

Lab4

Thinking

Thinking 4.1

- 内核在保存现场的时候是如何避免破坏通用寄存器的？

在 `SAVE_ALL` 宏中，内核将原用户进程的执行现场，保存至内核的异常栈中，随后的栈指针 (`sp`) 则指向保存的 `Trapframe`。执行现场信息包括：部分 `CP0` 寄存器和通用寄存器。

```
.macro SAVE_ALL
.set noat
.set noreorder
    //检查UM位:若STATUS_UM=0,则处理器处于内核态
    mfc0    k0, CP0_STATUS
    andi    k0, STATUS_UM
    beqz    k0, 1f
    //对于UM=1,非内核态(非异常重入)
    //延迟槽机制:将 sp 寄存器的值复制到 k0 寄存器,保存sp
    move    k0, sp
    //将sp寄存器指向内核异常栈
    li      sp, KSTACKTOP
1:
    subu   sp, sp, TF_SIZE
    sw     k0, TF_REG29(sp)
    mfc0   k0, CP0_STATUS
    sw     k0, TF_STATUS(sp)
    mfc0   k0, CP0_CAUSE
    sw     k0, TF_CAUSE(sp)
    mfc0   k0, CP0_EPC
    sw     k0, TF_EPC(sp)
    mfc0   k0, CP0_BADVADDR
    sw     k0, TF_BADVADDR(sp)
    mfhi   k0
    sw     k0, TF_HI(sp)
    mflo   k0
    sw     k0, TF_LO(sp)
    sw     $0, TF_REG0(sp)
    sw     $1, TF_REG1(sp)
    /*.....*/
    sw     $30, TF_REG30(sp)
```

```
    sw      $31, TF_REG31(sp)
.set at
.set reorder
.endm
```

- 系统陷入内核调用后可以直接从当时的 `$a0~$a3` 通用寄存器中得到用户调用 `msyscall` 留下的信息吗？

可以。因为陷入内核后，一般不会有操作改变 `$a0~$a3` 寄存器的值；但从内核栈中取出更安全。

- 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？

1. 在 `syscall_*` 函数调用 `msyscall` 函数，进行传参时：

将前4个参数被 `syscall_*` 函数存入 `$a0~$a3` 寄存器；

2. 调用 `SAVE_ALL` 宏，将通用寄存器 `$a0~$a3` 的值，保存至内核栈 `TF_REG0 ~ TF_REG3` 中；

3. 执行 `do_syscall` 函数时，从内核栈中取出对应参数：

```
void do_syscall(struct Trapframe *tf){
    int sysno = tf->regs[4];
    u_int arg1 = tf->regs[5];
    u_int arg2 = tf->regs[6];
    u_int arg3 = tf->regs[7];
}
```

- 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？

`do_syscall` 函数执行了 `tf->cp0_epc+=4;`，确保系统调用返回时，返回到下一条指令。

Thinking 4.2

- 为什么 `envid2env` 函数中要判断： `e->env_id != envid` 的情况

互逆函数：`mkenvid` 函数，`envid2env` 函数

`mkenvid` 函数：为进程块e分配进程id(由 `env_alloc` 调用)

`mkenvid` 中 `i` 为静态变量，每次调用时累加；因此对于同一进程控制块，在不同阶段分配时，其id不同。

所以，在 `envid2env` 函数中，需要检查：进程控制块 `e` 当前是否对应进程 `envid`。

```
u_int mkenvid(struct Env *e) {
    static u_int i = 0;
```

```
        return ((++i) << (1 + LOG2NENV)) | (e - envs);  
    }
```

```
int env_alloc(struct Env **new, u_int parent_id){  
    /*...*/  
    e->env_id=mkenvid(e);  
    /*...*/  
}
```

envid2env 函数：通过进程id，获取对应的进程块e(地址存在*penv中)

```
int envid2env(u_int envid, struct Env **penv, int checkperm){  
    /*...*/  
    e=&envs[ENVX(envid)];  
    if(envid==0){           //当前控制块curenv  
        *penv=curenv;  
        return 0;  
    }  
    if (e->env_status == ENV_FREE || e->env_id != envid){  
        return -E_BAD_ENV;  
    }  
}
```

