

BUAA_OS_Lab4笔记

BUAA_OS_Lab4笔记

重要宏
用户态、内核态、用户空间、内核空间
系统调用

```
kern/syscall_all.c
void do_syscall (struct Trapframe *tf)
int sys_mem_alloc(u_int envid, u_int va, u_int perm)
int sys_mem_map
int sys_mem_unmap(u_int envid, u_int va)
int sys_ipc_recv(u_int dstva)
int sys_ipc_try_send
```

```
Fork
kern/syscall_all.c
int sys_exofork(void)
int sys_set_tlb_mod_entry(u_int envid, u_int func)
int sys_set_env_status(u_int envid, u_int status)

kern/tlbex.c
void do_tlb_mod(struct Trapframe *tf)

user/lib/fork.c
static void duppage(u_int envid, u_int vpn)
static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf)
int fork(void)
```

添加系统调用
新增权限位
共享内存

重要宏

符号	类型	数组下标是什么	取出来的是什么
vpt[]	页表项数组	虚拟页号 (VPN)	虚拟地址对应的页表项 (Pte)
vpd[]	页目录项数组	页目录号 (PDX)	虚拟地址对应的页目录项 (Pde)

常用：

获得权限位：

```
perm = vpt[VPN(va)] & 0xffff; //获得va的perm
```

权限位概览 (定义于 mmu.h)：

```

1 // valid bit. will cause a tlb miss exception (TLBL/TLBS).
2 #define PTE_V (0x0002 << PTE_HARDFLAG_SHIFT) // 有效位, 用于遍历
3
4 // Dirty bit, but really a write-enable bit. 1 to allow writes, 0 and any
  store using this translation will cause a tlb mod exception (TLB Mod).
5 #define PTE_D (0x0004 << PTE_HARDFLAG_SHIFT) // 可写位
6
7 // Copy On Write. Reserved for software, used by fork.
8 #define PTE_COW 0x0001
9
10 // Shared memory. Reserved for software, used by fork.
11 #define PTE_LIBRARY 0x0002

```

遍历页面模板:

```

1 /* Step 3: Map all mapped pages below 'USTACKTOP' into the child's address
  space. */
2 for (i = 0; i < PDX(USTACKTOP); i++) {
3     if (vpd[i] & PTE_V) {
4         for (u_int j = 0; j < PAGE_SIZE / sizeof(Pte); j++) {
5             u_long va = (i * (PAGE_SIZE / sizeof(Pte)) + j) << PGSHIFT;
6             if (va >= USTACKTOP) {
7                 break;
8             }
9             if (vpt[VPN(va)] & PTE_V) {
10                 // duppage(child, VPN(va));
11             }
12         }
13     }
14 }
15 // 等价于:
16 for (u_long va = 0; va < USTACKTOP; va += PAGE_SIZE) {
17     if (vpd[PDX(va)] & PTE_V && vpt[VPN(va)] & PTE_V) {
18         // duppage(child, VPN(va));
19     }
20 }

```

用户态实现系统调用:

`syscall_mem_map(src_envid, src_va, dst_envid, dst_va, perm)` 是用户态系统调用接口, 用于在不同进程之间建立页映射。在 `duppage` 中, 它被用于: 将当前进程的页 `vpn * PAGE_SIZE` 映射到子进程 `envid` 的相同虚拟地址处, 并设置适当权限。

为什么不用自己改页表? 因为在用户态无法直接访问和修改页表 (保护机制), 所以必须通过 `syscall_mem_map` 让内核帮忙修改页表项。 `syscall_mem_map` 封装了内核的 `sys_mem_map`, 可以安全地建立页映射, 并传递权限。

将当前用户进程地址空间中的一页虚拟地址 `va` 对应的页面设置为某权限 (比如共享页, 加上 `PTE_LIBRARY` 权限) 以供多进程共享。

```

1 int make_shared(void *va) {
2     if ((u_int)va >= USTACKTOP) {
3         return -1; // 超出用户空间
4     }

