

Lab5 用户程序

一、实验原理

1 实验执行流程

一般在RISC-V下，用户程序通过**系统调用**借助ecall+sret在U/S态间进行切换。但在本次实验中，具体要解决的是如何从Lab4中创建的只有内核进程的S态进入用户进程U态。所以可以：

- 在S态主动构造一次异常/陷入（trap）的返回场景（手动设置好trap frame等上下文）
- 接着利用异常返回的机制（sret）把CPU从S态送到U态，让第一个用户程序开始执行

对整个实验流程进行一个大致说明：

一、内核启动阶段：只有内核线程/进程

根据Lab4，内核初始化后，会启动一个初始内核线程，此时先调用：

```
init_main(){
    kernel_thread(user_main);
}
/*
调用user_main时就创建了一个内核线程（仍在S模式、使用内核栈、内核地址空间）
把入口函数设置在user_main
```

二、在内核线程user_main里，准备“第一个用户进程”

此时user_main()任务是把一个用户进程拉起来当第一个用户进程运行。调用KERNEL_EXECVE ()，再往底层调用kernel_execve ()（在内核态伪造一次sys_exec）。如下列所示代码，并且用ebreak触发断点异常，用“a7 = 10”判断。

```
static int kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
    asm volatile(
        "li a0, %1\n"    // a0 = SYS_exec（系统调用号）----->注意此时将SYS_exec传入a0，下面调用内核态的syscall时还会用到。
        "lw a1, %2\n"    // a1 = name
        "lw a2, %3\n"    // a2 = len
        "lw a3, %4\n"    // a3 = binary
        "lw a4, %5\n"    // a4 = size
        "li a7, 10\n"    // a7 = 10，作为“特殊标记”----->用此处作为复用syscall框架做exec的标志。
        "ebreak\n"       // 触发断点异常（CAUSE_BREAKPOINT）
        "sw a0, %0\n"     // 把返回值写回 ret
        : "=m"(ret)
        : "i"(SYS_exec), "m"(name), "m"(len), "m"(binary), "m"(size)
        : "memory");
```

```

    cprintf("ret = %d\n", ret);
    return ret;
}

```

三、中断/异常入口: trapentry.S中的SAVE_ALL

触发ebreak / ecall 都会跳到stvec指定的入口 (_alltraps) 。

3.1 _alltraps入口

```

.globl __alltraps
__alltraps:
    SAVE_ALL --->此处根据trap的来源 (U/S) , 管理用户栈/内核栈 + 保存全部现场到
trapframe

    move a0, sp----->使用sp指向trapframe
    jal trap # C 语言的 trap 函数

```

3.2 SAVE_ALL:U栈/内核栈切换 + 保存寄存器

```

.macro SAVE_ALL
    LOCAL _restore_kernel_sp
    LOCAL _save_context

    # 交换 sp 和 sscratch
    csrrw sp, sscratch, sp
    # 如果 sp != 0, 说明之前是用户态, 此时 sp = 内核栈指针
    bnez sp, _save_context

_restore_kernel_sp:
    # 如果来自内核态 (sscratch = 0) , 上面 csrrw 后 sp = 0
    # 再从 sscratch 取回真正的内核栈指针
    csrr sp, sscratch
_save_context:
    # 在当前栈 (已是内核栈) 上分配 trapframe
    addi sp, sp, -36 * REGBYTES
    # 保存 x0~x31
    STORE x0, 0*REGBYTES(sp)
    ...
    STORE x31, 31*REGBYTES(sp)

    # 把 sscratch 的值取到 s0, 然后把 sscratch 清零
    csrrw s0, sscratch, x0
    csrr s1, sstatus
    csrr s2, sepc
    csrr s3, 0x143 # stval

```

```

        csrr s4, scause

# 把 s0 (如果来自用户态, 则是“用户栈指针”) 存到 trapframe 中的 x2 槽位
STORE s0, 2*REGBYTES(sp)
STORE s1, 32*REGBYTES(sp)
STORE s2, 33*REGBYTES(sp)
STORE s3, 34*REGBYTES(sp)
STORE s4, 35*REGBYTES(sp)
.endm

```

该部分的处理分为两种情况:

1、来自用户态U-mode

- 在进入trap前约定:
 - sp = 用户栈指针
 - sscratch = 内核栈指针
- 执行完 `csrrw sp, sscratch, sp` 后:
 - sp = 内核栈指针 (切换到内核栈)
 - sscratch = 用户栈指针。

此时sp 和sscratch二者的指向交换。

- bnez跳转到 `_SAVE_CONTEXT`
- 在内核栈(sp)上分配trapframe, 保存所有寄存器。
- 执行 `csrrw s0, sscratch, x0`
 - s0 = 用户栈指针
 - sscratch = 0
- 把s0 (用户栈指针) 存到trapframe的x2槽位, 将来RESTORE_ALL时恢复成用户sp

2、来自内核态S-mode

- 约定: sscratch = 0
- `csrrw sp, sscratch sp` 后:
 - sp=0
 - sscratch = 原来的内核sp
- 进入 `*_restore_kernel_sp*`
 - `csrr sp, sscratch` 恢复sp = 原内核指针
- 后续保存寄存器完全在内核栈上进行;
- `csrrw s0, sscratch, x0` 之后:
 - s0 = 原来的内核栈指针, sscratch = 0;
- trapframe 中 x2 槽位保存的是“中断前的内核 sp”。

于是，**SAVE_ALL同时完成了：**

- 若来自用户态：**切到内核栈 + 保存用户栈指针；**
- 若来自内核态：不切栈，仅保存现场；
- 把 `sscratch` 清零，以便后续再发生 trap 时能区分是否来自内核。

四、C 层 trap 处理：识别 ebreak / ecall、分发到 syscall

4.1 trap () ----->exception_handler()

trap () 会根据 `tf->cause` 调用 `exception_handler(tf)`，下列是针对 ebreak 和 ecall 的代码：

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_BREAKPOINT:
            cprintf("Breakpoint\n");
            if (tf->gpr.a7 == 10) { // 特殊标记
                tf->epc += 4;      // 返回时跳过 ebreak 那条指令
                syscall();        // 复用通用 syscall 分发
            }
            break;

        case CAUSE_USER_ECALL:
            tf->epc += 4;          // 用户态 ecall 情况
            syscall();            // 同样复用 syscall，只不过此时从 U 态出发。
            break;