Thinking 4.3

由于 $++i > 0$ ，因此： mkenvid 函数不会分配为0的进程id.

```
u_int mkenvid(struct Env *e) {  
    static u_int i = 0;  
    return ((++i) << (1 + LOG2NENV)) | (e - envs);  
}
```

Thinking 4.4

关于 fork 函数的两个返回值，说法正确的是： 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。

Thinking 4.5

应该对 $0 \sim USTACKTOP$ 范围内的用户空间页进行映射， $USTACKTOP$ 以上为内核空间，或者所有用户进程的共享空间（用户模式下只可读）

Thinking 4.6

1. vpt和vpd的作用是什么？怎样使用它们？

- vpt 是页表首地址，可获取 va 对应的页表项： `((Pte *)(*vpt))+(va>>12)` ;
- vpd 是页目录首地址，可获取 va 对应的页目录项： `((Pde *)(*vpd))+(va>>22)` .
有宏定义：

```
#define PDMAP (4 * 1024 * 1024)
#define PDSHIFT 22 // log2(PDMAP)
#define PGSHIFT 12
#define PDX(va) (((u_long)(va)) >> PDSHIFT) & 0x03FF

#define ULIM 0x80000000      //kseg0起始地址
#define UVPT (ULIM - PDMAP)
//页表首地址
#define vpt ((const volatile Pte *)UVPT)
//页目录首地址
#define vpd ((const volatile Pde *)((UVPT + (PDX(UVPT) << PGSHIFT))))
```

2. 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？

3. 它们是如何体现自映射设计的？

页目录首地址 vpd 为: `vpd | (vpd>>10)` ,是自映射的特点。

4. 进程能够通过这种方式来修改自己的页表项吗？

不能。若要修改页表项，这里开启了写时复制保护：`PTE_D=0, PTE_COW=1`，修改时需要抛出 `TLB Mod` 异常，陷入内核进行操作。

Thinking 4.7

在 `do_tlb_mod` 函数中，有向异常处理栈复制 `Trapframe` 运行现场的过程。

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”？

缺页中断时再次响应外部中断，在标志有 `COW` 的页面被修改时会出现。

内核为什么需要将异常的现场 `Trapframe` 复制到用户空间？

页写入异常是在用户态下完成的。

将异常现场保存至用户异常处理栈(栈顶为 `UXSTACKTOP`)后，使得：从异常恢复后，能够以异常处理栈中保存的现场参数，跳转至

`env_user_tlb_mod_entry` 域存储的用户异常处理函数的地址。

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？

- 减少数据拷贝的开销：如果在内核态处理页写入异常时，需要将数据从用户空间拷贝到内核空间进行处理，然后再将结果拷贝回用户空间。
 - 具有更好的可扩展性：用户态程序可以根据其特定需求来定制页写入异常的处理方式。

