> 常用的判断以及返回值
>
> `is_illegal_va(va) return -E_INVAL;` 虚拟地址是否为合法的用户虚拟地址
>
> `envid2env(envid, &env, 1) != 0 return -E_BAD_ENV` 查询的进程不是当前进程可以访问的 env（`checkperm` 为 0 时不检查是否为当前进程或者子进程）
>
> `!(e -> env_ipc_recving) return -E_IPC_NOT_RECV` 表示目标进程不接受信息

系统调用

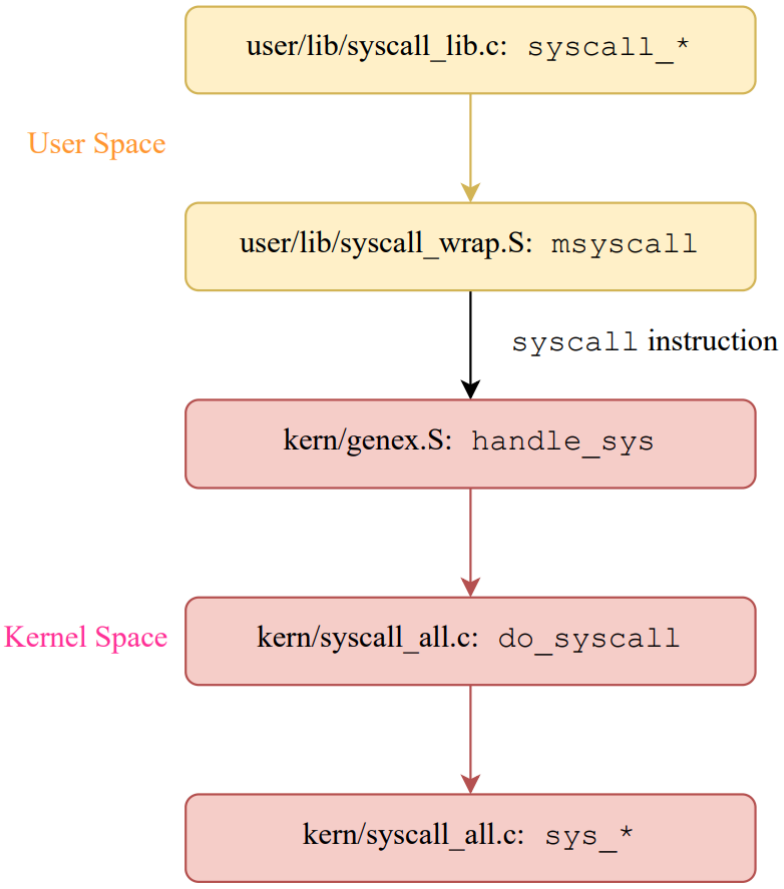
MOS已有系统调用函数表（不包括没学的）

函数	功能
`sys_putchar(int c)`	打印一个字符到屏幕
`sys_print_cons(const void *s, u_int num)`	打印给定长度的字符串到屏幕
`sys_getenvid(void)`	获取当前进程的进程号 `env_id`
`sys_yield(void)`	放弃当前进程剩余CPU时间片，另一个进程将被调度，这个函数永远不会返回
`sys_env_destroy(u_int envid)`	用于销毁进程控制块，被销毁的进程必须是调用者的子进程或半身
`sys_set_tlb_mod_entry(u_int envid, u_int func)`	注册 `envid` 用户空间的 TLB Mod 处理程序入口，设置为 `func`
`sys_mem_alloc(u_int envid, u_int va, u_int perm)`	申请一个物理页框并使用权限 `perm` 将 `envid` 的地址空间中的虚拟地址 `va` 映射到该物理页框，如果 `va` 已经被映射，那么原始页面被静默取消映射
`sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm)`	在 `srcid` 对应的进程地址空间中找到映射到 `srcva` 的物理页框，然后将 `dstid` 对应的进程地址空间中的 `dstva` 使用 `perm` 权限映射到该物理页框
`sys_mem_unmap(u_int envid, u_int va)`	取消 `envid` 对应进程地址空间中 `va` 到物理页框的映射，如果没有对应的映射则该函数默认成功