        /* other cases ... */
    }
}
```

4.2 内核 syscall()

```
void syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];

    int num = tf->gpr.a0; // a0 = SYS_exec
    if (num >= 0 && num < NUM_SYSCALLS && syscalls[num] != NULL) {
        arg[0] = tf->gpr.a1; // name
        arg[1] = tf->gpr.a2; // len
        arg[2] = tf->gpr.a3; // binary
        arg[3] = tf->gpr.a4; // size
        arg[4] = tf->gpr.a5; // (没用)
        tf->gpr.a0 = syscalls[num](arg); // 调 sys_exec(arg)，返回值放回 a0
        return;
    }
    // 未实现的 syscall, panic
}
```

五、do_execve / load_icode: 为用户程序构造“将要运行的上下文”

```
int do_execve(const char *name, size_t len, unsigned char *binary, size_t size) {
    struct mm_struct *mm = current->mm;
    // 1. 校验 name; 复制到 local_name
    // 2. 如果已有 mm (老的用户地址空间), 释放之:
    //    - 切回 boot_cr3
    //    - mm_count_dec / exit_mmap / put_pgdir / mm_destroy
    //    - current->mm = NULL;

    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}
```

该部分 (`load_icode(binary, size)`) 主要负责:

- 创建一个新的 `mm_struct*mm` 和对应的页表
- 把用户程序的代码/数据段根据 ELF/二进制格式加载到用户地址空间;
- 为用户栈分配一块虚拟地址空间;
- 设置当前进程的 `trapframe` / 寄存器状态:
 - `tf->epc` = 用户程序入口地址
 - 修改 `sstatus` 中的 `SPP` 位为 0 (表示 trap 返回时回到 U-mode);
 - 设置通用寄存器 (比如 `sp` 指向用户栈顶、清零其他寄存器等)。

此时内核已经把“第一次运行用户程序时的上下文”伪装成“刚从用户态陷入内核后要返回的上下文”——也就是 `trapframe`。

六、trap返回: 第一次真正切到U-mode, 跑起第一个用户进程 (RESTORE_ALL + sret)

在 `trap()` / `exception_handler` / `syscall()` 完成后, 会回到 `*_trapret*`:

```
.globl __trapret
__trapret:
    RESTORE_ALL
    sret
```

6.1 RESTORE_ALL: 恢复 `sstatus/sepc` + 寄存器 + 栈

```

.macro RESTORE_ALL
    LOCAL _save_kernel_sp
    LOCAL _restore_context

    LOAD s1, 32*REGBYTES(sp)    # sstatus
    LOAD s2, 33*REGBYTES(sp)    # sepc

    andi s0, s1, SSTATUS_SPP    # 读 SPP 位
    bnez s0, _restore_context    # SPP != 0 => 返回 S 模式 (来自内核)

_save_kernel_sp:
    # 如果 SPP==0, 即来自用户态, 这次返回到 U 态,
    # 需要把当前内核栈指针“记到 sscratch”里
    addi s0, sp, 36 * REGBYTES
    csrw sscratch, s0
_restore_context:
    csrw sstatus, s1
    csrw sepc, s2

    # 恢复除 x2 外的通用寄存器
    LOAD x1, 1*REGBYTES(sp)
    ...
    LOAD x31, 31*REGBYTES(sp)

    # 最后恢复 x2 (sp)
    LOAD x2, 2*REGBYTES(sp)    # 来自用户态时, 这里就是用户栈指针
.endm

```

分两种来源：

1、若spp==0 (返回U态)

- 说明这次 trap 是从 U-mode 进来的 (或者第一次 exec 后要跳到 U 态)；
- `_save_kernel_sp`: 把“展开后”的内核栈顶地址 `sp + 36*REGBYTES` 写入 `sscratch`, 以便下次从 U 态陷入时能找到内核栈;
- 之后恢复 `sstatus/sepc`;
- 从 `trapframe` 里恢复所有寄存器, 包括 `x2 (sp)`:
 - 因为 `SAVE_ALL` 时把用户栈指针保存在 `2*REGBYTES(sp)`, 所以这里恢复后 `sp` 变成“用户栈指针”。
- 执行 `sret`
 - 根据 `sstatus.SPP = 0`, 硬件返回到 **U-mode**;
 - `PC = sepc`, 也就是 `load_icode` 里设置的“用户程序入口”。

至此, 第一次真正切到了 U 态, 开始执行用户程序 (如 `exit.c` 的 `main`) 。

2、若spp==1 (返回S态)

- 说明这次 trap 来源于 S 模式 (例如内核里的 `ebreak` 或某些异常)；
- 不执行 `_save_kernel_sp`, 不更新 `sscratch`;

- 仅恢复 sstatus/sepc 和所有寄存器;
- `sret` 后回到 S 模式的某条指令继续执行 (比如 `kernel_execve` 的 `ebreak` 后一条)。

七、用户态运行&系统调用 (包括exit)

7.1 用户态的syscall封装

用户态的syscall () :

```
static inline int syscall(int num, ...) {
    ...
    // a0 = num, a1~a5 = 参数
    asm volatile (
        "ld a0, %1\n"
        "ld a1, %2\n"
        "ld a2, %3\n"
        "ld a3, %4\n"
        "ld a4, %5\n"
        "ld a5, %6\n"
        "ecall\n"
        "sd a0, %0"
        : "=m" (ret)
        : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]), "m"(a[3]), "m"(a[4])
        : "memory");
    return ret;
}
```

- 在用户进程中, 库函数 (如 `write`, `exit` 等) 最终都会调用这个 `syscall()`;
- 执行 `ecall` 时, CPU 从 U-mode 陷入 S-mode:
 - 再次进入 `__alltraps` -> `SAVE_ALL`;
 - 切换到内核栈, 保存用户寄存器 (包括用户 `sp`) ;
 - `trap()` -> `exception_handler()` -> `CAUSE_USER_ECALL`;
 - 内核态 `syscall()` 根据 `a0` 查表到具体 `sys_xxx`。

7.2 exit系统调用&进程退出流程

用户库中的exit () :