```

```

5
6     u_int addr = ROUNDDOWN((u_int)va, PAGE_SIZE);
7     u_int perm;
8
9     // 如果页不存在, 则分配新页
10    if (!(vpd[PDX(addr)] & PTE_V) || !(vpt[VPN(addr)] & PTE_V)) {           // 重
要模板
11        if (syscall_mem_alloc(0, (void *)addr, PTE_V | PTE_D) < 0) {
12            // 将envid设为0, 表示默认curenv
13            return -1;
14        }
15    }
16
17    perm = vpt[VPN(addr)] & 0xfff;
18
19    // 拒绝只读页参与共享
20    if (!(perm & PTE_D)) {
21        return -1;
22    }
23
24    // 加上共享权限
25    perm |= PTE_LIBRARY;
26
27    if (syscall_mem_map(0, (void *)addr, 0, (void *)addr, perm) < 0) {
28        return -1;
29    }
30
31    // 返回该页的物理地址 (高 20 位)
32    return ROUNDDOWN(vpt[VPN(addr)] & ~0xfff, PAGE_SIZE);
33 }

```

用户态、内核态、用户空间、内核空间

用户态和内核态是 CPU 运行的两种状态。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是**虚拟内存**（进程的地址空间）中的两部分区域。用户空间包括 kuseg，而内核空间主要包括 kseg0 和 kseg1。

- 用户空间系统调用函数： `syscall_*`
- 内核空间系统调用函数： `sys_*`

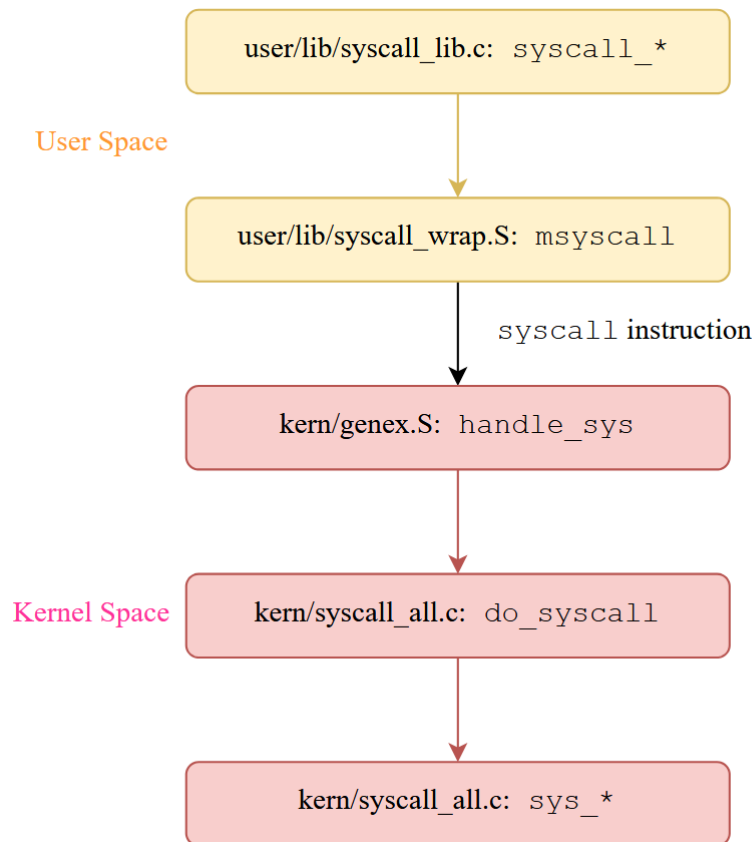
这些用户空间 `syscall_*` 的函数与内核中空间 `sys_*` 是一一对应的。

在所有 `syscall_*` 的函数的实现中，都调用了 `msyscall` (`user/lib/syscall_wrap.S`) 函数。

`msyscall` 函数的第一个参数是一个**系统调用号**。例： `SYS_print_cons`，定义于 `include/syscall.h`。**系统调用号是内核区分不同系统调用的唯一依据。**

`Trapframe` (`include/trap.h`) 结构体用于将原用户进程的运行现场（值）保存到内核空间，即**获取用户态现场中的参数**。

系统调用流程：



系统调用

kern/syscall_all.c

```
void do_syscall (struct Trapframe *tf)
```

接收系统调用号和参数，并**跳转到相应的具体的内核函数**执行，最后将结果返回给用户程序。

做完这一步，整个系统调用的机制已经可以正常工作，接下来要实现几个**具体的系统调用**。具体实现都在 `syscall_all.c` 中。

```
int sys_mem_alloc(u_int envid, u_int va, u_int perm)
```

用于**分配内存**。用户程序可以给该程序所允许的**虚拟内存空间**显式地分配实际的**物理内存**。对于操作系统内核来说，是一个进程请求将其运行空间中的某段地址与实际物理内存进行映射。

