$v1`	返回值2
`$4`	`$a0`	第 1 个参数（**sysno**）
`$5`	`$a1`	第 2 个参数
`$6`	`$a2`	第 3 个参数
`$7`	`$a3`	第 4 个参数
`$8`–`$15`	`$t0`–`$t7`	临时变量
`$16`–`$23`	`$s0`–`$s7`	被保留变量
`$24`	`$t8`	临时变量
`$25`	`$t9`	临时变量
`$26`	`$k0`	内核保留
`$27`	`$k1`	内核保留
`$28`	`$gp`	全局指针
`$29`	`$sp`	栈指针
`$30`	`$fp` / `$s8`	帧指针
`$31`	`$ra`	返回地址

Trapframe

定义和组成：Trapframe 被称为**陷阱帧**，当处理器从用户态进入内核态的时候（比如发生系统调用、中断或异常时），CPU 会自动把当前进程运行上下文保存起来，陷阱帧就是用来保存当前进程上下文（有关寄存器状态）的一个结构体，其结构如下

```
struct Trapframe {
        /* 32 个通用寄存器 */
        unsigned long regs[32];

        /* 特殊寄存器 */
        unsigned long cp0_status;     // CPU 的全局状态
        unsigned long hi;             // 高位寄存器
        unsigned long lo;             // 低位寄存器
        unsigned long cp0_badvaddr;   // 错误地址寄存器
        unsigned long cp0_cause;      // 中断异常原因记录寄存器
        unsigned long cp0_epc;        // 异常发生时的 PC
};
```

1. regs [32]：陷入内核的时候保存所有通用寄存器的值，回到用户态的时候恢复这些寄存器的值

2. cp0_status：协处理器0，用来控制中断和异常行为，控制中断是否可以被接收，标志当前是否在异常处理中
3. cp0_badvaddr：错误地址寄存器，只在**访存错误的时候**，用来记录引起中断或异常的那个虚拟地址，发生 `TLB Miss`，`TLB Invalid`，`Address Error` 等异常时，记录访问地址，内核据此判断是哪段内存出错，进行补页或终止进程
4. cp0_cause：异常原因寄存器，每次中断或异常时由 CPU 设置，内核据此决定如何处理异常（比如分派到哪个 handler）
5. cp0_epc：异常程序计数器，保存的是异常发生时的 PC 值**（即下一条指令）**，异常发生的时候，CPU 自动保存 EPC，中断返回的时候，使用 `eret` 恢复 EPC，让程序能够接着继续执行，因此它指向的应该是异常指令的下一条指令

需要区分 badvaddr 和 epc 中两个值，前者只在访存异常（TLB Mod，TLB Miss 等）时才需要被记录，而后者只要发生异常的时候都会被记录

注意 EPC 也有特殊情况，如果内核想要让用户自己处理某个异常，那么陷入内核的时候，内核代码会额外手动设置 EPC 为用户指定的异常处理程序的入口，这样程序回到用户态的时候会先跳转到那个异常处理程序，处理完异常之后再返回正常代码

存在形式：Trapframe 本质上是内核栈上的一段空间，对于用户进程而言，每当有异常 / 中断发生的时候，程序会动态创建一段结构体，保存在每个进程的内核栈**栈顶**，这段空间就是 Trapframe，换言之，Trapframe 是临时保存现场的结构，当处理完当前异常 / 中断，这段空间就应该被相应释放

作用：陷阱帧一般作为 Trapframe * 指针的形式作为函数入参，在汇编代码层面就是 a0 寄存器传参，其有四个作用

1. 陷入内核的时候，当前进程上下文存入陷阱帧（SAVE_ALL）
2. 给目标函数传递进程上下文**参数**，主要是 a0 ~ a4 寄存器，也就是 tf->regs [4] ~ tf->regs [7]，和 sp 栈指针对应栈上的值，也就是 tf->regs [29] 这个指针
3. 获取目标函数调用的**返回值**，返回值存入 v0 寄存器，也就是 tf->regs [2] 中，这里的返回值不能在内核态就 return，而是必须存入 v0 寄存器，这个寄存器会被传回用户态恢复到用户寄存器，进而在用户态才会真正 return
4. 回到用户态，陷阱帧中保存的上下文恢复给进程寄存器

env->tf

定义：需要与 Trapframe 进行辨析的，是进程控制块中的 tf 指针，它是一个 Trapframe * 类型的指针，用于指向当前进程 env 最近一次发生中断 / 异常所生成的陷阱帧 Trapframe 中的内容，在大部分情况下，我们可以粗略认为这两者是一样的，其实 env->tf 设置的主要目的就是，当我们切换了不同进程的时候，内核栈随之发生改变，我们就无法获得之前那个进程原有的陷阱帧了，因此需要找一个指针，使得这段上下文在不同的环境下都能被访问

有关概念

用户态 和 内核态（也称用户模式 和内核模式）：它们是 CPU 运行的两种状态，在 MOS 操作系统实验使用的仿真 4Kc CPU 中，该状态由 CP0 SR 寄存器中UM(User Mode) 位和 EXL(Exception Level) 位的值标志

用户空间 和 内核空间：它们是虚拟内存（进程的地址空间）中的两部分区域，MOS 中的用户空间包括 kuseg，而内核空间主要包括 kseg0 和 kseg1，每个进程的用户空间通常通过**页表**映射到不同的物理页，而内核空间则直接映射到固定的物理页以及外部硬件设备，CPU 在内核态下可以访问任何内存区域，对物理内存等硬件设备有完整的控制权，而在用户态下则只能访问用户空间

（用户）进程 和内核：进程是资源分配与调度的基本单位，拥有独立的地址空间，而**内核**负责管理系统资源和调度进程，使进程能够并发运行，与前两对概念不同，进程和内核并不是对立的存在，可以认为内核是存在于所有进程地址空间中的一段代码

系统调用函数

上面我们实现了系统调用的主要流程，最后系统会在内核态下执行`sys_*`函数，现在需要我们具体实现几个系统**调用函数**

envid2env

这里需要做一个区分：如果我们想得到当前进程的 env，我们特判传入 envid == 0，这只是为了避免一次额外遍历 envs 数组的过程，并不意味着 curenv 的 envid 就是 0，需要注意这两个条件相互不是充要的，不过这可以说明不存在 envid == 0 的进程，因此才可以用 0 这个编号进行特判

```
/* Overview:
  函数功能：根据给定的进程编号 envid 得到它对应的进程控制块 env，如果 envid == 0 则默认得到运行中进程 curenv; 如果 checkperm 非零，则要求指定的 env 必须是当前环境(curenv)或其直接子环境
 */
int envid2env(u_int envid, struct Env **penv, int checkperm) {
        struct Env *e;

// 如果传入的 envid == 0，则表示调用者想获取当前环境
        /* Exercise 4.3: Your code here. (1/2) */
        if (envid == 0) {
                *penv = curenv;
                return 0;
        }
// 根据 envid 查找进程结构体指针，用 ENVX(envid) 提取 envid 的低位索引部分，从 envs 数组中取出对应的进程结构体
        e = &envs[ENVX(envid)];

// 检查环境是否合法，即未被释放且 ID 匹配，即使数组索引对了(ENVX)，也要验证 env_id 是否匹配，避免访问错误/过期的环境结构，这是经典的 ID 冗余验证机制
        if (e->env_status == ENV_FREE || e->env_id != envid) {
                return -E_BAD_ENV;
        }

// 根据 checkperm 判断是否有权限操作这个环境，如果设置了 checkperm 标志，则说明需要检查调用者是否有权限操作目标环境，如果目标环境既不是自己(e != curenv)，也不是自己的直接子环境(e->env_parent_id != curenv->env_id)，则说明权限不足，这是因为为了安全起见，防止一个环境非法修改 / 终止其他无关进程，只允许操作自己或自己 fork 出来的子环境
        /* Exercise 4.3: Your code here. (2/2) */
        if (checkperm && e != curenv && e->env_parent_id != curenv->env_id)
{
                return -E_BAD_ENV;
        }

// 把找到的合法环境返回给调用者
        *penv = e;
        return 0;
}
```

sys_mem_alloc

sys_mem_alloc： 通过这个系统调用，用户程序可以给进程的某个虚拟地址**显式**地分配一段物理内存空间，可以认为是在任意进程下为 va 分配一个物理页 p，函数体本身是对 page_insert 函数的封装，但是因为用户态的 page_insert 只能作用于当前进程，没法给其他 envid 对应的进程申请映射，所以需要能操作所有进程的内核来实现这个功能

```c
/* Overview:
  函数功能 : 分配一个物理页，并将虚拟地址 va 在 env_id 所对应的进程的页目录下映射到该物理页，权限位为 perm
  1. 如果 va 已经映射了一个物理页，那么原有映射会被取消
  2. 使用 envid2env 函数的时候，必须设置 checkperm 参数，确保目标环境是调用者自身或者调用者的子进程

  后置条件 : 成功时返回 0; 如果 envid2env 检查权限失败，va 为非法地址，底层函数调用失败，各自返回对应错误码
 */
int sys_mem_alloc(u_int envid, u_int va, u_int perm) {
        struct Env *env;
        struct Page *pp;

// 检查虚拟地址是否合法
        /* Exercise 4.4: Your code here. (1/3) */

        if (is_illegal_va(va)) {
                return -E_INVAL;
        }

// 根据 envid 得到其对应的进程结构体指针 env
        /* Exercise 4.4: Your code here. (2/3) */
        try(envid2env(envid, &env, 1));

// 分配一个物理页 pp
        /* Exercise 4.4: Your code here. (3/3) */
        try(page_alloc(&pp));

// 在 env 的页目录 pgdir 之下，完成物理页 pp 和虚拟地址 va 的映射
        return page_insert(env->env_pgdir, env->env_asid, pp, va, perm);
}
```

sys_mem_map

sys_mem_map 函数：把源进程地址空间中，某个虚拟地址映射到的物理页，和目标进程中某个虚拟地址建立映射，此时两个进程的两个虚拟地址会共享**同一个物理页**的映射，这个函数在本章主要有两个作用：