函数	功能
`sys_exofork(void)`	申请一个新的进程作为当前进程的子进程，如果成功返回子进程的进程号（该系统调用在用户空间的 `fork` 函数和 `spawn` 函数中是重要的一步）
`sys_set_env_status(u_int envid, u_int status)`	设置 `envid` 对应进程的进程状态并更新 `env_sched_list`
`sys_set_trapframe(u_int envid, struct Trapframe *tf)`	设置进程的陷入帧
`sys_ipc_recv(u_int dstva)`	等待从另一个进程过来的消息，包括一个值和一个页面如果 `dstva` 不是 0，当前进程将被阻塞直到消息送达
`sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm)`	尝试发送一个值以及一个页面如果 `srcva` 非零，到 `envid` 对应的进程，对 `envid` 进程中的结构体进行修改，并更新 `env_sched_list`
`sys_cgetc(void)`	读取一个字符，内核处于忙等状态，阻塞所有进程

异常分发向量组的 **8 号异常**就是操作系统处理系统调用时的异常

```
user/lib/debugf.c:
    [ debugf() --解析参数为字符串--> debug_output() --> ] syscall_*() --> msyscall() --陷入内核态，异常分发--> handle_sys() --传递系统调用所需要的信息--> sys_*() --> 回到用户态
```



`syscall_*` 函数与系统调用函数（`sys_*`）一一对应

基础系统调用函数

- `sys_mem_alloc(u_int envid, u_int va, u_int perm)` 为用户空间分配内存
- `sys_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm)` 将源进程地址空间中相应的内存映射到目标进程的相应地址空间的相应虚拟内存中去（二者共享一页物理内存）
- `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
- `sys_yield(void)` 实现用户进程对CPU的放弃，从而调度其他进程

进程间通信机制（IPC）

> 通过内核这一块公共空间来进行传递

Env

```
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; // 1：等待接受数据中；0：不可接收数据
    u_int env_ipc_dstva; // 接收到的页面需要与自身的哪个虚拟页面完成映射
    u_int env_ipc_perm; // 传递的页面的权限位设置
}
```

相关函数

- `sys_ipc_recv(u_int dstva)`：接收消息（当前进程接收）
- `sys_ipc_try_send(u_int envid, u_int value, u_int srcva, u_int perm)` 发送消息（当前进程发送，如果 `srcva` 不是空的话，则连同 `value` 所在的页一起发送，完成映射）

Fork

Fork 机制用于让一个进程创建一个进程

(a) 其中新产生的进程称为原进程的**子进程**。

i. 子进程开始运行时的大部分上下文状态与原进程相同

A. 包括程序和 `fork` 运行时的现场（包括通用寄存器和程序计数器 PC 等）。

ii. 在子进程中，`fork()` 调用的返回值为 `0` 。

(b) 其中旧的进程称为子进程的**父进程**。

i. 在父进程中，`fork()` 调用的返回值为子进程的 `env_id` 。

A. `env_id` 一定大于 `0` 。

B. `fork` 失败的情况下，子进程不会被创建，且父进程将得到小于 `0` 的返回值。

- `fork` 是 Linux 操作系统中创建新进程最主要的方式。相比于独立开始运行两个进程，父子进程间的通信要方便得多。因为子进程中仍然能读取原属于父进程的部分数据，父进程也可以根据 fork 返回的子进程 id，通过调用其他系统接口控制其行为
- `exec` 会使进程抛弃现有的程序和运行现场，执行一个新的程序
- `fork-exec` 是 fork 的一种常见的应用，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

写时复制机制：使用 `PTE_COW = 1` 和 `PTE_D = 0` 来进行写时复制的实现

注意

- ENV 相关宏

```
#define LOG2NENV 10
#define NENV (1 << LOG2NENV) // env的最大个数
#define ENVX(envid) ((envid) & (NENV - 1))
```