```
// /user/libs/ulib.c
void exit(int error_code) {
    sys_exit(error_code);  ----->sys_exit是对syscall (SYS_exit,error_code) 的封装。
    cprintf("BUG: exit failed.\n");
    while (1);
}

//执行ecall后进入内核, 最终上述sys_exit会走到:
// /kern/syscall/syscall.c
static int sys_exit(uint64_t arg[]) {
```

```

int error_code = (int)arg[0];
return do_exit(error_code);
}

```

7.3 do_exit: 当前进程完成大部分资源回收

```

int do_exit(int error_code) {
    // 1. 特殊进程保护: idleproc / initproc 不能退出
    if (current == idleproc) panic("idleproc exit.\n");
    if (current == initproc) panic("initproc exit.\n");

    struct mm_struct *mm = current->mm;

    // 2. 如果 mm != NULL, 说明是“用户进程”, 需要释放用户地址空间
    if (mm != NULL) {
        lcr3(boot_cr3); // 切回内核页表, 确保可以安全访问内核结构

        if (mm_count_dec(mm) == 0) {
            // 没有别的进程共享这个 mm, 真正释放
            exit_mmap(mm); // 解除映射, 释放用户虚拟内存空间
            put_pgdir(mm); // 释放用户页表物理页等
            mm_destroy(mm); // 释放 mm 结构本身
        }
        current->mm = NULL; // 进程不再有用户地址空间
    }

    // 3. 把进程标记为“僵尸态”, 记录退出码
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;

    bool intr_flag;
    struct proc_struct *proc;

    local_intr_save(intr_flag);
    {
        // 4. 唤醒父进程 (如果它在等待子进程)
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);
        }

        // 5. 处理自己的子进程: 全部过继给 initproc
        while (current->cptr != NULL) {
            proc = current->cptr;
            current->cptr = proc->optr;

            // 重新挂到 initproc 的孩子链表上
            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
        }
    }
}

```



```

        proc->parent = initproc;
        initproc->cptr = proc;

        // 如果这些子进程也已经是 ZOMBIE, 则顺便唤醒 initproc
        if (proc->state == PROC_ZOMBIE) {
            if (initproc->wait_state == WT_CHILD) {
                wakeup_proc(initproc);
            }
        }
    }
}
local_intr_restore(intr_flag);

// 6. 调度, 下一个进程继续跑。当前进程不再返回到用户态
schedule();

panic("do_exit will not return!! %d.\n", current->pid);
}

```

- **进程自己不能释放“内核栈”和自己的 `proc_struct`**, 因为在执行 `do_exit` 时还在用内核栈、还需要 `current`;
 - 所以它只释放“用户态资源” (mm / 页表 / 用户空间虚拟内存), 把自己变成 ZOMBIE, 由父进程 (典型就是 `init_main`) 后续调用 `do_wait` 来完成最终回收。
- `do_exit` 结束时调用 `schedule()`, 当前进程不会再执行;
 - 将来父进程在 `do_wait` 中发现该子进程是 ZOMBIE, 就会释放它残留的资源 (包括内核栈和进程控制块)。

八、init进程等待并回收所有用户进程

回到最开始的 `init_main`:

```

while (do_wait(0, NULL) == 0) { // 循环等待子进程退出
    schedule();
}

cprintf("all user-mode processes have quit.\n");
// 一系列 assert, 确认:
// - 所有子进程都被正确回收;
// - 进程数 / 链表结构回到只剩 idleproc + initproc 的状态;
// - 空闲页数和内核分配计数恢复到最初值。

```

- 所有用户进程 (包括第一批通过 `user_main` 启动的 `exit` 之类) 都跑完后:
 - 每个用户进程调用 `exit()`;
 - 进入内核 `sys_exit` -> `do_exit`, 变成 ZOMBIE;
 - `init_main` 在 `do_wait` 中检测到子进程退出, 回收其内核栈和 `proc_struct`;
- 最终, 只剩 `idleproc + initproc`, 系统内存状态应该恢复到进入用户态前的水位 (除少量内核持久数据结构外)

这部分属于实验中最后的“检查点”：**确保“用户空间跑一圈之后，内核依然干净”。**

2 用户进程

该部分主要改动是在内核的proc.c部分代码上。

这是新的proc.c部分代码的内容：

```
// kern/process/proc.c (lab5)。

/*lab5中的init_main变成了真正的init进程。创建user_main作为用户态的入口，循环等待并回收
所有用户进程，最后检查- 进程列表和内存是否恢复到预期状态，确保“用户空间跑一圈之后内核依然
干净”。
*/

static int init_main(void *arg) {
    size_t nr_free_pages_store = nr_free_pages();
    size_t kernel_allocated_store = kallocated();

    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0) {
        panic("create user_main failed.\n");
    }

    while (do_wait(0, NULL) == 0) { //循环等待子进程退出
        schedule();
    }

    cprintf("all user-mode processes have quit.\n");
    assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->optr ==
    NULL); //检查initproc的三个子进程指针，从而检查所有子进程都被回收，
    assert(nr_process == 2); //检查进程总数和链表结构（只剩initproc自己为止）。
    assert(list_next(&proc_list) == &(initproc->list_link));
    assert(list_prev(&proc_list) == &(initproc->list_link));

    cprintf("init check memory pass.\n");
    return 0;
}
```

接着探究如何通过user_main从内核态跳转回用户态呢？

根据下列代码探究：

实现user_main的基础：

```
// kern/process/proc.c
/*下列几个宏都是实现从内核态跳转到用户态的主要工具。可以一个一个查看分析。
但主要都是基于内核函数“kernel_execve(name,binary,(size_t)(size))”的。这个函数根据传入
的二进制镜像binary和长度size，在当前进程上下文中加载这个程序，替换原有的地址空间，设置入
口等。最终跳转到用户态代码执行。
*/
```