1. 在 COW 机制下把父进程的某个页复制到子进程
2. 自映射复制修改某进程某个页的权限位

```c
/* Overview:
  函数功能 : 找到在 srcid 对应的进程的页目录中，映射到虚拟地址 srcva 的物理页，把 dstid 对应的进程中的虚拟地址 dstva 映射到这个物理页上，设置权限位为 perm

  后置条件 : 成功时返回 0; 如果 envid2env 转化失败，或者两个虚拟地址 srcva 和 dstva 是非法地址，或者 srcva 在 srcid 中没有映射，或者底层函数调用失败，返回对应的错误码
 */
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;

// 检查虚拟地址 srcva 和 dstva 的合法性
```

```
        /* Exercise 4.5: Your code here. (1/4) */

        if (is_illegal_va(srcva) || is_illegal_va(dstva)) {
                return -E_INVAL;
        }

// 根据 srcid 找到对应的进程结构体指针 srcenv
        /* Exercise 4.5: Your code here. (2/4) */
        try(envid2env(srcid, &srcenv, 1));

// 根据 dstid 找到对应的进程结构体指针 dstenv
        /* Exercise 4.5: Your code here. (3/4) */
        try(envid2env(dstid, &dstenv, 1));

// 在 srcenv 对应的页目录 pgdir 下，找到与 srcva 进行映射的那个物理页 pp
        /* Exercise 4.5: Your code here. (4/4) */
        pp = page_lookup(srcenv->env_pgdir, srcva, NULL);
        if (pp == NULL) {
                return -E_INVAL;
        }

// 在 dstenv 对应的页目录下，在 dstva 与刚刚找到的物理页 pp 之间建立映射
        return page_insert(dstenv->env_pgdir, dstenv->env_asid, pp, dstva,
perm);
}
```

sys_mem_umap

sys_mem_umap：取消掉进程的某个虚拟地址和它的物理地址之间的映射关系

```
/* Overview:
  函数功能 : 取消 envid 对应的进程的页目录下，虚拟地址 va 所映射的所有物理页
  后置条件 : 同上
 */
int sys_mem_unmap(u_int envid, u_int va) {
        struct Env *e;

// 检查虚拟地址 va 是否合法
        /* Exercise 4.6: Your code here. (1/2) */
        if (is_illegal_va(va)) {
                return -E_INVAL;
        }

// 根据 env_id 找到对应的结构体 env
        /* Exercise 4.6: Your code here. (2/2) */

        try(envid2env(envid, &e, 1));

// 取消进程 e 的页目录之下，虚拟地址 va 和其所有映射的物理页的映射关系
        page_remove(e->env_pgdir, e->env_asid, va);
        return 0;
}
```

sys_yield

sys_yield：实现用户进程主动对 CPU 的放弃，从而调度其他进程，注意这个函数是 noreturn 的，也就是会直接开始运行下一个进程块

```
/* Overview:
  函数功能 : 放弃当前进程(curenv)所剩余的 CPU 时间片
  后置条件 : 调度另一个环境 env 来运行
  __attribute__ : GCC 编译器扩展，告诉编译器该函数不会返回到调用它的地方，也就是说只要调用了 sys_yield 就不会继续执行程序后面的代码
 */
// void sys_yield(void);
void __attribute__((noreturn)) sys_yield(void) {
// 通过 schedule 函数中传 yield == 1 直接放弃剩余时间片即可
        /* Exercise 4.7: Your code here. */
        schedule(1);
}
```

sys_set_trapframe

虽然没有直接考这个函数的填写，但是后来发现还是比较重要的一个函数，它使用的位置是在进行写时复制异常处理分配新物理页的时候，在 cow_entry 函数中，最后确保能够直接跳转回正常用户态代码，而不是再次返回到 do_tlb_mod 的内核函数中

```
/* Overview:
  函数功能 : 把 envid 对应进程的上下文信息 trap_frame 设置为传入的参数 tf，以便于它下次被调度的时候上下文能够恢复成为 tf 对应的值
 */
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));
/*
  这里需要区分修改的是不是当前进程的上下文，如果是 curenv，它当前运行的上下文其实没存在 env->tf 中，而是存在内核栈的 Trapframe 中(也就是栈顶)，因此我们这时候修改它的 tf，相当于没修改到点上，它返回的时候还是从内核栈拿信息，需要直接修改内核栈；如果是 curenv 之外的别的进程，当后面他们被调度的时候会首先从 tf 中恢复上下文信息，因此直接修改他们存在进程块中的上下文即可
  返回值层面，如果修改的是当前进程，那么修改后的返回值要立刻返回，所以返回的是 tf->regs[0] 中的值，如果修改的不是当前进程，那么那个被修改过的进程被调度的时候，自己会返回这个设好的返回值(就在 tf->regs[2] 中)，那么这里只需要先返回个 0，相当于是垫了一个无意义的返回值
*/
        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;
        }
}
```

总结

进程和内核之间**并非对立**关系：在内核处理进程发起的系统调用时，我们并没有切换地址空间（页目录地址），也不需要将进程上下文（Trapframe）保存到进程控制块中，只是切换到内核态下，执行了一些内核代码，可以说，处理系统调用时的内核仍然是代表**当前进程**的（当前这个进程不会下处理机），这也是系统调用、TLB 缺失等同步异常与时钟中断等异步异常的本质区别

IPC

IPC 定义

定义：IPC 实现的是进程间某些数据的传递，这是微内核最重要的机制之一，IPC 是实现 fork，文件系统服务，管道，shell 的基础，它的实现需要遵循以下三点要求：

1. IPC 要能使得两个进程之间可以通信
2. IPC 需要通过系统调用实现
3. IPC 与进程数据，页面等信息有关

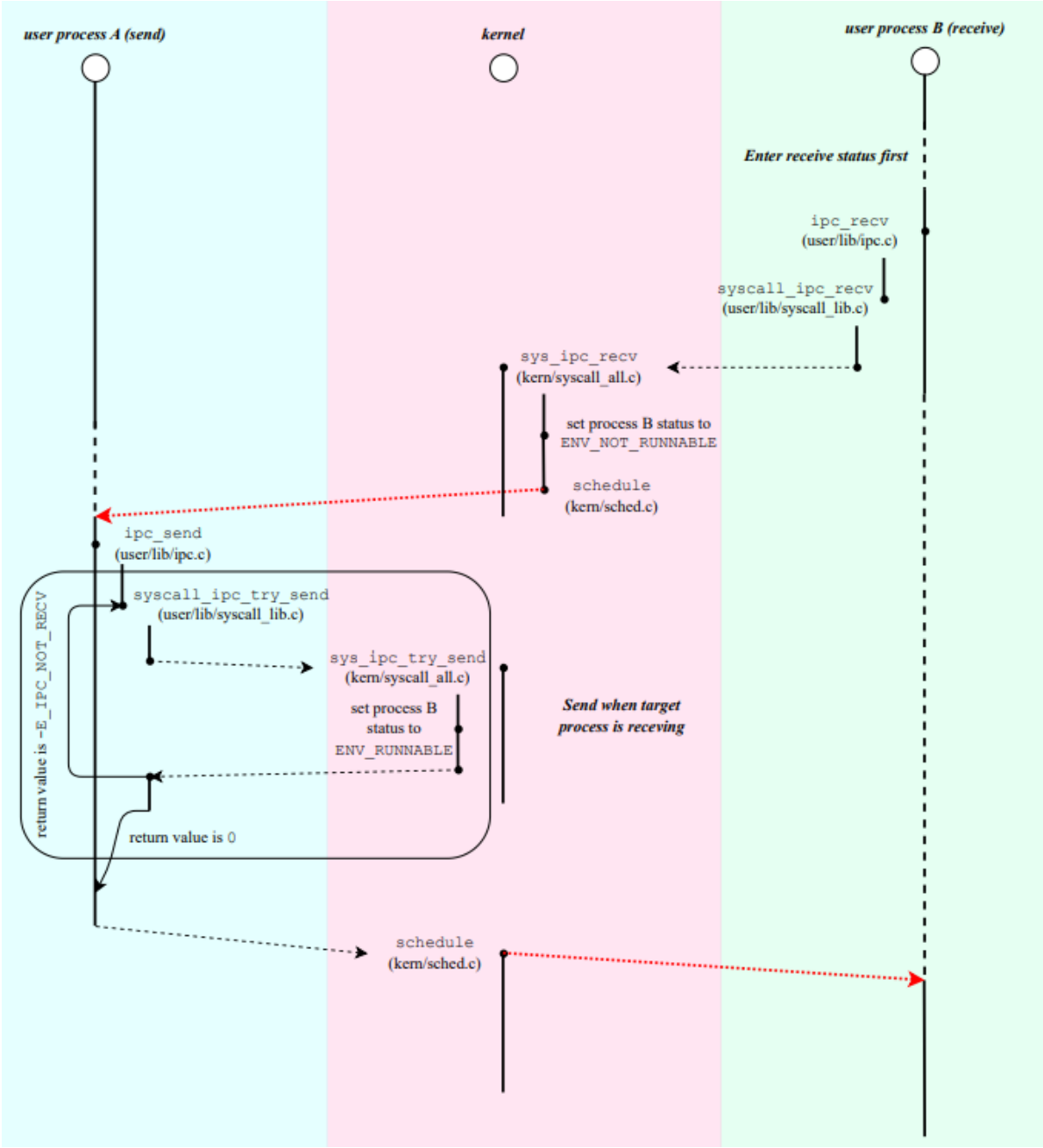
设计思路：因为每个进程都有独立空间，进程间信息传递本质上就是把一个地址空间中的内容传递给另一个地址空间，同时又因为所有进程都共享一个**内核空间**（kseg0），我们就可以借助内核空间进行信息传递：发送方进程可以将数据以系统调用的形式存放在进程控制块中，接收方进程同样以系统调用的方式在进程控制块中找到对应的数据，读取并返回

因此进程控制块中涉及了以下成员：

```
struct Env {
// lab 4 IPC
 u_int env_ipc_value;     // 进程传递的具体数值
 u_int env_ipc_from;      // 发送方的进程 id
 u_int env_ipc_recving;   // 进程是否在等待接收数据(判阻塞)
 u_int env_ipc_dstva;     // 接收到的页面需要与自身哪个虚拟页面完成映射，这个参数是在 sys_ipc_recv 中设置的
 u_int env_ipc_perm;      // 传送的页面权限位设置
};
```

进程间通信流程

接收者和发送者的顺序是：某个进程因为需要其他进程发送的信息而**先**开始等待接收，因此进入阻塞态；与此同时每个进程都**不断尝试**向外发送信息，如果它发送信息的接收者进程当前没有进入等待接收的状态，则这次信息发送会失败，如果对方也在等待接收信息，那么就进行一次正常的进程间通信，具体流程如下，我们记 B 为等待消息的进程，A 为发送消息的进程



等待方

1. B 进程首先进入**接收状态**，B 进程调用用户空间的`ipc_recv()`函数（user / lib / ipc.c），进而在该函数中调用`syscall_ipc_recv()`函数（user / lib / syscall_lib.c），由此进入内核态，调用内核函数`sys_ipc_recv()`（kern / syscall_alloc.c）
2. 在内核态系统调用中，B 进程完成两件事：设置 B 进程状态为 ENV_NOT_RUNNABLE，从调度队列中移除 B，表示 B 在等 A 的消息；调用 schedule 切换到其他进程运行

发送方

1. A 进程会不断调用用户空间的`ipc_send()`函数（user / lib / ipc.c），在该函数中会尝试调用`syscall_ipc_try_send()`函数，进而由用户态进入内核态，在内核态中会调用真正的系统调用函数`sys_ipc_try_send`（kern / syscall_alloc.c）
2. 在内核态系统调用中，会首先检查信息发送**目标进程**，也就是 B 进程是否在等待接收，如果不是，则返回错误码不进行通信，如果是，就把消息传给 B 进程，并且设置 B 进程的状态变回 ENV_RUNNABLE，相当于把 B 进程变成就绪态，放回调度队列即可

重新调度

接下来执行正常调度流程，A 进程可能下处理机，B 可能重新恢复运行

sys_ipc_recv

sys_ipc_recv：该函数用来接收信息

```
/* Overview:
  函数功能 : 等待从其他进程 env 接收一条消息，具体形式是一个值，如果 dstva 不是 0，还会包括一个页映射，当前环境 curenv 会被阻塞，直到有消息发送过来
 */
int sys_ipc_recv(u_int dstva) {
// 检查 dstva 既不是 0，也不能非法
        if (dstva != 0 && is_illegal_va(dstva)) {
                return -E_INVAL;
        }

// 设置 curenv 的 env_ipc_recving 字段为 1，表示进入消息接收状态
        /* Exercise 4.8: Your code here. (1/8) */

        curenv->env_ipc_recving = 1;

// 设置 curenv 的 env_ipc_dstva 字段为 dstva，保存目标虚拟地址，即如果当前进程收到一个页，就映射到这个位置
        /* Exercise 4.8: Your code here. (2/8) */

        curenv->env_ipc_dstva = dstva;

// 阻塞当前进程，设置 status 为 ENV_NOT_RUNNABLE，并且从调度队列 env_sched_list 中将其移除
        /* Exercise 4.8: Your code here. (3/8) */

        curenv->env_status = ENV_NOT_RUNNABLE;
        TAILQ_REMOVE(&env_sched_list, curenv, env_sched_link);

// 挂起当前进程，直到它再次被唤醒，此时需要给他一个返回值，等他被调度的时候，从陷阱帧中取出上下文的时候，这个返回值就会被返回
        ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0;
        schedule(1);
}
```

系统调度的返回值：内核系统调用的返回值，是如何被返回到用户态的?