Thinking 4.9

为什么需要将 `syscall_set_tlb_mod_entry` 的调用放置在 `syscall_exofork` 之前？

先注册当前进程的 TLB Mod 异常处理函数，再创建子进程。

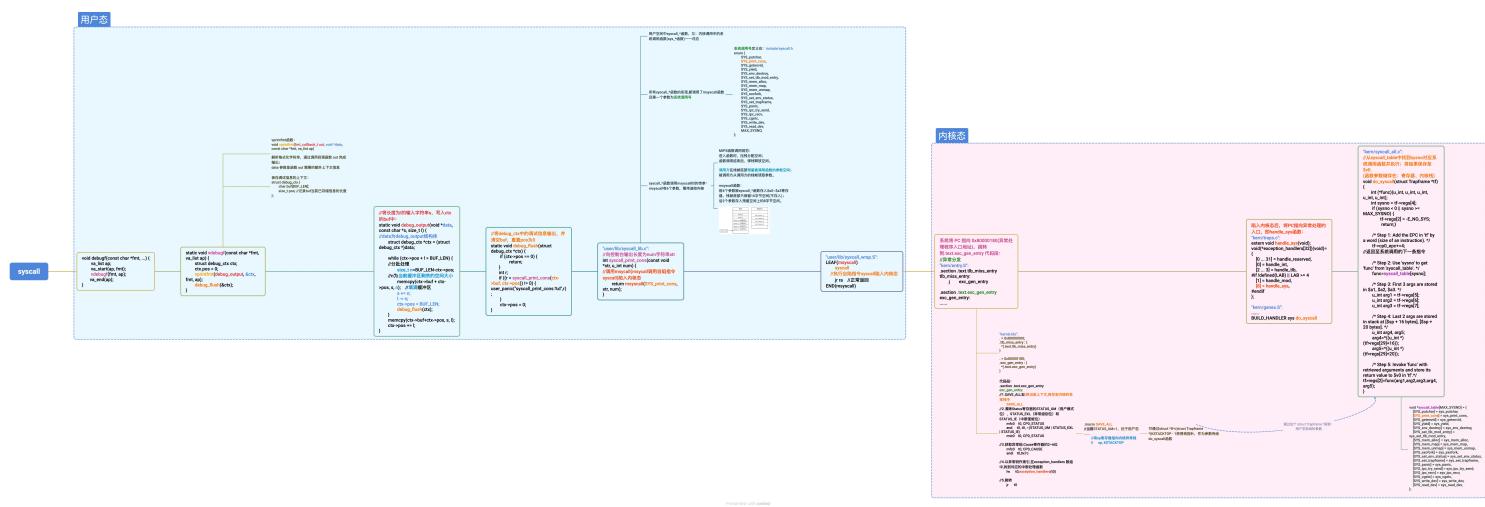
如果在创建子进程之后注册，由于 `syscall_exofork` 和 `syscall_set_tlb_mod_entry` 之间有一定的时间间隔，