```
//第一个"_KERNEL_EXECVE(name,binary,sieze)", 打印日志之后直接调用“kernel_execve()”。
#define __KERNEL_EXECVE(name, binary, size) ({
    cprintf("kernel_execve: pid = %d, name = \"%s\".\n",
        current->pid, name);
    kernel_execve(name, binary, (size_t)(size));
})

//“_KERNEL_EXECVE(x)”使用编译器自动生成的关于用户程序x的二进制数据的起始地址+二进制数据的长度，调用第一个宏，嵌套+嵌套。
#define KERNEL_EXECVE(x) ({
    extern unsigned char _binary_obj__user_##x##_out_start[], \
        _binary_obj__user_##x##_out_size[];
    __KERNEL_EXECVE(#x, _binary_obj__user_##x##_out_start, \
        _binary_obj__user_##x##_out_size);
})

//“_KERNEL_EXECVE2(x,xstart,xsize)”与“_KERNEL_EXECVE(x)”差不多，只不过比他更灵活，不用强制使用二进制命名，可以自己提供符号名等。当使用自己定义的名字时就是用该宏。
#define __KERNEL_EXECVE2(x, xstart, xsize) ({
    extern unsigned char xstart[], xsize[];
    __KERNEL_EXECVE(#x, xstart, (size_t)xsize);
})

#define KERNEL_EXECVE2(x, xstart, xsize)    __KERNEL_EXECVE2(x, xstart, xsize)
```

user_main:

```
// user_main - kernel thread used to exec a user program
static int
user_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE); //如果使用了宏定义TEST，那么调用
    KERNEL_EXECVE2
#else
    KERNEL_EXECVE(exit);
#endif
    panic("user_main execve failed.\n");
}
```

总的来说，整个过程中：`KERNEL_EXECVE(x)` 利用自动生成的 `_binary_obj__user_x_out_start/_size` 这些符号，得到**嵌入在内核中的用户程序 x 的二进制镜像和长度**，然后和程序名 "x" 一起传给 `kernel_execve`。

`kernel_execve` 再根据这份二进制镜像，在新的用户态地址空间中完成加载（解析 ELF、建立映射、设置入口地址等），最后切换到用户态入口处执行。

3 系统调用的实现

用户态如何获取内核态的服务呢？也就是把需求放到内核态实现呢？就要用到**系统调用**。

为了让用户态能够进行系统调用，那就要先让用户态陷入内核，首先存在两个不同状态下的syscall函数：

- 用户态syscall:

```
// 用户态 syscall()
static inline int syscall(int num, ...) {
    ...
    // a0 = num, a1~a5 = 参数
    asm volatile (
        "ld a0, %1\n"
        "ld a1, %2\n"
        "ld a2, %3\n"
        "ld a3, %4\n"
        "ld a4, %5\n"
        "ld a5, %6\n"
        "ecall\n"
        "sd a0, %0"
        : "=m" (ret)
        : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]), "m"(a[3]), "m"(a[4])
        : "memory");
    return ret;
}
/*用户态的syscall中，在执行ecall之前，做了：
1、a0 = num(系统调用号)
2、a1.....a5 = 参数1 .....参数5
*/
/*整个过程的语义：把系统调用号num和至多5个参数装进RISC-V规定的寄存器，执行ecall，然后把内核放到a0的返回值取回，作为函数返回。*/
```

- 内核态syscall:

```
void syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0; //a0寄存器保存了系统调用编号
    if (num >= 0 && num < NUM_SYSCALLS) { //防止syscalls[num]下标越界
        if (syscalls[num] != NULL) {
            arg[0] = tf->gpr.a1;
            arg[1] = tf->gpr.a2;
            arg[2] = tf->gpr.a3;
            arg[3] = tf->gpr.a4;
            arg[4] = tf->gpr.a5;
            tf->gpr.a0 = syscalls[num](arg);
            //把寄存器里的参数取出来，转发给系统调用编号对应的函数进行处理
            return ;
        }
    }
    //如果执行到这里，说明传入的系统调用编号还没有被实现，就崩掉了。
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
```

```

        num, current->pid, current->name);
    }
    /*内核态的ecall函数刚好相反:
    int num = tf->gpr.a0; // 取出系统调用号
    arg[0] = tf->gpr.a1; // 参数1
    arg[1] = tf->gpr.a2; // 参数2
    arg[2] = tf->gpr.a3;
    arg[3] = tf->gpr.a4;
    arg[4] = tf->gpr.a5;
    然后根据num (系统调用号) 在syscalls[]里查找对应的处理函数。
    tf->gpr.a0 = syscalls[num](arg);把返回值num写回a0,a
    */

```

二、实验练习

练习一：加载应用程序并执行（需要编程）

这里需要补充load_icode的第6步代码，补充代码如下