一般的内核系统调用函数，也就是在执行系统调用的时候不会阻塞当前进程的那些函数，会直接设一个返回值 return，这个 return 的返回值会传给调用这个`sys_*`的上层的`do_syscall`函数，由`do_syscall`函数控制存入当前 Trapframe 的对应寄存器中，进而由硬件恢复陷阱帧到用户态寄存器，所以正常情况下，内核系统调用的返回值**必须**通过 return 返回给 do_syscall 函数，不允许系统调用自己设置 Trapframe 的值；而本例的 sys_ipc_recv 函数则是特殊情况，因为在执行完这个函数之后，当前进程直接被阻塞了，不会走 do_syscall 函数返回，而是会等到它被调度的时候，直接由硬件控制恢复到用户态，所以此时的返回值不会被从 do_syscall 获得，因此它只能手动把自己要返回的值存入 Trapframe 中，等到后续对用户态恢复

同时我们又知道，Trapframe 是存在内核栈的栈顶位置的，又因为栈顶指针 KSTACKTOP 是一个虚指针，因此该位置 -1 得到的就是 Trapframe 存放的起始位置

因此，如果是在内核态下，我们就得到了两种访问 Trapframe 的方式：

1. 一个是像`sys_ipc_recv()`函数那样，借助内核栈指针直接访问内核栈获取 Trapframe
2. 一个是像`do_syscall()`函数那样，直接在陷入内核的时候，就把 Trapframe 作为参数传进去直接使用

sys_ipc_try_send

sys_ipc_try_send：该函数用来发送信息

```
/* Overview:
  函数功能 ： 尝试向目标进程 envid 发送一个 value 值，如果 srcva 不为 0，还包括一个页面，这个物理页就是 srcva 在 curenv 中映射到的那个物理页
  后置条件 ： 成功时返回 0，并且目标环境做如下更新
    1. env_ipc_recvint 设置为 0，阻止再次接收消息
    2. env_ipc_from 设置为发送者的 value
    3. env_ipc_value 设置为被发送的 value
    4. 如果 env_status 设置为 ENV_RUNNABLE，则恢复其运行
    5. 如果 srcva 非 NULL，则把 srcva 在发送者的 pgdir 中映射到的物理页，在映射者的页目录上和 env_ipc_dstva 之间进行映射
    如果目标不是在等待 IPC 消息的时候收到消息，则返回错误码
 */
int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm) {
        struct Env *e;
        struct Page *p;

// 确保 srcva 既不是 0，同时也不能非法
        /* Exercise 4.8: Your code here. (4/8) */

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

// 根据 envid 得到对应的进程结构体指针 e
        /* Exercise 4.8: Your code here. (5/8) */
        try(envid2env(envid, &e, 0));

// 检查当前进程是否在等待信息，即是否处在 env_ipc_recving == 1 的状态
        /* Exercise 4.8: Your code here. (6/8) */
        if (e->env_ipc_recving == 0) {
                return -E_IPC_NOT_RECV;
        }

// 设置目标进程 e 的有关字段信息
        e->env_ipc_value = value;
        e->env_ipc_from = curenv->env_id;
        e->env_ipc_perm = PTE_V | perm;
        e->env_ipc_recving = 0;

// 重新设置接收者进程为 ENV_RUNNABLE，同时把他插入回调度队列 env_sched_list 中
        /* Exercise 4.8: Your code here. (7/8) */
        e->env_status = ENV_RUNNABLE;
        TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);

// 如果 srcva 非 0，则在 curenv 中找到 srcva 映射到的物理页，然后把这个物理页在 e 中和 e 的 env_ipc_dstva 字段中的虚拟地址之间建立映射
        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));
}
  // 内核在这里使用了忙等待，会阻塞所有进程
int sys_cgetc(void) {
        int ch;
        while ((ch = scancharc()) == 0) {
        }
        return ch;
}
```

这里 sys_ipc_try_send 这个函数，是向 env_id 对应的进程发送信息，这个信息的**发送方**是 curenv，也就是认为信息的发出者都是当前运行中的进程，其中接收方的 dstva 是其在设置 recving 为 1 的时候同步设置的，而我们之前一直说的如果 srcva 不为 0 的时候还会同时传过来一个物理页，这个物理页其实就是 curenv 中 srcva 所映射的那个物理页，我们并没有显式地传递它

注意：由于在我们的用户程序中，会大量使用 srcva 为 0 的调用来表示只传 value 值，而不需要传递物理页面，换句话说，当 srcva 不为 0 时，我们才建立两个进程的页面映射关系，这其实对应了进程间通信的两种形式，如果只是简单的传输 value，借助**内核空间**中进程控制块的空间即可；如果还要额外传输一个物理页，则需要借助**共享内存空间**传递

这也侧面说明了，所有进程是共享同一份内核空间的

这两个函数是内核函数，我们在用户态下会设置对应的用户态函数来调用它们，对应的用户态函数在 `user/lib/ipc.c` 文件中

ipc_send

用户态发送函数 - ipc_send

```
// Hint: use syscall_yield() to be CPU-friendly.
void ipc_send(u_int whom, u_int val, const void *srcva, u_int perm) {
    int r;
    /* Step 1: 持续查询是否能进行传输 */
    while ((r = syscall_ipc_try_send(whom, val, srcva, perm)) == -E_IPC_NOT_RECV) {
    /* Step 2: 如果只因为对方未就绪，自身也停止这个 while 导致的忙等待，交出 CPU */
        syscall_yield();
    }
    /* Step 3: 如果 r != 0，说明退出 while 循环的原因是 -E_INVAL，说明出现错误 */
    user_assert(r == 0);
}
```

这里使用的 `syscall_yield` 非常巧妙：如果对方进程未就绪，那在这个时间片中也不可能转换为就绪状态（本时间片一直是本进程运行，只要对方进程上不了处理机，就肯定没法就绪），那为了避免当前时间片忙等，索性直接归还 CPU，等下一次调度到自己之后再查

ipc_recv

用户态接收函数 - ipc_recv

```
// Hint: use env to discover the value and who sent it.
u_int ipc_recv(u_int *whom, void *dstva, u_int *perm) {
    /* Step 1: 直接进系统调用，因为 recv 是先执行的，不用等 */
    /* 需要注意进程在下一条语句中调用了 schedule，传输完毕后才跳出 */
    int r = syscall_ipc_recv(dstva);
    /* Step 2: 检查返回值，错误值则 panic */
```

```
    if (r != 0) {
        user_panic("syscall_ipc_recv err: %d", r);
    }
    /* Step 3: 返回发送者的 env_id */
    if (whom) {
        *whom = env->env_ipc_from;
    }
    /* Step 4: 返回映射页权限位 */
    if (perm) {
        *perm = env->env_ipc_perm;
    }
    /* Step 5: 返回传输的单个 int */
    return env->env_ipc_value;
}
```

fork

内核通过 env_create 创建进程，直接创建即可；而用户进程如果想作为父进程创建一个子进程，则需要基于系统调用实现，这里我们借助 fork 机制来实现，由此可知，fork 函数是一个用户态级别的函数

实现原理

一个进程在调用 fork 函数以后，会从此分成两个进程进行：

1. 新产生的进程称为**子进程**，子进程开始运行时的大部分上下文状态和父进程相同，包括程序和 fork 运行时的现场，通用寄存器，程序计数器；但是子进程中 fork 调用的返回值为 0
2. 原有的进程称为**父进程**，父进程中 fork 调用的返回值为子进程的 env_id，如果 fork 失败，则不会创建子进程，返回一个小于 0 的错误值