- 1. 通过函数给定的 `envid` 得到 env 结构体的方法

```
try(envid2env(envid, &e, 0)); // 设置perm = 0, 不检查父子关系
```

2. 获取当前进程的 `env` 结构体

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

- 关于进程祖先关系判断

```
int is_father(u_int fa_envid, u_int ch_envid) {
    while (ch_envid != 0) {
        if (fa_envid == ch_envid) {
            return 1;
        }
        struct Env * env;
        envid2env(ch_envid, &env, 0);
        ch_envid = env->env_parent_id;
    }
    return 0;
}
```

- 在当前进程要调用 `schedule(1)` 进行让位时，需在之前使用 `((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0;` 对于返回值状态进行清零再返回
-

Reg	Name
0	zero
1	at
2-3	v0-v1
4-7	a0-a3
8-15	t0-t7

Reg	Name
24-25	t8-t9
16-23	s0-s7
26-27	k0-k1
28	gp
29	sp
30	s8/fp
31	ra

- 系统调用流程

  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` 为例归纳步骤：

  1. 在 $user/include/lib.h$ 中添加：

     ```
     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：

     ```c
     void syscall_func(u_int envid, u_int value, const void *srcva, u_int perm) {
         msyscall(SYS_func, envid, value, srcva, perm);
     }
     ```

  3. 在 $user/lib$ 中的使用 `user_lib_func` 函数的目标文件中编写实现该函数（注意在该函数过程中会调用 `syscall_func` 函数）

  4. 在 $include/syscall.h$ 中的 `enum` 的 `MAX_SYSNO` **前面**加上 `SYS_func,`

  5. 在 $kern/syscall\_all.c$ 的 `void *syscall_table[MAX_SYSNO]` 的**最后**加上 `[SYS_func] = sys_func,` （注意最后有逗号）

  6. 在 $kern/syscall\_all.c$ 的 `void *syscall_table[MAX_SYSNO]` 的**前面**具体编写实现函数

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

2024-1-exam

添加系统调用，实现信号量机制的逻辑代码，包括信号量打开、关闭和PV操作

```
/*
user/lib/ipc.c 实现用户态的封装函数
user/lib/syscall_lib.c 完善所有系统调用函数 syscall_*(), 均调用并传参给 msyscall
kern/syscall_all.c 中的 *syscall_table[] 记录了系统调用向量表, 定义了 msyscall
传递时用到的宏对应的处理函数 sys_*(), 完善sys_*()函数逻辑
*/

example
int sem_wait(int sem_id) {
    int r;
    // Lab 4-1-Exam: Your code here. (1/9)
    // Implement process blocking
    while ((r = syscall_sem_wait(sem_id)) == 0) {
        syscall_yield();
    }
    if (r == -E_SEM_NOT_OPEN) return r;
    return 0;
}
int sys_sem_wait(int sem_id) {
    // Lab 4-1-Exam: Your code here. (7/9)
    if (sem_id < 0 || sem_id >= 15 || !sems_valid[sem_id]) {
        return -E_SEM_NOT_OPEN;
    }
    if (sems[sem_id] == 0) {
        return 0;
    }
    sems[sem_id]--;
    return sems[sem_id] + 1;
}
```

2024-2-exam

基于信号量机制的同步关系（用屏障来实现），实现子进程向父进程的通信

```
/** user/lib/strace.c **/

void strace_send(int sysno) {
    if (!((SYS_putchar <= sysno && sysno <= SYS_set_tlb_mod_entry) ||
          (SYS_exofork <= sysno && sysno <= SYS_panic)) ||
```

```
        sysno == SYS_set_trapframe) {
        return;
    }

    // Your code here. (1/2)
    if (straced != 0) {
        straced = 0;
        struct Env *child = envs + ENVX(syscall_getenvid());
        u_int parent = child->env_parent_id;
        ipc_send(parent, sysno, NULL, 0);
        syscall_set_env_status(0, ENV_NOT_RUNNABLE);
        straced = 1;
    }
}

void strace_recv() {
    // Your code here. (2/2)
    int flag = 1;
    u_int child;
    int sysno;
    while (flag) {
        if ((sysno = ipc_recv(&child, NULL, NULL)) == SYS_env_destroy) {
            flag = 0;
        }
        strace_barrier(child); // 等待子进程阻塞
        recv_sysno(child, sysno);
        syscall_set_env_status(child, ENV_RUNNABLE);
    }
}
```