如果放置在写时复制保护机制完成之后会有怎样的效果？

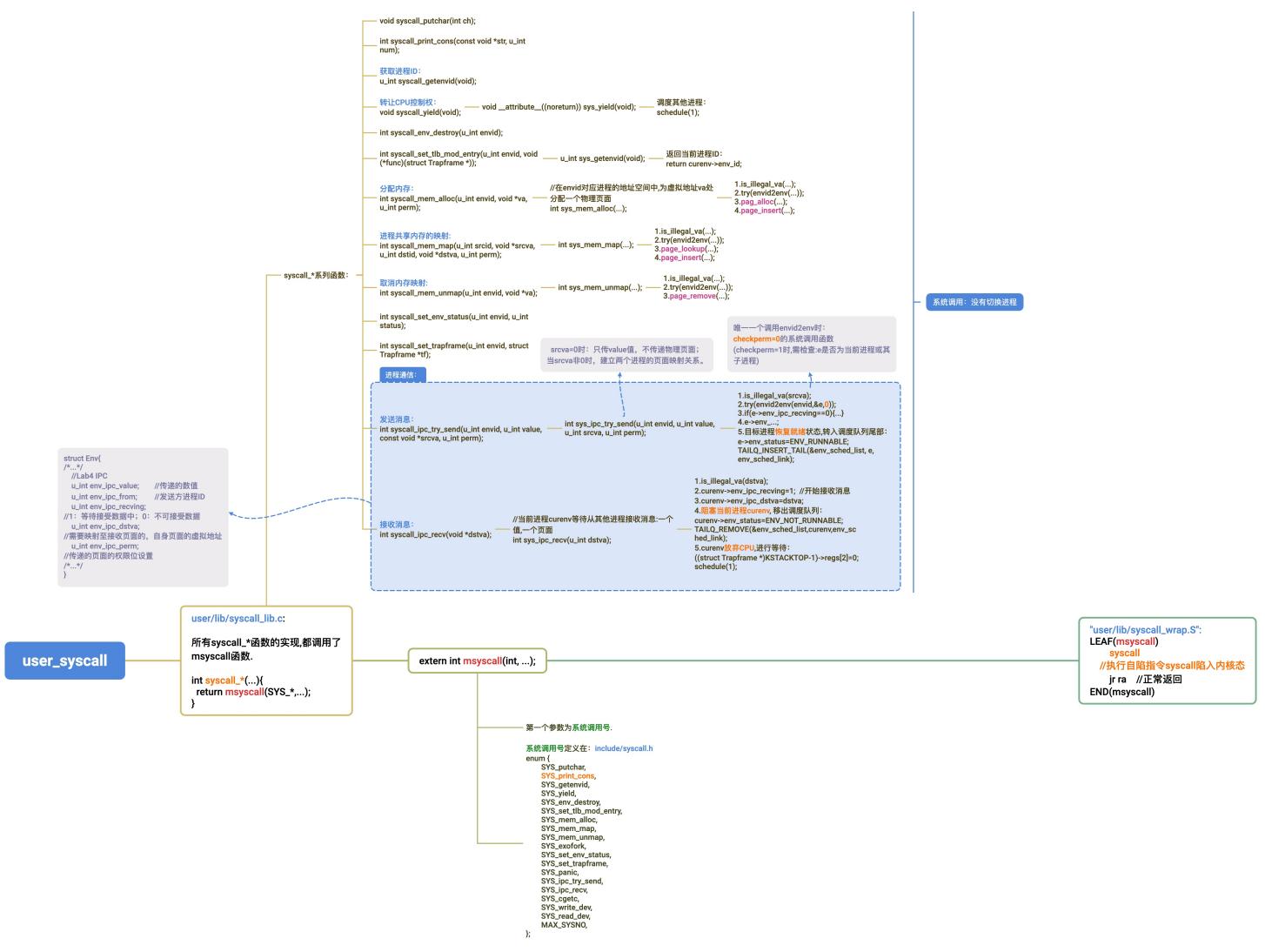
Notes

Mindmap

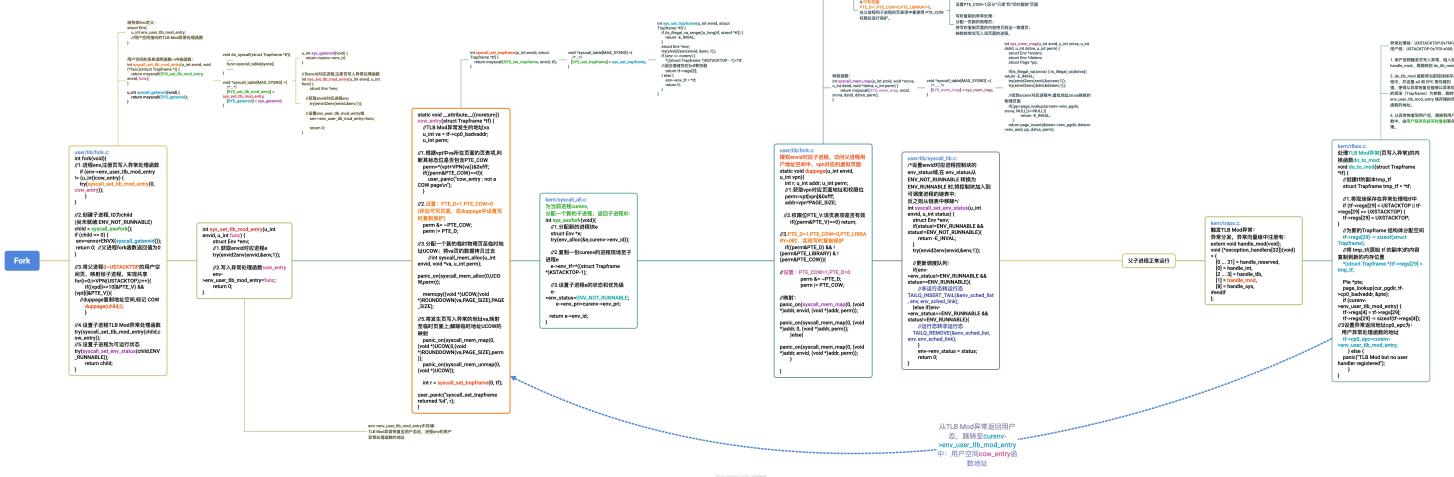
syscall过程



用户的系统调用函数(syscall_* 系列函数)