其中用到 `envid2env` 函数，功能是通过一个进程的 id 获取该进程控制块。第三个参数为 `perm` 权限位，注意，在内核态系统调用中永远设为 1！！记住模板：`try(envid2env(envid, &env, 1));`

```
int sys_mem_map
```

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。此时两者共享一页物理内存。

```
int sys_mem_unmap(u_int envid, u_int va)
```

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。

下面进入 IPC（进程间通信）

```
int sys_ipc_recv(u_int dstva)
```

接受消息。

```
int sys_ipc_try_send
```

发送消息。

Fork

注意此时全部回到用户态。即 user 目录下。

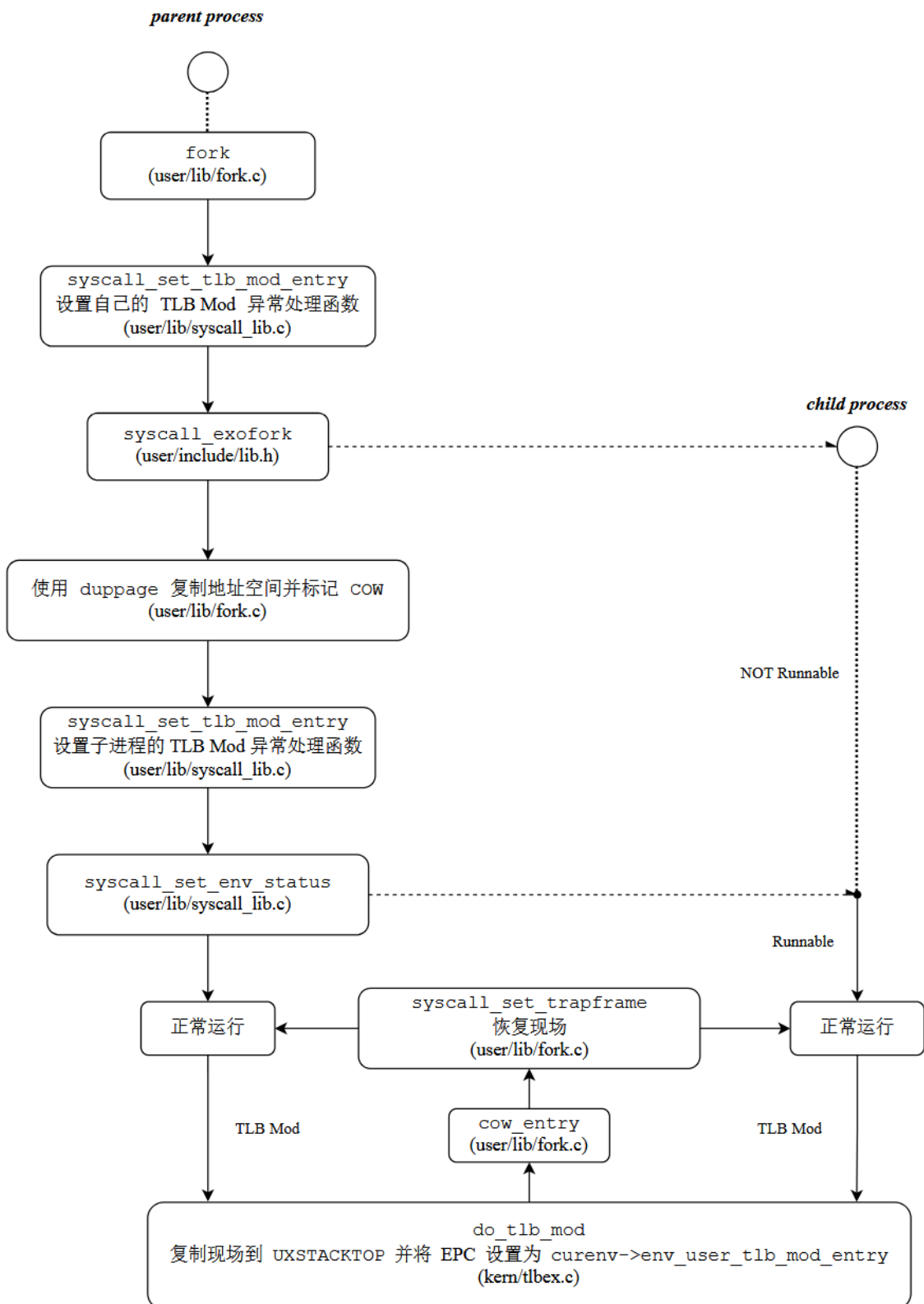


图 4.4: fork 流程图

kern/syscall_all.c

```
int sys_exofork(void)
```

用于新进程的创建。

凡是涉及到有错误返回的，直接用 try 就行。例如 `try(env_alloc(&e, curenv->env_id));`

```
int sys_set_tlb_mod_entry(u_int envid, u_int func)
```