2023-1-exam

> 附题目

加了一个限制，对于同组内的进程才可以互相通信，只需要将课下的 `sys_ipc_try_send` 复制粘贴，改一些东西，然后加一个判断就行

2023-2-exam

> 附代码

系统调用，实现一个“屏障”(barrier）同步原语，当有一组进程（或环境）到达屏障点时，只有最后一个到达的才被放行，其它都被挂起，直到屏障重置

2023-1-extra

仿照 `ipc_send` 函数在 `user/lib/ipc.c` 中实现 `ipc_broadcast` 函数，使得调用 `ipc_broadcast` 可以使当前进程向其后代进程（也即当前广播进程的子进程、子进程的子进程、子进程的子进程的子进程...以此类推）发起广播消息，当后代进程进入 recv 后进行发送。

```
int is_father(u_int may_child, u_int may_father) {
    if (may_child == may_father)
        return 0;
    while (may_child != 0) {
        if (may_child == may_father)
            return 1;
        struct Env * e = envs + ENVX(may_child);
        may_child = e->env_parent_id;
    }
    return 0;
}


void ipc_broadcast(u_int val, void * srcva, u_int perm) {
    struct Env * childs[NENV];
    int child_num = 0;
    for (int i = 0; i < NENV; i++) {
        if (is_father(envs[i].env_id, env->env_id)) {
            childs[child_num++] = envs + i;
        }
    }

    int r;
    for (int i = 0; i < child_num; i++) {
        while ((r = syscall_ipc_try_send(childs[i]->env_id, val, srcva,
perm)) != 0) {
            if (r == -E_IPC_NOT_RECV)
                syscall_yield();
        }
    }
}
```

2023-2-extra

实现信号量机制

```
struct Sem {
    int value;
    int checkperm;
    struct Env * creater;
    u_int blocks[500];
    int top;
} sems[10];

int sem_num = 0;

int sys_sem_init(int init_value, int checkperm) {
//  printk("init success\n");
    int sem_id = sem_num++;
    if (sem_id >= 10)
        return -E_NO_SEM;
    sems[sem_id].value = init_value;
    sems[sem_id].checkperm = checkperm;
    sems[sem_id].creater = curenv;
    sems[sem_id].top = -1;
//  printk("id=%d, value=%d\n", sem_id, sems[sem_id].value);
    return sem_id;

}

int is_father(u_int fa_envid, u_int ch_envid) {
    while (ch_envid != 0) {
        if (fa_envid == ch_envid) {
            return 1;
        }
        struct Env * env;
        envid2env(ch_envid, &env, 0);
        ch_envid = env->env_parent_id;
    }
    return 0;
}

int sys_sem_wait(int sem_id) {
//  printk("enter sem wait, sem id=%d\n", sem_id);
    if (sem_id < 0 || sem_id >= sem_num)
```

```
        return -E_NO_SEM;
    struct Sem * sem = sems + sem_id;
    if (sem->checkperm && !is_father(sem->creater->env_id, curenv->env_id)) {
//      printk("error in wait!!\n");
        return -E_NO_SEM;
    }

    if (sem->value > 0) {
        sem->value--;
        return 0;
    } else if (sem->value == 0) {
//      printk("value=%d\n", sem->value);
        sem->blocks[++sem->top] = curenv->env_id;
        curenv->env_status = ENV_NOT_RUNNABLE;
        TAILQ_REMOVE(&env_sched_list, curenv, env_sched_link);
        ((struct Trapframe *)KSTACKTOP - 1)->regs[2] = 0;
        schedule(1);
    }
    return 0;
}

int sys_sem_post(int sem_id) {
    if (sem_id < 0 || sem_id >= sem_num)
        return -E_NO_SEM;
    struct Sem * sem = sems + sem_id;
    if (sem->checkperm && !is_father(sem->creater->env_id, curenv->env_id)) {
        return -E_NO_SEM;
    }
//  printk("enter post\n");
//  printk("sem val=%d", sem->value);
    sem->value++;
//  printk("sem val=%d", sem->value);
    if (sem->top >= 0) {
        u_int envid = sem->blocks[sem->top--];
        struct Env * e;
        sem->value--;
        envid2env(envid, &e, 0);
        e->env_status = ENV_RUNNABLE;
        TAILQ_INSERT_TAIL(&env_sched_list, e, env_sched_link);
    }
    return 0;
```

```
}

int sys_sem_getvalue(int sem_id) {
    if (sem_id < 0 || sem_id >= sem_num)
        return -E_NO_SEM;
    struct Sem * sem = sems + sem_id;
    if (sem->checkperm && !is_father(sem->creater->env_id, curenv-
>env_id)) {
        return -E_NO_SEM;
    }

    return sem->value;
}
```