Fork函数



一些概念

用户态&内核态

- CPU访问权限：在内核态，访问任意区域；在用户态，只能访问用户空间。
- 映射机制：内核空间(kseg0,kseg1)，直接映射；用户空间(kuseg)，页表映射。

系统调用函数

用户空间中 `syscall_*` 函数(定义在 `user/lib/syscall_lib.c` 中)，与内核中的系统调用函数 `sys_*` 的函数(定义在 `kern/syscall_all.c` 中)是一一对应的.

内核栈进行传参.

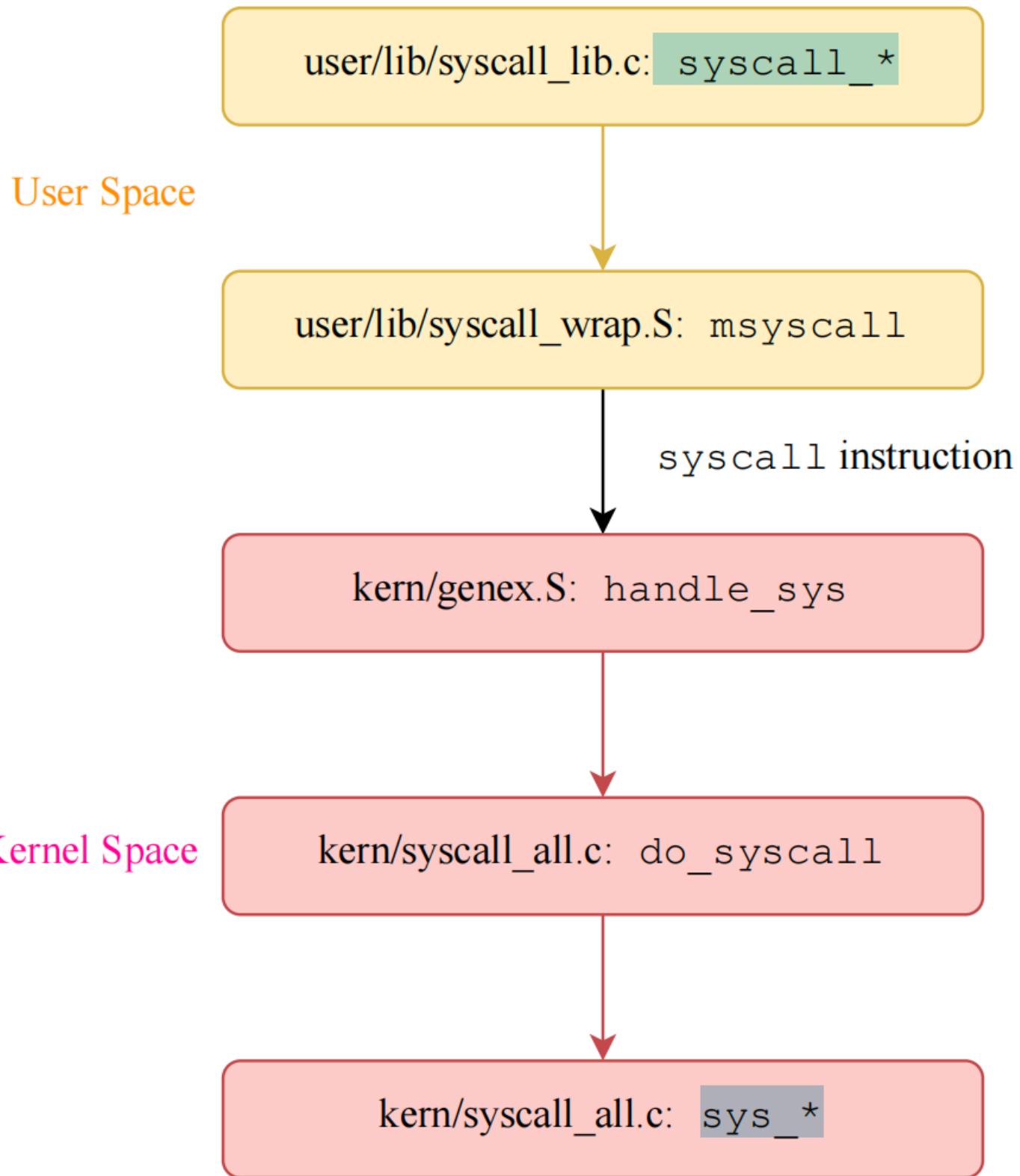


图 4.1: `syscall` 过程流程图

一个进程在调用 `fork()` 函数后，将从此分叉成为两个进程运行：

新的进程中，这一 `fork()` 调用的返回值为 `0`；父进程中，同一调用的返回值是子进程的 `env_id`。

名为 `exec` 的一系列系统调用：使得进程抛弃现有的程序和运行现场，执行一个新的程序。

- `fork` 之后，父子进程同时开始执行 `fork` 之后的代码段。
- `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。

头文件

include/mmu.h :地址部分

```
#define PDMAP (4 * 1024 * 1024) // 一个页面目录项映射的空间字节数  
#define ULIM 0x80000000 //kseg0起始地址  
#define UVPT (ULIM - PDMAP)
```

include/mmu.h :权限位部分

```
//全局位:若为1,TLB仅通过虚页号匹配页表项,不匹配ASID(进程号)  
#define PTE_HARDFLAG_SHIFT 6  
#define PTE_G (0x0001 << PTE_HARDFLAG_SHIFT)  
  
//有效位:若为1,该页表项有效,高20位是对应的物理页号;若为0,出现TLB miss  
#define PTE_V (0x0002 << PTE_HARDFLAG_SHIFT)  
  
//可写位:若为1,允许经由该页表项对物理页进行写作  
#define PTE_D (0x0004 << PTE_HARDFLAG_SHIFT)  
  
//写时复制位:(在TLB Mod 的异常处理函数中),若触发该异常的页面的PTE_COW为1,则:  
//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,再映射给尝试写入该页面的进程。  
#define PTE_COW 0x0001  
  
//共享权限位:若为1,则该页面可共享给子进程  
#define PTE_LIBRARY 0x0002