为某个用户进程设置 TLB 异常处理函数（即 TLB Mod 异常时的入口）。

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

改变某个用户环境（env）的运行状态（是否可调度），并相应更新调度队列 `env_sched_list`。

这是实现**进程调度控制**的核心接口，允许用户程序主动将自己或子进程设置为可运行 / 不可运行状态。

kern/tlbex.c

```
void do_tlb_mod(struct Trapframe *tf)
```


处理页写入异常的内核函数。

user/lib/fork.c

```
static void duppage(u_int envid, u_int vpn)
```


将当前进程（父进程）中编号为 `vpn` 的虚拟页，以适当的权限映射到子进程 `envid` 的地址空间中。


若该页是“可写但非共享”的普通页，则应用**写时复制（Copy-On-Write, COW）**机制：父子共享该页，只要有一方尝试写，就触发页异常，系统会自动为其分配新的私有页。

 Step 1: 计算地址与权限

```
1 addr = vpn << PGSHIFT;
2 perm = vpt[vpn] & ((1 << PGSHIFT) - 1);
```

- 得到该页对应的地址（字节地址）；
- `vpt[vpn]` 是页表项，包含权限信息；
- `& ((1 << PGSHIFT) - 1)` 的意思是：只保留低 12 位（权限字段），因为高位是物理页帧地址。

 Step 2: 处理不同类型的页

 情况1: 共享页 或 只读页 或 已是COW

```
1 if ((perm & PTE_D) == 0 || (perm & PTE_LIBRARY) || (perm & PTE_COW)) {
```

说明：

- 如果该页**不是可写的**（没有 PTE_D），说明**原本就是只读页**；
- 或者该页是**共享页**（PTE_LIBRARY），**不进行写时复制**；
- 或者它**已经是 COW 页**。

👉 直接映射一份副本到子进程，权限不变

```
1 syscall_mem_map(0, (void *)addr, envid, (void *)addr, perm)
```

⚠ else: 需要设为写时复制 (Copy-On-Write)

说明：

- 当前页是 **父进程独占的、可写的私有页**；
- 若直接复制物理页，会造成浪费；
- 所以要做 COW —— 写的时候再复制。

关键顺序：先映射子进程，再映射父进程！

```
1 syscall_mem_map(0, (void *)addr, envid, (void *)addr, (perm & ~PTE_D) |  
PTE_COW);  
2 syscall_mem_map(0, (void *)addr, 0, (void *)addr, (perm & ~PTE_D) | PTE_COW);
```

- 把原页清除写权限（去掉 PTE_D）；
- 添加 COW 标志；
- 父子都共享这一个物理页；
- 将来如果某个进程写了这个页，会触发异常，内核将分配新页。

为什么先映射子，再映射父？

如果先处理父进程，会导致父进程失去 PTE_D 权限（变成 COW），影响你读取页表权限去设置子进程的副本。所以顺序不能错。

```
static void __attribute__((noreturn)) cow_entry(struct
Trapframe *tf)
```

写时复制 (Copy-On-Write, COW) 异常处理机制的核心:

当子进程或父进程试图**写入一个 COW 页**时，会触发 TLB Mod 异常，系统跳转到这个用户级异常处理函数 `cow_entry`：它将触发写的 COW 页复制出一份新的物理页，并将原虚拟地址 `remap` 成这块新页的可写副本。

```
int fork(void)
```

一句话总结：在用户态调用 `fork()` 时，会创建一个子进程，并通过写时复制（COW）机制共享父进程的内存页，直到任意一方写入该页时再触发真正复制。

1
2 用户调用 `fork()`
3
4

内核函数：

```
1 int sys_print_sum(u_int a, u_int b);
```

功能是打印 `a + b` 到控制台（你可以换成任意你想实现的功能）。

✓ 1. 修改 `user/include/lib.h` 添加声明

```
1 void user_print_sum(u_int a, u_int b);
2 void syscall_print_sum(u_int a, u_int b);
```