2024-1-extra

实现一个消息队列，发消息的进程往里边放信息，获取消息的进程从里边取信息

```
int sys_msg_send(u_int envid, u_int value, u_int srcva, u_int perm) {
    struct Env *e;
    struct Page *p;
    struct Msg *m;

    if (srcva != 0 && is_illegal_va(srcva)) {
        return -E_INVAL;
    }
    try(envid2env(envid, &e, 0));
    if (TAILQ_EMPTY(&msg_free_list)) {
        return -E_NO_MSG;
    }

    /* Your Code Here (1/3) */
    m = TAILQ_FIRST(&msg_free_list);
    m->msg_tier++;
    m->msg_status = MSG_SENT;
    TAILQ_REMOVE(&msg_free_list, m, msg_link);
    m->msg_from = curenv->env_id;
    m->msg_value = value;
    m->msg_perm = PTE_V | perm;
    if (srcva != 0) {
        p = page_lookup(curenv->env_pgdir, srcva, NULL);
        if (p == NULL) {
            return -E_INVAL;
```

```
        }
        p->pp_ref++;
        m->msg_page = p;
    } else {
        m->msg_page = NULL;
    }
    TAILQ_INSERT_TAIL(&(e->env_msg_list), m, msg_link);
    return msg2id(m);

}

int sys_msg_recv(u_int dstva) {
    struct Msg *m;
    struct Page *p;

    if (dstva != 0 && is_illegal_va(dstva)) {
        return -E_INVAL;
    }
    if (TAILQ_EMPTY(&curenv->env_msg_list)) {
        return -E_NO_MSG;
    }

    /* Your Code Here (2/3) */
    m = TAILQ_FIRST(&(curenv->env_msg_list));
    TAILQ_REMOVE(&(curenv->env_msg_list), m, msg_link);
    if (m->msg_page != NULL && dstva != 0) {
        p = m->msg_page;
        try(page_insert(curenv->env_pgdir, curenv->env_asid, p, dstva, m-
>msg_perm));
        page_decref(p);
    } else if (m->msg_page != NULL) {
        page_decref(m->msg_page);
    }

    curenv->env_msg_value = m->msg_value;
    curenv->env_msg_from = m->msg_from;
    curenv->env_msg_perm = m->msg_perm;

    m->msg_status = MSG_RECV;
    TAILQ_INSERT_TAIL(&msg_free_list, m, msg_link);
    return 0;
}
```

```c
int sys_msg_status(u_int msgid) {
    struct Msg *m;

    /* Your Code Here (3/3) */
    m = &msgs[MSGX(msgid)];
    u_int id = msg2id(m);
    if (id == msgid) {
        return m->msg_status;
    } else if (id > msgid) {
        return MSG_RECV;
    } else {
        return -E_INVAL;
    }
}
```

2024-2-extra

实现进程创建，但父子进程完全共享内存，实现 `sys_clone()` 等函数