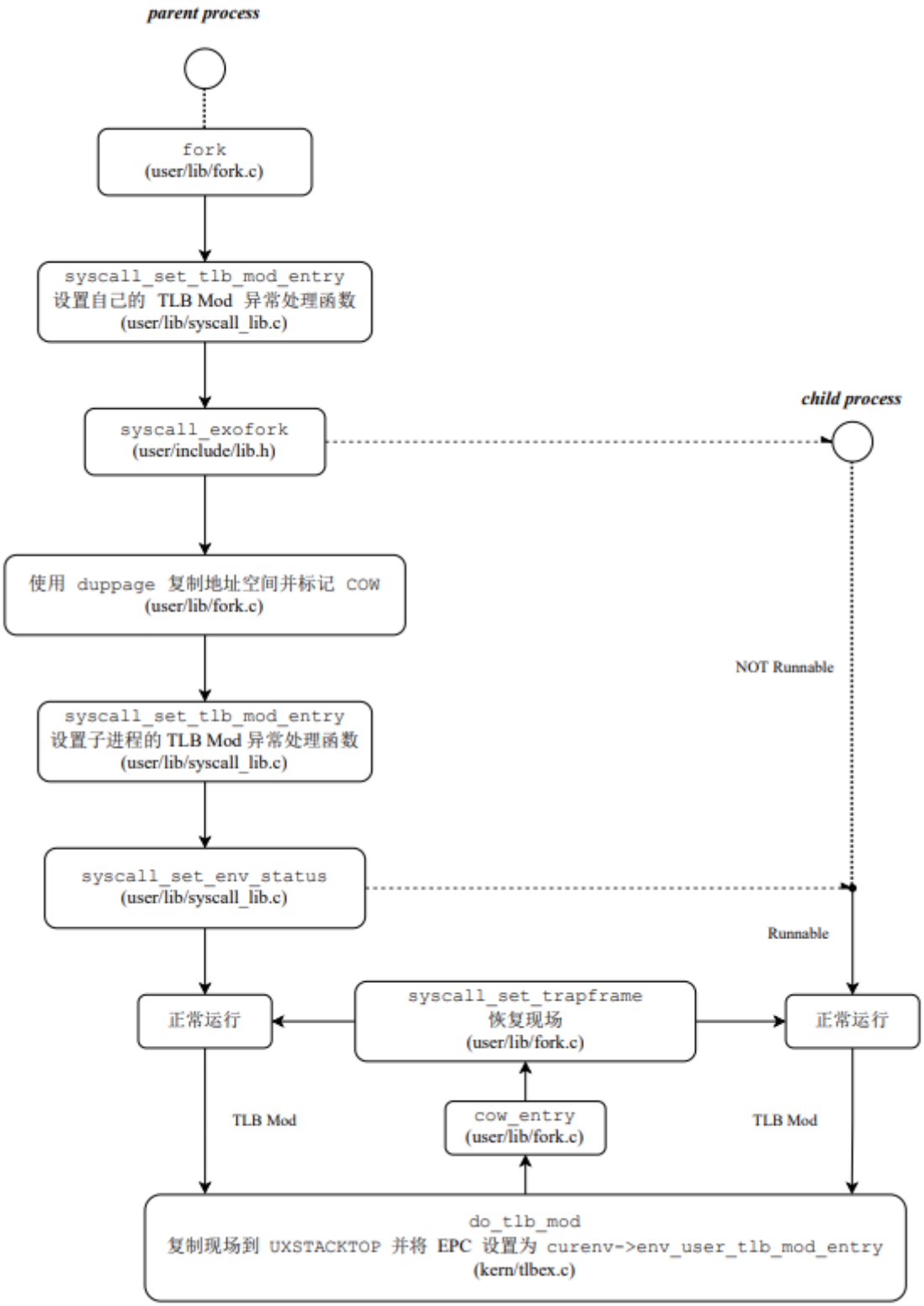
fork 在父子进程中产生不同返回值这一特性，让我们能够在代码中调用 fork 后判断当前在父进程还是子进程中，以执行不同的后续逻辑，也使父进程能够与子进程进行通信

fork 创建出和父进程运行上下文几乎一致的子进程，主要是为了方便父子进程之间的通信行为，与之相对应的另一种创建方式是 exec 系列的系统调用，他会是的进程抛弃现有的程序和运行现场，从头开始执行新的程序

若在进程中调用 exec，进程的地址空间（以及在内存中持有的所有数据）都将被重置，新程序的二进制镜像将被加载到其代码段，从而让一个从头运行的全新进程取而代之，fork 的一种常见应用就被称作 fork-exec，指在 fork 出的子进程中调用 exec，从而在创建出的新进程中运行另一个程序

进程创建流程

fork 执行的具体流程如下：



1. 父进程发起 fork 调用，调用 fork() 函数
2. 父进程设置 TLB Mod 异常处理入口，通过系统调用 syscall_set_tlb_mod_entry(handler) 设置当发生 TLB 修改异常（写某些只读页）的时候跳转的用户态处理函数，而 TLB Mod 异常是 COW 的关键触发机制
3. 调用 syscall_exofork() 创建子进程，子进程的初始状态为 ENV_NOT_RUNNABLE，在内核中复制当前继承的基本结构（Trapframe）但是不复制页表，返回新建子进程的 ID
4. 调用 duppage() 函数复制地址空间，并标记为 COW，把父进程的每个页都映射到子进程，并标记为**只读**和**写时复制**，父进程本身这些页页标记为**只读**和**写时复制**，这会使得后续的写操作触发 TLB Mod 异常，注意要先标记子进程后标记父进程
5. 子进程也设置 TLB Mod 异常处理入口，同样是调用 syscall_set_tlb_mod_entry(child)，通过分析可知，进程创建过程中，父子进程都需要设置 TLB Mod 异常处理入口，确保子进程也可以正确处理写时复制

6. 设置子进程为可运行状态
7. 子进程开始正常运行，首先由调度器调度子进程上处理机，调用 syscall_set_trapframe() 函数恢复子进程运行的上下文
8. 当子进程或者父进程执行写操作的时候，触发 TLB Mod 异常，此时进程在用户态跳转到 cow_entry() 函数，这个函数用来实现检测写异常并且分配新的物理页，拷贝内容，设置权限位等功能

注意：cow_entry() 这个函数是用户态下的函数，所执行的操作总体上看也都是在用户态下完成的，但是其中涉及到内存分配、页面映射都函数还是会进行系统调用，进入内核态执行，可以认为用户态负责逻辑判断、页内容复制等，系统调用负责真正的页表操作，这样可以**将页错误（TLB Mod 异常）的一部分处理逻辑下放到用户空间**，以便灵活实现 Copy-On-Write（COW）等机制，同时减轻内核负担，尽管 cow_entry() 是用户态的函数，但是因为其内部也涉及到系统调用与内核交互，因此内核捕捉到 TLB Mod 异常以后，仍然会保护异常现场，也就是切换到**用户异常栈**，并且控制 EPC 指向 syscall_set_tlb_mod_entry() 这个函数的入口

可以认为 cow_entry() 是由内核指导，总体上在用户态下完成的操作，具体实现也涉及到系统调用和内核交互

TLB Mod 异常

在这里，TLB Mod 异常是触发 COW 机制的入口，它可以看成是一种特殊的页异常

定义：当一个虚拟地址页被写入的时候，TLB 中的映射存在，但是该页的映射中缺少**可写**的标志位，就会触发 TLB Mod 异常，这里的 Mod 是 Modification 的意思

如果 TLB 中的映射不存在，在我们的体系结构下，会首先查页表写入 TLB，再查 TLB，那么如果对应页面没有可写权限的话，即便它一开始不在 TLB 中，最终还是会进入 TLB，并且触发 TLB 异常

作用：TLB Mod 异常是实现 COW 的关键机制，利用它可以拦截首次非法的写入行为，触发 TLB Mod 异常的下一步就是进入 cow_entry() 函数

触发：

1. 进程写入一个 COW 页面
2. 页表中该页无 `PTE_W`，触发 `TLB Mod` 异常

进入内核态（注意这里进入内核态是为了让内核控制切换用户异常栈和 EPC 的设置，最后还要返回用户态执行 cow_entry() 函数）：

1. CPU 捕获异常，保存上下文到 trapframe
2. 检查是否注册了用户页异常处理函数 `cow_entry`
3. 切换到**用户异常栈**，跳转到用户空间的 `cow_entry()` 函数，这里是为了借助用户异常栈的公用空间进行页复制

在用户态处理（执行 `cow_entry()`）：

1. 判断触发异常的是 COW 页面；
2. 调用系统调用：
3. `syscall_mem_alloc()` 在一个临时页分配新物理页
4. `memcpy()` 把原页内容拷贝过去；
5. `syscall_mem_map()` 把新页重新映射到原虚拟地址，添加 `PTE_W`
6. `syscall_mem_unmap()` 清理临时页

最后，返回用户态，继续执行写入指令

用户异常栈

用户栈、用户异常栈、内核栈关系：

名称	用途	所属态	特点
用户栈	运行用户程序时函数调用、局部变量等	用户态	普通应用使用的主栈
用户异常栈	用户态下处理异常（如 page fault）用栈	用户态	仅特殊异常处理时使用
内核栈	CPU陷入内核态后用于执行内核函数	内核态	每个进程内核态唯一栈

三者相对地址关系：

```
用户进程地址空间：
+-------------------------+  高地址
| 用户异常栈 UXSTACKTOP  | ← page fault 会跳这里
+-------------------------+
|       用户栈            | ← 普通程序运行时用这里
+-------------------------+
|      heap, data, text   |
+-------------------------+  低地址

内核空间（每进程一份）：
+-------------------------+  高地址
|       内核栈（kernel）  | ← 进入内核态后栈切换
+-------------------------+
```

用户异常栈：某些操作系统（如 JOS、Xv6 用户态异常机制）允许**在用户态**处理异常（比如 `page fault`，`syscall` 等），这时候，不能再用普通用户栈，因为异常可能就是在普通用户栈上触发的，所以，系统会提前为用户注册一个**专门的异常栈**来保存异常时的上下文

用户异常栈虽然还是属于用户地址空间，但是从普通用户栈到用户异常栈的切换，还是需要在内核态下完成

用户注册异常入口

定义：什么是用户注册的异常入口？用户告诉内核，如果我发生某种异常（比如 TLB Mod 异常），请不要由你内核来直接处理，而是跳转到我指定的某段代码去处理

具体来讲，在操作系统中，**异常**（如页错误、TLB miss、TLB Mod 等）默认都是由内核处理的，但有些操作系统支持让用户程序参与异常处理，这样做的好处是：

1. 用户可以自己决定如何应对一些“可恢复”的异常
2. 增强用户程序的灵活性，比如实现用户态的“线程库”、“垃圾回收器”、“页面调度”等

如何设置：操作系统通常会提供一个系统调用，如：

```
int sys_set_user_tlb_mod_entry(uint32_t entry_point);
```

用户程序在初始化时调用它，把自己写好的一个异常处理函数**地址**（也就是系统调用函数表中存入的地址）告诉内核，内核会把这个地址记录在 `curenv->env_user_tlb_mod_entry` 这个字段中，下次再处理 tlb mod 异常的时候，内核就知道这个异常应该交给用户自己处理

需要实现的函数：

fork

fork：父进程首先调用 fork 函数，开始创建子进程，本身是一个在**用户**级别实现的函数，但是会通过若干系统调用来实现自身功能

```
/* Overview:
  函数功能 : 用户级 fork 的实现，创建一个子进程并复制父进程的地址空间，设置父进程和子进程的 TLB 修改异常入口为 cow_entry
  后置条件 : 子进程的 env 结构体被正确地设置
 */
int fork(void) {
        u_int child;// 存储子进程的 envid
        u_int i;

// 设置父进程的 TLB 修改异常处理入口，首先检查当前进程是否已经设置 COW 处理函数，如果没有设置，就调用 syscall_set_tlb_mod_entry 设置异常处理入口为 cow_entry
        if (env->env_user_tlb_mod_entry != (u_int)cow_entry) {
                try(syscall_set_tlb_mod_entry(0, cow_entry));
        }

// 创建子进程环境，首先调用 syscall_exofork 创建子进程环境但是不调度，如果子进程编号为 0，则说明子进程就是当前运行中进程，设置 env 指针指向子进程的环境结构体，ENVX 用来从 envid 中提取索引
        // correct value.
        child = syscall_exofork();
// 注意 : 其实下面这个分支不会立刻执行，因为子进程创建好之后不会立刻被调度，直到它被调度之后，才会为他恢复它的进程上下文，让他跳回到父进程 fork 函数本行的位置，接着执行下面的分支
        if (child == 0) {
                env = envs + ENVX(syscall_getenvid());
                return 0;
        }

// 复制地址空间到子进程，外层循环遍历页目录项(PDX)，vpd[i] & PTE_V 检查月录像是否有效，PDX(UXSTACKTOP)为用户异常栈顶的页目录索引
        /* Exercise 4.15: Your code here. (1/2) */
        for (i = 0/*其实这里从 PDX(UTEXT) 段开始即可*/; i < PDX(UXSTACKTOP); i++) {
                if (vpd[i] & PTE_V) {
                // 内层循环遍历页表项，计算虚拟地址 va，跳过用户栈以上的地址 USYACKTOP，vpt[VPN(va)] & PTE_V 检查页表项是否有效
                        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) {
                                // 调用 duppage 复制有效页面到子进程
                                        duppage(child, VPN(va));
                                }
                        }
                }
        }
// 设置子进程状态，syscall_set_tlb_mod_entry 设置子进程的 COW 处理函数为 cow_entry，syscall_set_env_status 设置子进程状态为可运行 ENV_RUNNABLE
        /* Exercise 4.15: Your code here. (2/2) */
```

```
        syscall_set_tlb_mod_entry(child, cow_entry);
        syscall_set_env_status(child, ENV_RUNNABLE);
        return child;
}
```