```

user/include/lib.h :

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

include/syscall.h :系统调用号

```
#ifndef SYSCALL_H
#define SYSCALL_H

#ifndef __ASSEMBLER__


//系统调用号的宏定义:
//msyscall函数的第一个参数
enum {
    SYS_putchar,
    SYS_print_cons,
    SYS_getenvid,
    SYS_yield,
    SYS_env_destroy,
    SYS_set_tlb_mod_entry,
    SYS_mem_alloc,
    SYS_mem_map,
    SYS_mem_unmap,
    SYS_exofork,
    SYS_set_env_status,
    SYS_set_trapframe,
    SYS_panic,
    SYS_ipc_try_send,
    SYS_ipc_recv,
    SYS_cgetc,
    SYS_write_dev,
    SYS_read_dev,
    MAX_SYSNO,
};

#endif

#endif
```

include/env.h :

```
#define LOG2NENV 10
#define NENV (1 << LOG2NENV) //最大进程数: 1024个
//生成进程标识符(envid):
//将envid限制在[0,NENV-1](即[0,1023])的范围内
#define ENVX(envid) ((envid) & (NENV - 1))

// All possible values of 'env_status' in 'struct Env'.
#define ENV_FREE 0
```

```
#define ENV_RUNNABLE 1
#define ENV_NOT_RUNNABLE 2

struct Env{
/*...*/
    //Lab4 IPC
    u_int env_ipc_value;          //进程传递的数值
    u_int env_ipc_from;          //发送方的进程ID
    u_int env_ipc_recving;       //1: 等待接受数据中; 0: 不可接受数据
    u_int env_ipc_dstva;         //需要映射至接收的页面的, 自身页面的虚拟地址
    u_int env_ipc_perm;          //传递的页面的权限位设置
/*...*/
}
```