✓ 2. 实现 `syscall_print_sum` 到 `user/lib/syscall_lib.c`

```
1 int syscall_print_sum(u_int a, u_int b) {
2     return msyscall(SYS_print_sum, a, b);
3 }
```

✓ 3. 实现 `user_print_sum` 到用户函数实现文件（可放在某个测试用例文件）

```
1 void user_print_sum(u_int a, u_int b) {
2     syscall_print_sum(a, b);
3 }
```

✓ 4. 修改 `include/syscall.h`

```
1 enum 的MAX_SYSNO 前面加上 SYS_print_sum,
```

✓ 5. 修改 `kern/syscall_all.c` 中 `syscall_table`

```
1 // 在 syscall_table[] 最后添加（注意有逗号）：
2 [SYS_print_sum] = sys_print_sum,
```

✓ 6. 实现内核函数 `sys_print_sum`（写在 `syscall_table` 上面）

```
1 int sys_print_sum(u_int a, u_int b, u_int unused1, u_int unused2, u_int
  unused3) {
2     // 打印两个数的和
3     printk("Syscall: a + b = %d\n", a + b);
4     return 0;
5 }
```

✓ 总结新增流程

步骤	文件	修改
1	<code>user/include/lib.h</code>	添加函数声明
2	<code>user/lib/syscall_lib.c</code>	实现 <code>syscall_...</code> ，调用 <code>msyscall()</code>
3	用户函数文件	写 <code>user_...</code> 函数
4	<code>include/syscall.h</code>	定义 <code>SYS_...</code> 宏 + 加到枚举
5	<code>kern/syscall_all.c</code>	加入 <code>syscall_table[]</code>
6	<code>kern/syscall_all.c</code>	实现 <code>sys_...</code> 逻辑

新增权限位

现在要**新增**一种“**保护权限 (protected permission)**”，目的是在当前已有的**私有页 (PTE_D 写权限)**和**共享页 (PTE_LIBRARY)**基础上，引入**第三种：保护页**，从而实现权限分类为：

- ✅ 私有页（默认 `PTE_D` 写权限）
- ✅ 共享页（`PTE_LIBRARY`）
- ✅ 🛡️ 受保护页（你要新增）

页表项中哪一位可以用来新增 `PTE_PROTECTED` 权限？根据 `mmu.h` 权限宏表，软件位只占用低位，你可以继续用第2位之后的空位（避免冲突）。建议新增：

```
1 #define PTE_PROTECTED 0x0004 // 第2位
```

- 与 `PTE_COW = 0x0001`、`PTE_LIBRARY = 0x0002` 并列
- 不会和硬件权限位（已经左移 `PTE_HARDFLAG_SHIFT`）冲突

使用规范设计建议：你需要给 `PTE_PROTECTED` 添加一些访问限制（否则它就成了个无用 flag），例如：

- ❌ 不能被 `syscall_mem_map` 映射给其他进程（防止泄漏）
- ✅ 可以通过 `is_protected(void *va)` 进行权限检查

实现 `is_protected(void *va)`（用户态）

```
1 int is_protected(void *va) {
2     u_int perm = vpt[VPN(ROUNDDOWN((u_int)va, BY2PG))] & 0xffff;
3     return (perm & PTE_PROTECTED) != 0;
4 }
```

在内核中限制行为的地方：

- ✅ 1. `sys_mem_map`

```

1 // Step 4: 找到物理页之后, 检查是否为保护页
2 if (vpt[VPN(srcva)] & PTE_PROTECTED) {
3     return -E_INVALID; // 是受保护页, 禁止操作
4 }

```

目标: 禁止任何环境把这个页映射给别人, 包括 fork/cow 调用时 duppage()

✓ 2. sys_ipc_try_send

加保护:

```

1 if (srcva != 0) {
2     p = page_lookup(curenv->env_pgdir, srcva, NULL);
3     if (p == NULL) return -E_INVALID;
4
5     // 新增保护检查
6     if (vpt[VPN(srcva)] & PTE_PROTECTED) {
7         return -E_INVALID; // 是受保护页, 禁止操作
8     }
9
10    try(page_insert(e->env_pgdir, e->env_asid, p, e->env_ipc_dstva, perm));
11 }

```

共享内存

你需要实现一个完整的共享内存系统调用框架, 包含:

- ✓ 创建共享页 (带唯一 key)
- ✓ 映射共享页到某进程某地址
- ✓ 解绑共享页 (引用计数减一)
- ✓ 释放共享页 (只有引用计数为 0 时才释放)

我们使用一个共享页表 (数组) `struct SharedPage shared_pages[MAX_SHARED_PAGE]` 来全局维护共享内存页。

✓ 一、设计共享页的数据结构

```

1 #define MAX_SHARED_PAGE 64
2
3 struct SharedPage {
4     int key; // 唯一标识符 (正整数)
5     struct Page *pp; // 指向共享物理页
6     int ref_count; // 映射到进程的数量
7     int is_used; // 是否已被分配
8 };
9
10 struct SharedPage shared_pages[MAX_SHARED_PAGE];

```

✓ 二、定义 4 个系统调用接口

1 创建共享页

```
1 int sys_shared_page_create(int key) {
2     for (int i = 0; i < MAX_SHARED_PAGE; i++) {
3         if (shared_pages[i].is_used && shared_pages[i].key == key) {
4             return -E_INVALID; // 已存在
5         }
6     }
7     for (int i = 0; i < MAX_SHARED_PAGE; i++) {
8         if (!shared_pages[i].is_used) {
9             struct Page *pp;
10            if (page_alloc(&pp) < 0) return -E_NO_MEM;
11            shared_pages[i].key = key;
12            shared_pages[i].pp = pp;
13            shared_pages[i].ref_count = 0;
14            shared_pages[i].is_used = 1;
15            return 0;
16        }
17    }
18    return -E_NO_MEM;
19 }
```

2 映射共享页到当前进程地址空间

```
1 int sys_shared_page_map(int key, u_int va, u_int perm) {
2     if (is_illegal_va(va)) return -E_INVALID;
3
4     for (int i = 0; i < MAX_SHARED_PAGE; i++) {
5         if (shared_pages[i].is_used && shared_pages[i].key == key) {
6             struct Page *pp = shared_pages[i].pp;
7             if (page_insert(curenv->env_pgdir, curenv->env_asid, pp, va,
perm) < 0) {
8                 return -E_NO_MEM;
9             }
10            shared_pages[i].ref_count += 1;
11            return 0;
12        }
13    }
14    return -E_INVALID; // 没找到
15 }
```

3 解绑共享页（当前进程取消映射）

```
1 int sys_shared_page_unmap(int key, u_int va) {
2     if (is_illegal_va(va)) return -E_INVALID;
3
4     for (int i = 0; i < MAX_SHARED_PAGE; i++) {
5         if (shared_pages[i].is_used && shared_pages[i].key == key) {
6             page_remove(curenv->env_pgdir, curenv->env_asid, va);
7             if (shared_pages[i].ref_count > 0) {
8                 shared_pages[i].ref_count -= 1;
9             }
10        }
11    }
```

```

10         return 0;
11     }
12 }
13 return -E_INVAL;
14 }

```

4 释放共享页（只有 ref_count == 0 时才释放）

```

1 int sys_shared_page_free(int key) {
2     for (int i = 0; i < MAX_SHARED_PAGE; i++) {
3         if (shared_pages[i].is_used && shared_pages[i].key == key) {
4             if (shared_pages[i].ref_count > 0) {
5                 return -E_SHARE_IN_USE; // 正在使用，不能释放
6             }
7             page_free(shared_pages[i].pp);
8             shared_pages[i].is_used = 0;
9             return 0;
10        }
11    }
12    return -E_INVAL;
13 }

```

✓ 三、系统调用注册

include/syscall.h 添加：

```

1 #define SYS_shared_create  XX
2 #define SYS_shared_map    XX
3 #define SYS_shared_unmap  XX
4 #define SYS_shared_free   XX

```

并加到 enum syscall 中。

syscall_all.c 注册：

```

1 [SYS_shared_create] = sys_shared_page_create,
2 [SYS_shared_map] = sys_shared_page_map,
3 [SYS_shared_unmap] = sys_shared_page_unmap,
4 [SYS_shared_free] = sys_shared_page_free,

```

✓ 四、用户接口 (user/include/lib.h) :

```

1 int shared_page_create(int key);
2 int shared_page_map(int key, void *va, u_int perm);
3 int shared_page_unmap(int key, void *va);
4 int shared_page_free(int key);

```

实现对应 msyscall 即可。