并行进程

我们观察到 fork 函数实现的时候，会有两次返回值，第一次返回 0，第二次返回子进程的 envid，这是由 fork 函数的性质所决定的

在父进程执行 fork 函数以后，子进程和父进程会并发执行，fork 函数会在两个进程中同时继续执行，并且各自有一个返回值，此时返回 0 的就是子进程，返回子进程 envid 的就是父进程，换言之，我们通过判断 syscall_exofork 的返回值来决定 fork 的返回值以及后续动作

既然 fork 的目的是使得父子进程处于几乎相同的运行状态，我们可以认为在返回用户态时，父子进程应该经历了同样的恢复运行现场的过程，只不过对于父进程是**从系统调用中返回**时恢复现场（sys_exofork 是系统调用），而对于子进程则是在进程被**调度**时恢复现场（子进程刚创建出来的时候是 NOT_RUNNABLE），在现场恢复后，父子进程都会从内核返回到 msyscall 函数中，而它们的现场中存储的返回值（即 $v0 寄存器的值）是不同的，这一返回值随后再被返回到 syscall_exofork 和 fork 函数，使 fork 函数也能根据最终的**返回值**区分两者，fork 函数是用户态函数，直接 return 返回值即可

函数指针

这里涉及到函数的底层定义，在 C 语言中，函数名在编译的时候会被编译器处理成指向函数体的一个指针，因此这里如果我们把函数名 cow_entry 当做参数传递，实际上传递的是 cow_entry 这个函数的地址

env 指针更新

在用户态进程下，会设置一个 env 全局指针，这个指针负责在用户程序进入运行入口的时候，指向当前进程的控制块，对于子进程而言，特获得了一个与父进程不同的进程控制块，但是 fork 出来之后，它的 env 指针还没来得及更新，因此我们需要更新子进程的 env 指针，也就是

```
env = envs + ENVX(syscall_getenvid());
```

因为子进程执行 sys_exofork 返回值是 0，则它必须通过系统调用，也就是执行 syscall_getenvid 来获得自己的进程 envid，然后根据这个 envid 在 envs 数组中索引到自己的那个进程块，正确修改 env 指针的值

地址空间复制

我们详细解析一下父进程的哪些地址空间，以及他们是怎么复制给子进程的

```
// 外层循环 : 页目录项遍历，遍历范围是 0 到 UXSTACKTOP(用户异常栈顶)，也就是说只需要复制用户正常栈部分的地址即可，异常栈不归 fork 管
for (i = 0; i < PDX(UXSTACKTOP); i++) {
  // 借助 vpd 数组，用户进程可以索引到页目录项，检验它的有效位
  if (vpd[i] & PTE_V) {
    // PAGE_SIZE 是一个页目录页的大小，sizeof(Pte) 是一个页目录项的大小，二者相除就能得到一个页目录页中含有多少个页目录项，每个页目录项都对应一张页表
    for (u_int j = 0; j < PAGE_SIZE / sizeof(Pte); j++) {
      // 根据页目录索引 i 和页表索引 j，得到虚拟地址 va (lab2 考过)，(i << 10 + j) << 12 即可
      u_long va = (i * (PAGE_SIZE / sizeof(Pte)) + j) << PGSHIFT;
      // 如果虚拟地址越界就直接 break
      if (va >= USTACKTOP) {
        break;
      }
```

```
      // 借助 vpt 数组，用户进程可以索引到页表项，其中 VPN(va) 就是 j，检验它的有效位
      if (vpt[VPN(va)] & PTE_V) {
        // 如果这个页表项有效，就调用 duppage 把他拷贝到子进程
        duppage(child, VPN(va));
      }
    }
  }
}
```

思考：对于为什么第一层的循环的是`for (i = 0; i < PDX(UXSTACKTOP); i++)`，这里可以类比页表自映射的思路来理解，假设我们有一个页目录表，这个页目录表中的每一个页目录项从下到上会依次按页映射用户空间中的地址，第 0 个页目录就映射第 0 个虚拟页，那么第 PDX(va) 个页目录项就相应映射第 va 个虚拟页（这里默认 va 是页对齐的），也就是满足 1 : 1024 的映射关系（也就是自下到上按顺序，每个 4B 大小的页目录项，都对应一个 1024 × 4B 大小的页的映射），首先我们知道，这里复制父进程到子进程只需要复制用户正常栈部分的地址即可，也就是复制 0 到 UXSTACKTOP 范围内的地址，那么对应的页目录项范围就应该是 0 到 PDX(UXSTACKTOP)，换言之我们只需要遍历这么多的页目录项，就可以把这段虚拟地址映射过去，而 vpd 数组中每个页目录项又是从 0 开始索引的（实际上他是在用户空间的 UVPT 段，只是 vpd 数组帮我们做好了映射），因此我们只需要索引 0 到 PDE(UXSTACKTOP) 范围的页目录项即可

fork 实现流程

在父进程的 fork 函数是一个**用户函数**中实现了以下几个功能，下面调用的四个函数都是内核实现的：

1. 调用 sys_set_tlb_mod_entry 设置父进程的 TLB Mod 异常处理入口
2. 调用 sys_exofork 真正创建出一个子进程
3. 拷贝父进程的进程上下文给子进程
4. 调用 duppage 函数拷贝父进程的页面映射关系给子进程
5. 调用 sys_set_tlb_mod_entry 设置子进程的 TLB Mod 异常处理入口
6. 调用 sts_set_env_status 设置子进程状态为 ENV_NOT_RUNNABLE

sys_exofork

sys_exofork：具体执行一个创建子进程的过程，单纯的创建（进程块的分配，块内参数的设置），不涉及父子进程间地址或信息的拷贝

```
/* Overview:
  函数功能 : 为当前进程 curenv 分配一个新的子环境
  后置条件 : 新环境的 env_tf 从内核栈中复制，除了 $v0 设置为 0，表示子环境的返回值；新环境的 env_status 设置为 ENV_NOT_RUNNABLE；新环境的 env_pri 从当前环境 curenv 复制
  返回值 : 子进程的 env_id
 */
int sys_exofork(void) {
        struct Env *e;

// 调用 env_alloc 函数为 curenv 分配一个子环境 e，e 的父进程 id 为 curenv 的 env_id
        /* Exercise 4.9: Your code here. (1/4) */
        try(env_alloc(&e, curenv->env_id));

// 从内核栈中取出子进程的进程上下文，这里获得的其实是父进程的上下文，系统调用的时候会父进程上下文会压入内核栈
        /* Exercise 4.9: Your code here. (2/4) */

        e->env_tf = *((struct Trapframe *)KSTACKTOP - 1);

// 手动额外设置子进程的 $v0 寄存器为 0，确保它回到用户态的时候，返回值记录的是 0
```

```
        /* Exercise 4.9: Your code here. (3/4) */

        e->env_tf.regs[2] = 0;

// 设置子进程的的 env_status 和 env_pri 字段
        /* Exercise 4.9: Your code here. (4/4) */

        e->env_status = ENV_NOT_RUNNABLE;
        e->env_pri = curenv->env_pri;

        return e->env_id;
}
```

sys_exofork() 函数在为子进程分配进程控制块以后，还要用父进程（此时父进程就是 curenv）的某些信息作为模版来填充这个控制块，也就是复制父进程的一些基本信息到子进程：

1. 运行现场：父进程的进程上下文给到子进程
2. 返回值：需要手动设置子进程中，Trapframe 对应的 v0 寄存器为 0，使得它返回用户态的时候，返回值是 0，用来和父进程区分（父进程的返回值是子进程 id）
3. 进程状态：不能让子进程在父进程的 syscall_exofork 返回后就直接被调度，因为这时候它还没有做好充分的准备，所以我们需要将它的状态设为 ENV_NOT_RUNNABLE 且避免它被加入调度队列
4. 其他信息：还设了一个 env->pri，这里和父进程相同即可

在父进程执行这个函数的过程中，会把子进程的 v0 寄存器设置为 0；随后父进程从系统调用中返回并恢复现场，`syscall_exofork` 的返回值为子进程 env_id，而子进程**并没有执行这个函数**，只是在被调度时才首次开始运行，恢复进程控制块中存放的 trapFrame，其中的 EPC 也令进程从 fork 函数中的 `syscall_exofork` 结束后运行，这时存放于 v0 寄存器的返回值是 0，因此子进程走 `child == 0` 这个分支结束，**看起来好像子进程执行了**这个函数并返回了 0，这就是 `syscall_exofork` 实现两个不同返回值的过程

duppage

duppage：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，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

```
/* Overview:
  函数功能 : 把 curenv 中的某个虚拟页(用 vpn 虚拟页号表示)，映射到子进程 envid 中，根据权限做共享或者写时复制处理
  后置条件 : 如果虚拟页 vpn 有 PTE_D 且没有 PTE_LIBRAYR，则我们原始的虚拟页和 envid 进程新映射的虚拟页都应该被标记为 PTE_COW，且不带走 TTE_D，同时保留其他权限位；如果不满足上述条件，则 envid 中新映射的虚拟页应具有与原始虚拟页完全相同的权限
  1.PTE_LIBRARY : 表示页面在父进程与子进程之间共享
  2.syscall_* 有关的系统调用函数中，如果传入 envid == 0，则表示 curenv，因为内核 envid2env 会把 0 这个编号转化为 curenv
  本函数用于把页面复制到子进程，envid 表示子进程编号，vpn 是要复制的虚拟页号
 */
static void duppage(u_int envid, u_int vpn) {
        int r;// 系统调用返回值
        u_int addr;// 计算出的虚拟地址
        u_int perm;// 存储页面权限标志

// 获取页面的权限，首先把虚拟页号 vpn 左移 12 位转化为虚拟地址，接着通过页表数组 vpt 索引 vpn 获得对应的页表项，进而提取出低 12 位存入 perm 中
        /* Exercise 4.10: Your code here. (1/2) */
        addr = vpn << PGSHIFT;
```

```
        perm = vpt[vpn] & ((1 << PGSHIFT) - 1);
/* 处理可写页面的 COW 映射，如果满足以下三个条件中任意一个，则可以直接把原权限映射到子进程

    1. (perm & PTE_D) == 0: 页面不可写
    2. (perm & PTE_LIBRARY): 页面是共享库页面
    3. (perm & PTE_COW): 页面已经是COW页面
*/
        /* Exercise 4.10: Your code here. (2/2) */
        // 直接从 curenv 映射到 envid，保持原权限
        if ((perm & PTE_D) == 0 || (perm & PTE_LIBRARY) || (perm & PTE_COW)) {
                if ((r = syscall_mem_map(0, (void *)addr, envid, (void *)addr, perm)) < 0) {
                        user_panic("user panic mem map error: %d", r);
                }
          // 以下是 COW 机制的关键 : 对于可写非共享页面，先映射到子进程，权限改为 (perm & ~PTE_D) | PTE_COW，也就是移除可写标志(PTE_D)，添加 COW 标志(PTE_COW)，再重新映射到当前进程的页面，同样修改权限，确保父子进程都使用 COW 机制，注意，这里一定是要先修改子进程，再修改父进程
        } else {
                if ((r = syscall_mem_map(0, (void *)addr, envid, (void *)addr,
                                         (perm & ~PTE_D) | PTE_COW)) < 0) {
                        user_panic("user panic mem map error: %d", r);
                }
                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);
                }
        }
}
```