```

tf->gpr.sp = USTACKTOP; //将用户态堆栈指针初始化为为用户栈顶，确保用户代码有有效的栈。
tf->epc = elf->e_entry;
tf->status = sstatus & ~SSTATUS_SPP;//使SPP=0，保证执行 sret 时会切换到用户态。
tf->status |= SSTATUS_SPIE; //从内核返回到用户态后应允许用户态中断

```

问题：请简要描述这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

答：

1. 调度器选择进程：某个可运行进程被调度器选中，进入准备运行阶段。
2. 切换到该进程的地址空间：调用 proc_run(), 内核先禁中断、设置 current = proc, 然后用 Isatp(proc->pgdir) 将 SATP (页表基) 切换到该进程的 page table, 从而启用该进程的用户虚拟内存视图。
3. 内核上下文切换：proc_run() 调用 switch_to(&prev->context, &proc->context), 在内核态保存前一个内核上下文并恢复目标进程的内核上下文。
4. 在新进程的内核上下文中继续执行：context 切换后 CPU 在新进程的内核函数 forkret() 处运行；forkret() 调用 forkrets(current->tf) (负责从 trapframe 恢复到寄存器并执行返回用户态的指令)。
5. 使用 trapframe 恢复用户寄存器：之前在 load_icode() 中已准备好 current->tf, 其中关键字段包括 tf->gpr.sp (用户栈指针, 设为 USTACKTOP)、tf->epc (用户入口地址, 设为 ELF 的 e_entry)、tf->status (sstatus 的适当值: SPP 清为用户态、SPIE 置位, 保证 sret 后中断行为正确)。
6. 执行特权级返回：forkrets (或底层的 trap 返回代码) 把 tf 的寄存器值写回 CPU 状态寄存器 (把 tf->status 写回 sstatus、tf->epc 写回 sepc、把通用寄存器恢复或装载), 然后执行 sret (或等价恢复指令), 该指令依据 SPP/SPIE 切换到用户态并恢复中断使能。
7. 用户态开始执行：sret 完成特权级切换后, CPU 进入用户特权级并跳转到 sepc (即 tf->epc, ELF 的入口), 使用 tf->gpr.sp 作为用户栈, 随后用户程序开始执行其第一条指令。

练习二：父进程复制自己的内存空间给子进程（需要编码）

创建子进程的函数do_fork在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过copy_range函数（位于kern/mm/pmm.c中）实现的，请补充copy_range的实现，确保能够正确执行。

请在实验报告中简要说明你的设计实现过程。

如何设计实现Copy on Write机制？给出概要设计，鼓励给出详细设计。

- Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

扩展练习Challenge：1.实现 Copy on Write（COW）机制

给出实现源码、测试用例和设计报告（包括在cow情况下的各种状态转换（类似有限状态自动机）的说明）。

这个扩展练习涉及到本实验和上一个实验“虚拟内存管理”。在ucore操作系统中，当一个用户父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进程占用的用户内存空间中的页面（这就是一个共享的资源）。当其中任何一个进程修改此用户内存空间中的某页面时，ucore会通过page fault异常获知该操作，并完成拷贝内存页面，使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请在ucore中实现这样的COW机制。

由于COW实现比较复杂，容易引入bug，请参考 <https://dirtycow.ninja/> 看看能否在ucore的COW实现中模拟这个错误和解决方案。需要有解释。

这是一个big challenge.

2.说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

问题回答

1.代码补全：

```
// (1) 获取源页的内核虚拟地址
uintptr_t src_kvaddr = page2kva(page);

// (2) 获取目标页的内核虚拟地址
uintptr_t dst_kvaddr = page2kva(npage);

// (3) 复制内存内容
memcpy((void *)dst_kvaddr, (void *)src_kvaddr, PGSIZE);

// (4) 建立页表映射
ret = page_insert(to, npage, start, perm);
```

代码解析：子进程在虚拟地址上与父进程相同，但是在物理地址上有区别，所以需要重新建立页表映射。初始时，子进程内存的内容是父进程内存内容的拷贝，如果加入COW机制，则在写操作时再进行拷贝，后续会接着介绍。2.Copy on Write机制设计：

- 核心出发点：父子进程共享同一物理页（只读），所以当其中一个进程尝试写入时，操作系统会复制该页到新的物理内存位置，并更新相应的虚拟地址映射。
- 优势：不同与传统的复制整个内存页，COW机制只在需要时才进行拷贝，从而大量节省了fork操作的内存消耗，减少了大部分的缓存消耗。
- 代码：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share)
{
    //前面的代码相同

    if (*ptep & PTE_V)
    {
        if ((nptep = get_pte(to, start, 1)) == NULL)
        {
            return -E_NO_MEM;
        }
        uint32_t perm = (*ptep & PTE_USER);
        struct Page *page = pte2page(*ptep);

        if (share && (perm & PTE_W))
        {
            // COW 模式：共享物理页，设为只读
            page_ref_inc(page); // 增加引用计数

            // 父进程页设为只读
            *ptep &= ~PTE_W;

            // 子进程页指向同一物理页，只读
            *nptep = page2pa(page) | PTE_V | (perm & ~PTE_W);

            // 标记为 COW 页
            // 需要额外的数据结构跟踪 COW 页
        }
    }
}
```

3.扩展练习：

- Challenge 2：首先，该用户进程指的是 exit 程序，它在操作系统启动时就被加载到内存中。它与常用操作系统的区别在于，ucore的操作系统内核和用户程序是紧密相连的，而常规操作系统通常将内核与用户空间分开管理。这样做的原因是简化设计和提高效率，并且不需要文件系统支持。// 启动流程
kern_init() → kernel_thread(init_main) → kernel_thread(user_main) → KERNEL_EXECVE(exit) → 执行 exit 用户程序

练习三：阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现（不需要编码）

1、请分析fork/exec/wait/exit的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？

2.1 fork/exec/wait/exit的执行流程

ucore中的fork/exec/wait/exit都是通过系统调用syscall实现的。用户程序在用户态发起syscall，陷入内核态执行具体逻辑，然后返回用户态。

其实整个实验可以分为两个部分说明：

①已经存在用户进程：

也就是用户为了调用某一函数(fork/exec/wait.....)，然后执行用户态下的syscall，把函数调用号num传入寄存器a0，参数填入a1-a5。执行ecall，进入内核态。但在进入内核态之前，需要对原来的地址进行保存，接着进入内核态*_alltraps*。此时已经进入S态，先开辟一个trapframe保存通用寄存器，接着进入trap，trap中进行syscall的内核调用，把先前的a0寄存器中的调用号拿出来用syscalls判断要进行什么内核级别的操作。结束后再从*_trapret*返回U态。

②还没有用户态，要从内核出发伪造用户态：

此时系统开始跑是在S态的内核线程(user_main)，目的是和上面一样，使用一个统一的trap/syscall执行一个exec逻辑，于是此处使用**KERNEL_EXECVE / kernel_execve + ebreak**的伪造系统调用技巧。

此时在user_main中，调用KERNEL_EXECVE（其实这步和用户态的syscall差不多，只不过此时没有用户态，但为了让两种情况处理的框架一样，所以用KERNEL_EXECVE）会将传入的操作的系统调用号和参数存入寄存器，并且进行"a7==10"的魔数标记。然后和上面的情况一样，根据指令*_alltraps*、trap、trapret进行对应的操作，只不过由于没有用户态，所以此时的trapframe等都是构造的，这些细节可以参考上述原理部分。