函数

用户部分

`user/lib/syscall_lib.c` :`syscall_*`函数是执行系统调用的自陷指令

范式:

```
int syscall_(...){
    return msyscall(SYS_,...);
}
```

例子:

- `u_int syscall_getenvid(...);`
- `int syscall_set_tlb_mod_entry(...);`
- `int syscall_mem_alloc(...);`
- `int syscall_mem_map(...);`
- `int syscall_mem_unmap(...);`
-

`user/lib/fork.c` :

`int fork(void);`

```
int fork(void) {
    u_int child;
    u_int i;
```

```

/* Step 1: Set our TLB Mod user exception entry to 'cow_entry' if not done
yet. */
//为envid对应进程,注册页写入异常处理函数
//int sys_set_tlb_mod_entry(u_int envid, u_int func); 0对应curenv
if (env->env_user_tlb_mod_entry != (u_int)cow_entry) {
    try(syscall_set_tlb_mod_entry(0, cow_entry));
}

/* Step 2: Create a child env that's not ready to be scheduled. */
// Hint: 'env' should always point to the current env itself, so we should fix
it to the
// correct value.
child = syscall_exofork();
if (child == 0) {
    env = envs + ENVX(syscall_getenvid());
    return 0;
}

/* Step 3: Map all mapped pages below 'USTACKTOP' into the child's address
space. */
// Hint: You should use 'duppage'.
//static void duppage(u_int envid, u_int vpn);
/* Exercise 4.15: Your code here. (1/2) */
//i为虚拟页号:前10位为页目录偏移量,后10位为页表偏移量
for(i=0;i<VPN(USTACKTOP);i++){
    if((vpd[i]>>10)&PTE_V) && (vpt[i]&PTE_V){
        duppage(child,i);
    }
}

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

```

内核部分

kern/syscall_all.c :

分配内存: int sys_mem_alloc(u_int envid, u_int va, u_int perm);

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

    //1. 检查地址va的合法性
    if(is_illegal_va(va)) return -E_INVAL;

    //2. 获取envid对应的进程控制块
    try(envid2env(envid, &env, 1));

    //3. page_alloc函数: 分配空闲物理页pp
    try(page_alloc(&pp));

    //4. page_insert函数: 实现va到物理页pp的映射
    return page_insert(env->env_pgd, env->env_asid, pp, va, perm);
}
```

进程共享内存的映射:int sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm);

```
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;

    //1. 检查地址srcva,dstva的合法性
    if(is_illegal_va(srcva) || is_illegal_va(dstva)) return -E_INVAL;

    //2. 获取srcid对应进程块
    try(envid2env(srcid, &srcenv, 1));
    //3. 获取dstid对应进程块
    /* Exercise 4.5: Your code here. (3/4) */
    try(envid2env(dstid, &dstenv, 1));

    //4. 找到srcenv对应进程中,虚拟地址srcva映射的物理页面pp
    if((pp=page_lookup(srcenv->env_pgd, srcva, NULL))==NULL){
        return -E_INVAL;
    }
    //5. 在dstenv对应进程中,将dstva映射至物理页pp
    return page_insert(dstenv->env_pgd, dstenv->env_asid, pp, dstva, perm);
}
```

取消内存映射: int sys_mem_unmap(u_int envid, u_int va);

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

    //1. 检查地址va的合法性
    if(is_illegal_va(va)) return -E_INVAL;
    //2. 获取envid对应的进程块e
    try(envid2env(envid,&e,1));
    //3. 取消映射
    page_remove(e->env_pgdir, e->env_asid, va);
    return 0;
}
```