这里我们虽然传入的是虚拟页号 vpn，主要是为了所引导对应的那个页表项，vpt [vpn]，而根据这个 vpn 得到虚拟地址 addr 也很容易实现，直接左移 12 位即可

注意：在 duppage 函数中，需要根据有不同权限位的页使用不同方式进行处理：

1. **只读页面**：对于不具有 PTE_D 权限位的页面，按照相同权限（只读）映射给子进程即可
2. **写时复制页面**：即具有 PTE_COW 权限位的页面，这类页面是之前的 fork 时 duppage 的结果，且在本次 fork 前必然未被写入过
3. **共享页面**：即具有 PTE_LIBRARY 权限位的页面。这类页面需要保持共享可写的状态，即在父子进程中映射到相同的物理页，使对其进行修改的结果相互可见
4. **可写页面**：即具有 PTE_D 权限位，但是不符合以上特殊情况的页面，这类页面需要在父进程和子进程的页表项中都是用 PTE_COW 权限位进行保护

其中对于父进程中可写但非共享的页面，此时就是 COW 需要特殊处理的情况，需要在父进程（id 为 0）和子进程（id 为 envid）中同时删除它们的可写位，再添加上 COW 标志，这就是 COW 实现的关键

注意：这里添加 COW 标志的时候，一定是先添加子页面，再添加父页面，这是因为子进程被创建出来以后，不会立即运行，则不会出现在子进程和父进程添加权限位之间，子进程被写入的情况，而父进程则不能保证这一点，假设我们先修改父进程的权限位，修改完以后父进程写这个页，此时发现这个页是 COW 页，则父进程重新获得一个没有 COW 标记的新页，然后再给子进程修改权限位，此时子进程以为自己和父进程共享旧页，它共享的其实是父进程的新页，那么以后再修改父进程的时候，父进程认为自己可以修改，那么子进程共享的这个页也就被随之修改了，这就会出错

sys_set_env_status

sys_set_env_status：用来在子进程刚创建好的时候，设置其运行状态为 ENV_NOT_RUNNABLE，之所以不能让子进程立刻被调度，是因为还要为为写时复制特性的页写入异常处理做好准备

```
/* Overview:
  函数功能 : 把 env_id 对应进程的 env_status 设置为指定的 status，并且更新调度列表 env_sched_list
  后置条件 : 同上
 */
int sys_set_env_status(u_int envid, u_int status) {
        struct Env *env;

// status 必须只能是 ENV_RUNNABLE 和 ENV_NOT_RUNNABLE
        /* Exercise 4.14: Your code here. (1/3) */

        if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE) {
                return -E_INVAL;
        }

// 根据 env_id 得到对应的 env 进程结构体指针
        /* Exercise 4.14: Your code here. (2/3) */

        try(envid2env(envid, &env, 1));

// 根据 env 对应 status 的变化，更新调度队列 env_sched_list
        /* 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);
        }

// 设置 env 的 status
        env->env_status = status;

// 如果修改的是当前正在运行中的进程，就让他主动放弃处理机，触发一次进程切换调度
        if (env == curenv) {
                schedule(1);
        }
        return 0;
}
```

sys_set_tlb_mod_entry

sys_set_tlb_mod_entry：在父进程创建子进程的时候，分别设置父子进程处理 TLB Mod 异常的入口，因为将来他们二者都可能因为 COW 触发这个异常，所以需要提前设好异常处理入口，确保陷入内核以后能够回到用户态来处理这个异常

```
/* Overview:
  函数功能 : 把指定 env_id 进程对应的用户态 TLB Mod 异常处理程序入口设置为 func 这个函数
  后置条件 : 把 env_id 对应环境的 TLB Mod 异常处理程序入口设置为 func，成功时返回 0，错误时返回对应错误码
```

```
  入参含义 : func 表示用户空间中异常处理函数的入口地址(函数的虚拟地址)
 */
int sys_set_tlb_mod_entry(u_int envid, u_int func) {
        struct Env *env;

// 由 env_id 得到它对应的 Env 结构体指针
        /* Exercise 4.12: Your code here. (1/2) */
        try(envid2env(envid, &env, 1));

// 设置这个 Env 结构体的 env_user_tlb_mod_entry 字段为 func
        /* Exercise 4.12: Your code here. (2/2) */

        env->env_user_tlb_mod_entry = func;

        return 0;
}
```

通过上面这五个函数，fork 是主函数，其余三个是被 fork 调用的子函数，就可以由父进程创建出一个子进程

do_tlb_mod

do_tlb_mod：内核态将一部分内核操作暴露给用户态，这样可以减轻内核压力，维护一定的功能简洁性，TLB Mod 异常处理就是一个特殊的异常，它是在用户态被处理的

```
/* Overview:
  函数功能 : 在内核态下，处理 TLB Mod 异常的函数(do_tlb_mod 这个函数本身是在内核态下执行的)
  内核允许用户程序在用户态下处理 TLB Mod 异常，因此我们把异常上下文 tf 拷贝到用户异常栈(UXSTACK)中，并且把 EPC 设置为用户注册的异常入口
  前置条件 : env_user_mod_entry 是用户通过调用 sys_set_user_tlb_mod_entry 注册的用户空间异常入口地址，用户自定义的入口应该处理这个 TLB Mod 异常，并且恢复上下文
 */
void do_tlb_mod(struct Trapframe *tf) {
        struct Trapframe tmp_tf = *tf;// tf 是陷入内核时保存的寄存器与状态信息的结构体(Trapframe),将原始的陷入上下文 tf 拷贝到一个临时变量 tmp_tf 中,这样做的目的是保存当前状态，以便之后将它压入用户栈

// 检查寄存器 $sp(即 regs[29]，栈顶指针)是否已经处于用户异常栈(UXSTACK)中，如果不在 UXSTACK 范围(即当前在普通用户栈或非法区域)，则重置 $sp 为 UXSTACK 栈顶，这是为了确保接下来的压栈操作安全地进行在异常栈中，不会覆盖普通用户栈
        if (tf->regs[29] < USTACKTOP || tf->regs[29] >= UXSTACKTOP) {
                tf->regs[29] = UXSTACKTOP;
        }
        // 为新的 Trapframe 留出空间，相当于用户栈先向下扩展腾地，压栈的时候在加回来;接着把临时保存的 Trapframe 拷贝到用户异常栈中，以便用户态处理程序能恢复
        tf->regs[29] -= sizeof(struct Trapframe);
        *(struct Trapframe *)tf->regs[29] = tmp_tf;
// 引发 TLB Mod 异常的虚拟地址此时被存在 tf 的 cp0_badvaddr 中，我们要查找这个地址对应的页表项(可能是 lazy use)
        Pte *pte;
        page_lookup(cur_pgdir, tf->cp0_badvaddr, &pte);
        // 先检查当前环境是否注册了用户态的 TLB Mod 异常处理入口，如果有，就将 Trapframe 的地址传递给用户态异常处理程序，通过 $a0(寄存器4)，用户处理函数就可以用它来读取 Trapframe 并在处理后恢复原状态，之后再调整栈顶弹出 Trapframe 的空间
        if (curenv->env_user_tlb_mod_entry) {
                tf->regs[4] = tf->regs[29];
```

```
                tf->regs[29] -= sizeof(tf->regs[4]);
// 关键一步：将 EPC(异常返回地址)设置为用户注册的 TLB Mod 处理函数地址，下次返回用户态时会跳转到该地址，执行用户自定义的异常处理逻辑
                /* Exercise 4.11: Your code here. */
                tf->cp0_epc = curenv->env_user_tlb_mod_entry;
          // 如果用户没有注册，直接报错
        } else {
                panic("TLB Mod but no user handler registered");
        }
}
#endif
```

注意：内核态下的这个 do_tlb_mod 函数其实并没有真正解决 tlb 异常，而是相当于起了一个中转的作用，因为引发异常的时候，进程上下文默认你是传给内核栈的，而我们要在用户态下处理这个异常，就需要把传入内核栈的进程上下文接着传给用户异常栈，并且在内核态结束要返回 EPC 的时候，把 EPF 设置成用户自己注册的 tlb 异常处理入口，这样程序就会通过内核态，进一步跳转到用户态的 tlb 异常处理程序中，再完成真正的异常处理

cow_entry

cow_entry：我们刚刚说，do_tlb_mod 接着跳转到用户态进行 TLB Mod 处理，它跳转到的函数就是 cow_entry 函数，这个函数才是真正负责处理异常的函数，是在用户态实现的

```
/* Overview:
  函数功能 : 当某个进程尝试写一个共享的页面的时候，触发异常，把这个页面复制一份可写的副本
  前置条件 : va 是导致 TLB 修改异常的地址
  后置条件 : 如果 va 不是一个写时复制的页面，则触发 user_panic; 否则，该处理程序应该在相同地址映射一个私有的可写副本
  本函数用来处理写时复制的异常(COW)
 */
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf) {
// tf 中存指向陷阱帧的指针，包含 CPU 状态和异常信息，其中 cp0_badvaddr 是陷阱帧中存储的导致异常的虚拟地址
        u_int va = tf->cp0_badvaddr;
        u_int perm;

// 查找导致异常的虚拟地址 va 的权限，其中 VPN 用来提取 va 的虚拟页号，vpt 是存储页表的数组，可以通过虚拟页号索引获取页表项，PTE_FLAGS 用于提取页表项中的权限标志位，检查权限中是否包含 PTE_COW 标志，如果没有就出发 panic
        /* Exercise 4.13: Your code here. (1/6) */
        perm = PTE_FLAGS(vpt[VPN(va)]);
        if ((perm & PTE_COW) == 0) {
                user_panic("PTE_COW not found, va=%08x, perm=%08x", va, perm);
        }
// 修改权限标志，移除写时复制标志，添加可写标志，表示页面已经被修改
        /* Exercise 4.13: Your code here. (2/6) */
        perm = (perm & ~PTE_COW) | PTE_D;
// 使用系统调用在 UCOW 地址分配新页面，参数 0 表示当前进程，使用修改后的权限 perm
        /* Exercise 4.13: Your code here. (3/6) */
        syscall_mem_alloc(0, (void *)UCOW, perm);
// 把原始页面的内容复制到 UCOW 地址处的新页面，这里需要把虚拟地址 va 向下对其到页面边界，然后把原始页面整个页面的内容复制到 UCOW 处的新页面
        /* Exercise 4.13: Your code here. (4/6) */
        memcpy((void *)UCOW, (void *)ROUNDDOWN(va, PAGE_SIZE), PAGE_SIZE);
// 把 UCOW 处的页面映射到原始地址 va
        /* Exercise 4.13: Your code here. (5/6) */
```

```
        syscall_mem_map(0, (void *)UCOW, 0, (void *)va, perm);
// 取消 UCOW 地址处的临时映射
        /* Exercise 4.13: Your code here. (6/6) */
        syscall_mem_unmap(0, (void *)UCOW);
// 返回到异常发生时的代码，恢复陷阱帧，返回到异常发生时的状态
        int r = syscall_set_trapframe(0, tf);
        user_panic("syscall_set_trapframe returned %d", r);
}
```