接下来分析题干要求：

• fork：创建子进程（拷贝当前进程）

- 用户态操作：用户程序调用fork()（在user/lib/syscall.c中定义），在内部调用用户态syscall（SYS_FORK，.....），把参数放入a0-a5，执行ecall
- U-->S切换：ecall触发“CAUSE_USER_ECALL”，跳入_alltraps，进行SAVE_ALL
- 内核态操作：
 - exception_handler捕获判断信息“CAUSE_USER_ECALL”，调用syscall（）函数（内核态），根据a0数据找到sys_fork，调用内核函数do_fork(tf)
 - do_fork:

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)的主要流程：
1、检查进程数上限 nr_process;
2、alloc_proc(): 分配并初始化新的 proc_struct (状态 PROC_UNINIT 等) ;
3、setup_kstack(proc): 为子进程分配内核栈;
4、保护临界区，设置父子关系：
   local_intr_save(intr_flag);
   proc->parent = current;
   local_intr_restore(intr_flag);
5、copy_mm(clone_flags, proc): 复制或共享父进程的 mm (地址空
```


间) :

若 `current->mm == NULL` (内核线程), 直接返回 0;

否则分配新 mm、新页表, 用 `dup_mmap/copy_range` 复制用户地址空间;

把 `proc->mm` 和 `proc->pgdir` 设为新 mm。

6、`copy_thread(proc, stack, tf)`: 为子进程设置 trapframe 和内核上下文:

6.1 在子进程内核栈顶构造 `proc->tf`:

`proc->tf = (struct trapframe *) (proc->kstack + KSTACKSIZE) - 1;`

`*(proc->tf) = *tf;` // 复制父进程当前 trapframe

`proc->tf->gpr.a0 = 0;` // 关键: 让子进程的 `fork` 返回值为 0

`proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf - 8 :`

`esp;`

6.2 `proc->context.ra = (uintptr_t)forkret;`

6.3 `proc->context.sp = (uintptr_t)(proc->tf);`

(以后调度到子进程第一次运行时, 会从 `forkret` 开始, 进而返回到 `syscall` 的 trap 返回路径)

7、把子进程挂入全局结构:

7.1 在关中断临界区内:

`proc->pid = get_pid();`

`hash_proc(proc);`

`set_links(proc);` // 建立父子、兄弟链表关系, 更新 `nr_process`

8、`wakeup_proc(proc)`: 把子进程状态置为 `PROC_RUNNABLE`, 加入就绪队列;

9、`do_fork` 返回子进程 `pid` 作为 `ret`。

- `sys_fork`把`do_fork`的返回值写入当前`trapframe->gpr.a0`

- `trap`返回:

- 在父进程中, `a0`=子进程的`pid`, `fork()`返回>0

- 在子进程中, 第一次被调度运行时, 从 `forkret/forkrets` 恢复 `proc->tf`, 其中 `a0` 已预设为 0, 因此 `fork()` 返回 0。

- 用户/内核交错与返回值:

- 交错: 用户态 `fork()` → `ecall` → 内核 `do_fork` 创建子进程 → 恢复父/子 `trapframe` → 各自在用户态继续执行 `fork()` 之后的代码。

- 返回方式: 通过修改父/子各自 `trapframe` 中的 `a0`, 父进程得到子 `pid`, 子进程得到 0。

- **exec: 在当前进程里装载并运行一个新的程序镜像**

- 用户态部分: 用户程序调用`execve(path, argv.....)`, C库封装为`SYS_exec`的系统调用, 其中, `a0=SYS_exec`; `a1=程序名`, `a2=ELF镜像起始地址`, `a3=size`

- 内核态部分:

1、`sys_exec → do_execve(name, len, binary, size)`:

```
int do_execve(const char *name, size_t len, unsigned char *binary, size_t size)
```

1.1 检查用户态字符串是否合法 `user_mem_check(mm, name, len, 0);` 1.2 拷贝程序名到内核缓冲 `local_name;` 1.3 如当前进程已有旧 `mm`:

```
lsatp(boot_pgdir_pa);
if (mm_count_dec(mm) == 0) {
    exit_mmap(mm);
    put_pgdir(mm);
    mm_destroy(mm);
}
current->mm = NULL;
```

即先切回内核页表，释放旧用户地址空间。

1.4 调用 `load_icode(binary, size)` 建立新的用户程序。

2、`load_icode` 核心步骤:

2.1 创建新 `mm`: `mm_create();`

2.2 创建新页表: `setup_pgdir(mm);`

2.3 解析 ELF: 历每个 `ELF_PT_LOAD` 段, 调用 `mm_map` 建立 VMA; 使用 `pgdir_alloc_page` 为代码段、数据段分配物理页, 拷贝数据内容; 对 BSS 段清零。

2.4 构建用户栈:

```
vm_flags = VM_READ | VM_WRITE | VM_STACK;
mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL);
// 分配若干页作为栈页
pgdir_alloc_page(mm->pgdir, USTACKTOP - PGSIZE, PTE_USER);
...
```

2.5 安装新页表并切换:

```
mm_count_inc(mm);
current->mm = mm;
current->pgdir = PADDR(mm->pgdir);
lsatp(PADDR(mm->pgdir));
```

2.6 设置 `trapframe`, 使得从内核返回时会跳到新程序入口、U 态运行:

```
struct trapframe *tf = current->tf;
uintptr_t sstatus = tf->status; // 原状态
memset(tf, 0, sizeof(struct trapframe));
```