转让CPU控制权: void sys_yield(void);

```
//实现用户进程对 CPU 的放弃，从而调度其他的进程。
void __attribute__((noreturn)) sys_yield(void) {
    schedule(1);
}
```

接收消息: int sys_ipc_recv(u_int dstva);

```
//当前进程curenv等待从其他进程接收消息:一个值,一个页面
int sys_ipc_recv(u_int dstva) {
    //1. 检查地址dstva的合法性
    if (dstva != 0 && is_illegal_va(dstva)) {
        return -E_INVAL;
    }

    //2. 设为接收数据的状态
    curenv->env_ipc_recving=1;

    //3. 赋值:之后要将dstva映射至, 接收到的页面
    curenv->env_ipc_dstva=dstva;

    //4. 阻塞当前进程curenv, 移出调度队列
    curenv->env_status=ENV_NOT_RUNNABLE;
    TAILQ_REMOVE(&env_sched_list,curenv,env_sched_link);

    //5. curenv放弃CPU, 等待发送数据
    ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0;
    schedule(1);
}
```

发送消息：int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm);

srcva =0时，表示只传 value 值，而不传递物理页面；
当 srcva 不为0时，才建立两个进程的页面映射关系。

```
//将值value,一个页面发送给目标进程块envid
int sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm) {
    struct Env *e;           //e为目标进程(通过envid获取)
    struct Page *p;

    //1.检查地址srcva的合法性
    if (srcva != 0 && is_illegal_va(srcva)){
        return -E_INVAL;
    }

    //2.获取envid对应的进程块e
    /*sys_ipc_try_send函数是唯一一个调用envid2env时无须检查的系统调用函数(设置checkperm=0)
    (checkperm=1时,需检查:e是否为当前进程或其子进程)
    但sys_ipc_try_send用于两进程通信,无上述要求*/
    try(envid2env(envid,&e,0));

    //3.检查:目标进程是否处于接收消息的状态
    if(e->env_ipc_recv==0) return -E_IPC_NOT_RECV;
    //4.填入数据, 传递页面的映射关系
    e->env_ipc_value = value;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_perm = PTE_V | perm;
    e->env_ipc_recv = 0;      //结束接收状态

    //5.目标进程恢复就绪状态,转入调度队列尾部
    e->env_status=ENV_RUNNABLE;
    TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);

    //6.将发送方进程curenv中srcva对应的页面,映射至接收方进程e中env_ipc_dstva对应的页面
    if (srcva != 0){
        /* Exercise 4.8: Your code here. (8/8) */
        //①找到srcva对应的页面
        if((p=page_lookup(cur_pgdir,srcva,NULL))==NULL){
            return -E_INVAL;
        }
        //②将目标进程e的地址dstva,映射至页面p
        return page_insert(e->env_pgdir,e->env_asid,p,e->env_ipc_dstva,e-
>env_ipc_perm);
    }
    return 0;
}
```

```
}
```

kern/env.c :

从id获取进程块： int envid2env(u_int envid, struct Env **penv, int checkperm);

```
//通过进程标识符envid, 获取对应的进程控制块e, 其地址存储在*penv中
int envid2env(u_int envid, struct Env **penv, int checkperm){
    struct Env *e;

    //1. 通过envid, 在数组中找到对应进程控制块e
    e=&envs[ENVX(envid)];
    if(envid==0){           //当前控制块curenv
        *penv=curenv;
        return 0;
    }
    if (e->env_status == ENV_FREE || e->env_id != envid){
        return -E_BAD_ENV;
    }
    //2. checkperm=1时, 检查当前进程curenv是否有足够权限操作进程e
    //(e必须是curenv, 或者其子进程)
    if(checkperm){
        if(e!=curenv&&e->env_parent_id!=curenv->env_id) return -E_BAD_ENV;
    }

    //3. 存储e
    *penv=e;
    return 0;
}
```