该函数有许多要注意的地方：

1. 这是一个**用户态**的函数，因此它不能直接调用内核系统调用函数，而是必须借助用户态下的接口，比如只能调用 syscall_mem_alloc，syscall_mem_map 这些函数，而不能直接调用 sys_mem_alloc，sys_mem_map
2. cow_entry 函数是无返回值的，它被 do_tlb_mod 所调用，按理说它在执行完之后会返回 do_tlb_mod 调用它的地方，但是如果设成**无返回值**，它会通过系统调用 syscall_set_trapframe 把 CPU 状态恢复到 tlb mod 异常发生前的状态，直接跳转到用户程序中断的位置，接着直接执行原来正常的用户态代码
3. 这里即便给发生写异常的那个进程分配了新的物理页，与之共享旧物理页的那个进程也不能取消 **PTE_COW** 标志，这是因为有可能是多个进程共享一个物理页，而不能想当然的认为共享这个物理页的只有一对父子进程

我们好像没看到给 va 分配新的物理页的过程，其实这个过程不是显式完成的：

1. 首先我们借助了一块虚拟的用户地址 UCOW，首先用 syscall_mem_alloc 为它分配了一个物理页，这个物理页现在和 va 一点关系都没有，就是一个全新的物理页
2. 接着我们使用 memcpy 这个函数，这个函数实际复制的是**物理地址**的内容，也就是把 va 映射到的物理页内容，复制到 UCOW 所映射到的物理页，那么此时 UCOW 所映射的物理页内容就和 va 相同了，但他们本质上还是两个物理页
3. 最后我们只需要借助 syscall_mem_map 函数再让 va 也映射到这个新物理页，在映射的同时修改好新的权限 perm，然后再取消掉 UCOW 对这个物理页的映射即可（保证下次 UCOW 还可以共享使用）

这里为什么不用 sys_mem_map 代替 memcpy？这是因为前者是地址共享，后者是内容复制，如果使用前者，va 得到的还是原来的旧物理页，而我们要的是分配一个全新的（只是内容和旧物理页相同）的物理页

逻辑关系：do_tlb_mod 和 cow_entry 的逻辑关系如下

```
写入共享只读页（COW 页）
        ↓
触发 TLB Mod 异常
        ↓
内核进入 do_tlb_mod()
        ↓
do_tlb_mod 把控制权转交给用户态 cow_entry
        ↓
cow_entry 处理异常，完成真正的页复制
        ↓
恢复原始程序继续执行写操作
```

需要了解的函数：

entry.S

entry.S：用户进程的入口

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

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

libos.c

libos.c：用户进程入口的 C 语言部分，负责完成执行用户程序 main 前后的准备和清理工作

```
#include <env.h>
#include <lib.h>
#include <mmu.h>

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");
}

const volatile struct Env *env;
extern int main(int, char **);

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

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

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

genex.S

genex.S：实现异常处理流程

```
#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
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
#endif

BUILD_HANDLER reserved do_reserved
```

COW 技术

基本概念

定义：COW 技术被称为**写时复制**（Copy-On-Write），用于高效地复制或共享资源

原理： COW 的核心思想在于，多个进程可以共享同一份资源，直到其中一个进程尝试修改该资源的时候，系统才会真正复制该资源供修改进程专用，这是因为，在调用 fork 前后，子进程会继承父进程地址空间中的代码段和数据段等内容，而只要进行了 fork，子进程和父进程就是两个相互独立的进程了，因此他们对各自内存的修改应该是互不影响的

如果我们在 fork 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，而这些物理内存中，如**代码段**部分，父子进程通常不会对其进行写入，对于这样的页面，我们希望能够避免对它们进行复制，从而可以节省物理内存，为了父子进程能够共用尽可能多的物理内存，我们希望引入一种写时复制（Copy-on-write，COW）机制：

1. 在 fork 的时候，只需要把父进程地址空间中的所有页度标记为**写时复制**页面，也就是打上 COW 标记
2. 根据 COW 标记，在父进程或者子进程对写时复制页面进行写入的时候，能够产生一种**异常**，这个异常就是我们上文由 TLB Mod 异常所引发的，在用户态处理的 cow_entry() 异常
3. 在这个异常处理函数内部，操作系统会为当前试图写入的虚拟地址分配新的物理页面，新的页面会拷贝旧页面中的内容，处理完异常后回到正常执行原有用户程序
4. 此时原有的写进程就可以对新生成的物理页面进行写入

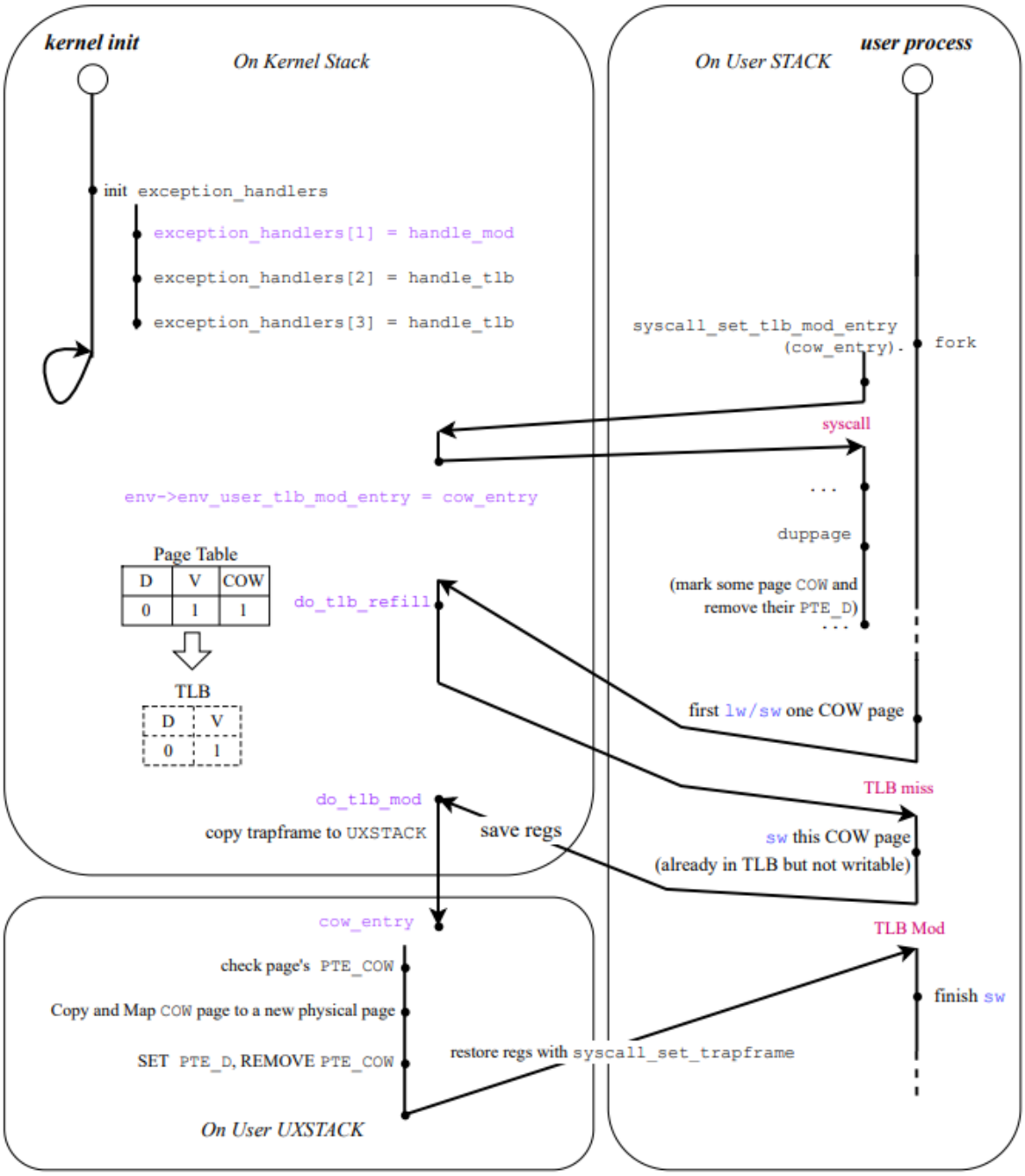
总结：当父进程创建子进程的时候，传统的 fork() 会完整复制父进程的地址空间，而使用 COW 的 fork() 只复制父进程的页表，父子进程共享物理内存，当任一进程尝试写入的时候，会触发 TLB Mod 异常，此时才会创建一个新的物理页给写进程使用

PTE_COW 位设置

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 TLB Mod 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

进程调用 fork 时，需要对其所有的**可写入**的内存页面， 设置页表项标志位 PTE_COW 并取消可写位 PTE_D，以实现写时复制保护，在这样的保护下，用 户程序可以在逻辑上认为 fork 时父进程内存的状态被完整复制到子进程中，此后父子进程都可以独立操作各自的内存

页写入异常流程



思考题

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？
>
> ■

通用寄存器的保护：在 SAVE_ALL 宏中，先 使用 `move k0,sp`，把通用寄存器的 sp 指针复制到 k0 寄存器，再使用 `sw k0,TF_REG29(sp)` `sw $2,TF_REG2(sp)` 进行现场保存，保存现场需要使用 v0 作为协寄存器到内存的中专寄存器，写到内存时需要 sp，所以正式保存协寄存器和通用寄存器需要这两个寄存器

可以获得：这是因为，从用户函数 `syscall_*()` 到内核函数 `sys_*()` 的过程中，a1 到 a3 寄存器并未改变，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_*()` 中仅作为占位符，不会被用到；同时，在内核态中可能使用这些寄存器，因此寄存器原有值会被改变，再次使用这些寄存器的时候需要以 sp 为基地址，按照相应偏移从用户栈中获得这四个寄存器

用户调用时的参数：

1. 前四个参数用 a0 到 a3 存储，只需要在栈中留出相应空间，但不需要真正参数入栈
2. 后两个参数用栈存储，分别在 sp+16 和 sp+20 位置存入

系统调用对进程上下文的修改：

1. 把栈中存储的 EPC 寄存器的值 +4，这是因为系统调用以后会直接执行触发调用的下一条指令，而用户程序可以保证系统调用不在延迟槽内发生，因此直接 +4 得到的就是下一条指令的地址
2. 系统调用的返回值会存入 v0 寄存器中

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？ ■

因为进程的唯一标志是 mkenvid 中的静态变量 i，而不是 envid 的后 10 位，同一个数组可能多次生成不同的进程，envid 的后 10 位是可能相同的，如果不判断 envid 是否相同，则可能会取到错误的进程块，或者曾经创造过现在已经被销毁的进程块

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。 ■

该函数确实不会返回 0，这是由 ++i（i 是 static 变量）所保证的，而 envid2env() 函数在 envid 为 0 的时候会返回 curenv

因为 curenv 为内核态的变量，用户态不能获取 curenv 的 envid，所以用 0 表示 curenv 的envid，这是为了方便用户进程调用 `syscall_*()` 的时候把当前进程的 envid 作为参数传给内核函数，方便用户在内核变量不可见的前提下调用内核接口

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 ■

C 为正解，fork() 只在父进程中执行，因为 fork() 之前只有父进程没有子进程，子进程由 fork() 产生，且二者的返回值不同，在子进程中返回值为 0，在父进程中返回值为子进程的 pid（Linux 进程专属的 id，类似于 MOS 中的 envid）

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 0 - USTACKTOP 范围的内存需要用 duppage 进行映射

在 USTACKTOP - UTOP 范围内的 user exception stack 是用来处理页写入异常的，不会在处理 COW 异常的时候调用 fork()，因此这一页不需要共享

在 USTACKTOP 到 UTOP 范围内的 invalid memory 是为处理页写入异常做缓冲区用的，也不需要进行共享

在 UTOP 之上页面的内存与页表是索引进程共享的，且用户进程无权限访问，不需要做父子进程之间的 duppage，其上范围的内存要么属于内核，要么是所有用户进程的共享空间，用户模式下只可以读不可以写

除去只读和共享之外的页面都需要设置 PTE_COW 保护

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？
>
> ■

作用：在用户态下通过访问进程自己的物理页内存，获取用户页的页目录项和页表项的 perm，用于 duppage根据不同的 perm 类型完成父子进程之间不同物理页的映射

具体使用：

1. vpd 是**页目录首地址**，以 vpd 为基地址，加上页目录偏移数即可指向 va 所对应的页目录项，即 `vpd[va >> 22]` 或 `*(vpd + (va >> 22))`
2. 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 >> 22]` 或 `*(vpt + (va >> 22))`

自映射设计的体现：

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

vpd 的地址在 UVPT 和 UVPT + PDMAP 之间，说明把页目录映射到了某一个页表的位置，即实现了自映射

进程不可以修改自己的页表，尽管用户程序可以访问这部分地址，但只能只读访问，不能进行写操作，只有内核能够对用户页表进行修改

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？
>
> ■

异常重入：当出现 COW 异常的时候，需要使用用户态的系统调用发生中断，即进行中断重入

因为处理 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`，推出内核中断后跳转到 epc 所在用户态的异常处理函数，此时用户态把异常处理完毕后，仍然处在用户态进行现场恢复，因此此时要把内核保存的现场保存在用户空间的用户异常栈，供用户获取有关的上下文恢复信息

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？ ■

有以下优势：

1. 解放内核，不用内核执行大量的页面拷贝工作
2. 内核态处理失误产生影响大，可能使操作系统崩溃；用户态则不会产生这么严重的后果
3. 用户态访问内存权限有限，可以防止越界访问，修改不该访问的位置
4. 在微内核模式下，用户态进行新的页面分配映射更加灵活和方便

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()`，因此该函数的调用应该位于 `syscall_exofork()` 之前

父进程在调用写时复制保护机制可能会引发却页异常，而异常处理未设置好，此时不能正常处理

其余系统调用