```
tf->gpr.sp = USTACKTOP;           // 用户栈顶
tf->epc     = elf->e_entry;         // ELF 入口地址
tf->status  = sstatus & ~SSTATUS_SPP; // SPP=0 表示 sret 返回到 U 模式
tf->status |= SSTATUS_SPIE;        // 返回后开中断
```

3、`load_icode` 成功后，`do_execve` 设置进程名并返回 0，`sys_exec` 把 0 写回 `a0`，`trap` 返回。

4、然而，由于 `trapframe` 中的 `epc` 和 `sp` 已经被改成“新程序入口 + 用户栈”，`sret` 执行后不会回到旧 `execve` 调用点，而是“仿佛从用户程序入口第一次开始运行”。

****交错：** **用户在 U 态调用 `execve` → `ecall` → S 态 `do_execve/load_icode` 完全替换地址空间 → 通过 `trap` 返回路径切到 U 态新程序入口；

返回值：成功时不会再回到原来的用户代码，`execve` 只有在失败时在原程序中返回 -1。

- **wait：父进程回收子进程，获取其退出状态，防止产生僵尸进程**

- 用户态：

- 父进程调用 `wait(&status)` 或 `waitpid(pid, &status, ...)`;
- C 库封装为 `SYS_wait` 系统调用，执行 `ecall`。

- 内核态：

- `sys_wait` → `do_wait(pid, code_store)`:

```
int do_wait(int pid, int *code_store)
```

1、若 `code_store != NULL`，先 `user_mem_check` 确认用户缓冲区可写；

2、寻找子进程：

2.1 若指定了 `pid`，则 `find_proc(pid)` 并检查 `proc->parent == current`;

2.2 否则从 `current->cptr` 链表遍历所有子进程；

2.3 若找到 `state == PROC_ZOMBIE` 的子进程，跳转到 `found`;

2.4 若存在子进程但都没退出：

2.4.1 把当前进程状态置为 `PROC_SLEEPING`;

2.4.2 `current->wait_state = WT_CHILD`;

2.4.3 调用 `schedule()` 让出 CPU，当前进程阻塞；

2.4.5 被唤醒后若 `PF_EXITING` 置位，则 `do_exit(-E_KILLED)`;

2.4.6 否则 `goto repeat`; 重新寻找；

2.5 若根本没有子进程，返回 `-E_BAD_PROC`。

3、found: 分支 (回收目标子进程) :

```
if (code_store != NULL) {
    *code_store = proc->exit_code;    // 把退出码写回用户缓冲区
}
local_intr_save(intr_flag);
unhash_proc(proc);
remove_links(proc);                // 从进程链表和父子关系中摘除
local_intr_restore(intr_flag);
put_kstack(proc);                  // 释放子进程内核栈
kfree(proc);                        // 释放 proc_struct
return 0;
```

◦ 用户/内核交错与返回值:

- 交错: 用户态 `wait` → `ecall` → 内核 `do_wait` 把父进程睡眠 → 其他进程运行 → 子进程 `do_exit` 唤醒父 → 父再进内核 `do_wait` 找到 ZOMBIE 子 → 回收并返回;
- `sys_wait` 把返回值写入 `a0`, `sret` 后用户态 `wait` 得到返回值。

• **exit: 让当前进程终止运行, 并把退出状态传给父进程。**

◦ 用户态:

- 用户程序调用 `exit(code)` 或在 `main` 中 `return code`;
- C 库将其封装为 `SYS_exit` 系统调用, 执行 `ecall`

◦ 内核态:

`sys_exit` → `do_exit(error_code)`:

```
int do_exit(int error_code)
```

1. 禁止 `idleproc`、`initproc` 退出 (panic) ;

2. 若当前进程有用户地址空间 `mm`:

- 切换回内核页表 `lsatp(boot_pgdir_pa)`;
- `mm_count_dec(mm)` 若为 0, 则:
 - `exit_mmap(mm)`: 回收所有用户页;
 - `put_pgdir(mm)`: 释放页表;
 - `mm_destroy(mm)`: 销毁 `mm` 结构;
- `current->mm = NULL`;

3. 将当前进程标记为 ZOMBIE:

```
ccurrent->state = PROC_ZOMBIE;
current->exit_code = error_code;
```

4. 处理父子关系与唤醒父进程:

```

clocal_intr_save(intr_flag);
proc = current->parent;
if (proc->wait_state == WT_CHILD) {
    wakeup_proc(proc);          // 唤醒正在 wait 的父进程
}
// 把当前进程的所有子进程过继给 initproc, 并唤醒 initproc(如有需要)
while (current->cptr != NULL) {
    proc = current->cptr;
    current->cptr = proc->optr;
    proc->yptr = NULL;
    if ((proc->optr = initproc->cptr) != NULL) {
        initproc->cptr->>yptr = proc;
    }
    proc->parent = initproc;
    if (proc->state == PROC_ZOMBIE && initproc->wait_state == WT_CHILD)
    {
        wakeup_proc(initproc);
    }
}
local_intr_restore(intr_flag);

```

5. 调度到其他进程:

```

cschedule();
panic("do_exit will not return!! %d.\n", current->pid);

```

当前进程不会再回到用户态；真正的资源完全释放在父进程随后调用的 `do_wait` 中完成。

◦ 交错与返回:

- 用户态 `exit` → `ecall` → 内核 `do_exit`:
 - 释放用户空间;
 - 标记为 ZOMBIE;
 - 唤醒父进程;
 - 调度到其他进程;
- 对当前进程来说没有“返回值”，它的执行就此结束；对父进程来说，退出码通过 `do_wait` 写入用户缓冲并返回。

2、请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

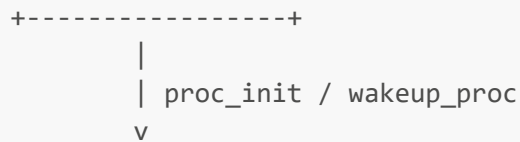
创建阶段 (`alloc_proc`, `do_fork`, `do_execve`)

|
v

+-----+

| PROC_UNINIT | <----- PROC_UNINIT: 刚通过

`alloc_proc` 分配, 尚未加入就绪队列;



`PROC_RUNNABLE`: 就绪可运行 (可能正在运行或等待被调度), 如 `do_fork` 创建完子进程后 `wakeup_proc` 设置;



- 主要状态:
- `PROC_UNINIT`: 刚通过 `alloc_proc` 分配, 尚未加入就绪队列;
- `PROC_RUNNABLE`: 就绪可运行 (可能正在运行或等待被调度), 如 `do_fork` 创建完子进程后 `wakeup_proc` 设置;
- `PROC_SLEEPING`: 如在 `do_wait` 中找不到已退出子进程时, 设为 `PROC_SLEEPING + WT_CHILD` 并调用 `schedule()`;
- `PROC_ZOMBIE`: `do_exit` 中释放完用户空间, 保留 `proc_struct`/内核栈/`exit_code` 等, 等待父进程 `do_wait`。
- 关键转换触发:
 - `alloc_proc/do_fork/do_execve`: 创建并进入 `PROC_RUNNABLE`;

- `schedule/proc_run`: 在多个 `PROC_RUNNABLE` 间切换;
- `do_wait`: 把当前进程从 `RUNNING` 变为 `PROC_SLEEPING`;
- `wakeup_proc`: 把 `PROC_SLEEPING` 变回 `PROC_RUNNABLE`;
- `do_exit`: 把 `RUNNING` 进程状态改成 `PROC_ZOMBIE`;
- `do_wait` 中 `unhash_proc/remove_links/put_kstack/kfree`: 彻底销毁子进程结构。

Lab2分支任务: gdb调试页表查询过程

1.准备工作

重新配置并编译QEMU, 然后将ucore的Makefile中的QEMU的定义修改为新编译的调试版QEMU。

接着分别打开三个终端:

- 终端A:执行`make debug`, 启动我们新编译的调试版QEMU, 并暂停在初始状态。
- 终端B:附加调试QEMU进程
- 执行`make gdb`, 调试ucore内核

2.调试过程

在终端B中在`get_physical_address`函数处打断点并继续运行。