```
extern struct Env *curenv;

/* Overview:
  函数功能：c 本质上是个字符，函数功能即为把 c 打印到屏幕
 */
void sys_putchar(int c) {
        printcharc((char)c);
        return;
}

/* Overview:
  函数功能：把长度为 num 的字符串 s 打印到屏幕
 */
int sys_print_cons(const void *s, u_int num) {
        if (((u_int)s + num) > UTOP || ((u_int)s) >= UTOP || (s > s + num))
{
                return -E_INVAL;
        }
        u_int i;
        for (i = 0; i < num; i++) {
                printcharc(((char *)s)[i]);
        }
        return 0;
}

/* Overview:
  函数功能：获取当前运行中进程 curenv 的进程编号 env_id，其实要获取 curenv->env_id
的时候，写这个和直接写 0 没啥区别
 */
u_int sys_getenvid(void) {
        return curenv->env_id;
}

/* Overview:
  函数功能 : 销毁当前进程
  前置条件 : env_id 必须是一个进程的环境 ID，该进程必须是调用者的子进程或者调用者自己
  后置条件 : 成功时返回 0，错误时返回对应错误码
 */
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;
}

/* Overview:
  函数功能 : 检查虚拟地址 va 是否非法，返回 1 为非法
```

```
 */
static inline int is_illegal_va(u_long va) {
// UTEMP 是用户空间允许映射的最低虚拟地址，其下的空间用于捕捉 NULL 指针访问，内核使用，或者将来用于扩展；UTOP 是用户栈顶，合法的 va 应该介于 [UTEMP, UTOP) 之间
        return va < UTEMP || va >= UTOP;
}

/* Overview:
  函数功能 : 检查一段虚拟地址 [va, va + len) 是否非法，返回 1 为非法
 */
static inline int is_illegal_va_range(u_long va, u_int len) {
        if (len == 0) {
                return 0;
        }
        return va + len < va || va < UTEMP || va + len > UTOP;
}

/* Overview:
  函数功能 : 使用消息 msg 触发内核崩溃 panic
  后置条件 : 该函数被调用以后，系统停止运行(halt)
 */
void sys_panic(char *msg) {
        panic("%s", TRUP(msg));
}

/* Overview:
  函数功能 : 把位于虚拟地址 va，长度为 len 的数据写入设备的物理地址 pa，需要注意 va 和 pa 的有效性

  前置条件 : len 必须是 1 2 4 中之一

  后置条件 : 把 [va, va + len) 中的数据复制到 pa

  提示 : 可以使用内核地址空间(KSEG1)中未映射、非缓存的段来进行 MMIO(内存映射I/O)，方法是直接给对应长度的地址赋值；或者直接使用 include/io.h 中定义的 IO 写函数，如 iowrite32、iowrite16、iowrite8

  所有有效的设备及其物理地址范围如下 :
       * ---------------------------------*
       |   device    | start addr | length |
       * -----------+------------+--------*
       |  控制台      | 0x180003f8 | 0x20   |
       |  IDE硬盘     | 0x180001f0 | 0x8    |
       * ---------------------------------*
 */
/*
  有效物理设备的地址数量为 2，控制台和IDE硬盘
*/
int valid_addr_space_num = 2;
/*
  存储每个有效设备的物理起始地址，控制台的起始地址为 0x180003f8，IDE 硬盘的起始地址为 0x180001f0
*/
unsigned int valid_addr_start[2] = {0x180003f8, 0x180001f0};
/*
  存储每个有效设备的物理结束地址，控制台的结束地址为 0x18000418，IDE 硬盘的结束地址为 0x180001f8
*/
```

```c
unsigned int valid_addr_end[2] = {0x180003f8 + 0x20, 0x180001f8};

/* Overview
  函数功能 : 检查物理地址 pa 和长度 len 是否在某一个合法设备地址范围内，返回 1 表示无法找到这样的设备; 返回 0 表示可以找到这样的合法设备
*/
static inline int is_illegal_dev_range(u_long pa, u_long len) {
// 检查地址对齐 : 如果 pa 不是 4 字节对齐，则只能写 1 2 字节; 如果 pa 不是 2 字节对齐，则只能写 1 字节
        if ((pa % 4 != 0 && len != 1 && len != 2) || (pa % 2 != 0 && len != 1)) {
                return 1;
        }
        int i;
        u_int target_start = pa;// 目标地址起点
        u_int target_end = pa + len;// 目标地址终点
// 遍历所有有效地址范围，如果当前访问的地址范围 [target_start, target_end] 在某个 [valid_addr_start, valid_addr_end] 之间，则说明找到合法的设备范围，直接返回 0 即可
        for (i = 0; i < valid_addr_space_num; i++) {
                if (target_start >= valid_addr_start[i] && target_end <= valid_addr_end[i]) {
                        return 0;
                }
        }
        return 1;
}
/* Overview :
  函数功能 : 把虚拟地址 va 开始，长度为 len 的数据，写到设备对应物理地址为 pa 的区域中
  前置条件 : len 必须是 1 2 4 之一
  后置条件 : 写入成功时，虚拟地址范围 [va, va + len) 区域内的数据会被复制到物理地址 [pa, pa + len) 区域内；写入失败时会返回响应的错误码
*/
int sys_write_dev(u_int va, u_int pa, u_int len) {
        /* Exercise 5.1: Your code here. (1/2) */
        if (is_illegal_va_range(va, len) || is_illegal_dev_range(pa, len) || va % len != 0) {
                return -E_INVAL;
        }
        if (len == 4) {
                iowrite32(*(uint32_t *)va, pa);
        } else if (len == 2) {
                iowrite16(*(uint16_t *)va, pa);
        } else if (len == 1) {
                iowrite8(*(uint8_t *)va, pa);
        } else {
                return -E_INVAL;
        }
        return 0;
}

/* Overview:
  函数功能 : 从设备的物理地址中读取数据
  前置条件 : len 必须是 1 2 4 其中之一
  后置条件 : 从物理地址 pa 中读取的数据会被复制到虚拟地址空间 [va, va + len) 之中，成功时返回 0，失败返回对应错误码
 */
int sys_read_dev(u_int va, u_int pa, u_int len) {
        /* Exercise 5.1: Your code here. (2/2) */
```

```
150         if (is_illegal_va_range(va, len) || is_illegal_dev_range(pa, len)
    || va % len != 0) {
151                 return -E_INVAL;
152         }
153         if (len == 4) {
154                 *(uint32_t *)va = ioread32(pa);
155         } else if (len == 2) {
156                 *(uint16_t *)va = ioread16(pa);
157         } else if (len == 1) {
158                 *(uint8_t *)va = ioread8(pa);
159         } else {
160                 return -E_INVAL;
161         }
162         return 0;
163 }
```

后两个调用是和硬件有关的调用，目前还涉及不到，其中检验虚拟地址和虚拟地址范围是否合法的这两个系统调用，可能会经常调用，别忘了他们俩就行

参考与引用

笔记中的很多内容和代码示例都来自以下两位学长 / 学姐的博客内容，在此表示深刻感谢：

https://cookedbear.top/p/1727.html

https://yanna-zy.github.io/2023/04/18/BUAA-OS-3/