```
(gdb) b get_physical_address
Breakpoint 1 at 0x5ba7c0bb6e13: file /home/he/桌面/qemu-4.1.1/target/riscv/cpu_helper.c, line 158.
(gdb) c
Continuing.
```

接着在终端C中在函数`kern_init`处打断点并执行至该断点, 执行指令`x/8i $pc`查看后续指令

```
Breakpoint 1, kern_init () at kern/init/init.c:30
30      memset(edata, 0, end - edata);
(gdb) x/8i $pc
=> 0xfffffffffc02000d8 <kern_init>:  auipc    a0,0x5
0xfffffffffc02000dc <kern_init+4>:  addi     a0,a0,-192
0xfffffffffc02000e0 <kern_init+8>:  auipc    a2,0x5
0xfffffffffc02000e4 <kern_init+12>: addi     a2,a2,-104
0xfffffffffc02000e8 <kern_init+16>: addi     sp,sp,-16
0xfffffffffc02000ea <kern_init+18>: sub     a2,a2,a0
0xfffffffffc02000ec <kern_init+20>: li      a1,0
0xfffffffffc02000ee <kern_init+22>: sd      ra,8(sp)
```

发现`0xfffffffffc02000ee`处的访存指令`sd ra,8(sp)`, 执行单步执行指令`si`直到下一条执行就是访存指令。使用命令`p/x $sp`打印出`sp`的值为`0xfffffffffc0203ff0`。之后执行命令`c`。

```
(gdb) x/i $pc
=> 0xfffffffffc02000ee <kern_init+22>:    sd    ra,8(sp)
(gdb) p/x $sp
$1 = 0xfffffffffc0203ff0
(gdb) c
Continuing.
```

执行访存指令sd后发现终端B触发断点get_physical_address函数，使用命令p/x addr打印地址值为0xfffffffffc02000d8,正好为终端中sp的值加8

```
Thread 1 "qemu-system-ris" hit Breakpoint 1, get_physical_address (
    env=0x5ba7ffa8f040, physical=0x7fffe7e75f38, prot=0x7fffe7e75f30,
    addr=18446744072637907160, access_type=0, mmu_idx=1)
  at /home/he/桌面/qemu-4.1.1/target/riscv/cpu_helper.c:158
158 {
(gdb) p/x addr
$1 = 0xfffffffffc02000d8
```

Lab5分支任务：gdb 调试系统调用以及返回

1. 调试环境准备

分别打开三个终端：

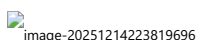
- 终端 A: `make debug`，启动 qemu 并等待连接；
- 终端 B: `make gdb`，连接到 qemu，作为 gdb1，主要用于观察从 U 态进入内核、以及从内核返回 U 态的流程；
- 终端 C: `gdb -p <qemu-pid>` 作为 gdb2，附加到 qemu 进程，用于在宿主机上辅助观察（如果系统默认 ptrace 限制过严，需要先执行 `echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope` 临时放宽限制）。

（具体打开可以参考下图）：

分别打开终端A、B，执行“make debug”“make gdb”



再开终端C，执行“”



此处attach时注意可以先临时放宽ptrace 限制（执行“echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope”）

2. 从用户态 `ecall` 进入内核：__alltraps 和 SAVE_ALL

在 gdb1 中，先确认当前位于用户态的 `syscall()` 封装函数中（`user/libs/syscall.c` 的内联汇编）。通过单步指令 `si`，观察 `ecall` 前后 PC 的变化：

```
gdb(gdb) si
0x0000000000800102 19      asm volatile (
(gdb) i r pc
pc                0x800102 0x800102 <syscall+42>
(gdb) si
0x0000000000800104 19      asm volatile (
(gdb) i r pc
pc                0x800104 0x800104 <syscall+44>

// 执行前（用户态）：
(gdb) x/3i $pc
=> 0x800104 <syscall+44>: ecall
    0x800108 <syscall+48>: sd a0,28(sp)
    0x80010c <syscall+52>: lw a0,28(sp)
```

可以看到，`0x800104` 处正是 `ecall` 指令。继续单步执行一次：

```
gdb(gdb) si
0xfffffffffc0200e44 in __alltraps () at kern/trap/trapentry.S:123
123      SAVE_ALL
(gdb) i r pc
pc                0xfffffffffc0200e44 0xfffffffffc0200e44 <__alltraps+4>
(gdb) x/7i $pc
=> 0xfffffffffc0200e44 <__alltraps+4>:
    bnez sp,0xfffffffffc0200e4c <__alltraps+12>
    0xfffffffffc0200e48 <__alltraps+8>: csrr sp,sscratch
    0xfffffffffc0200e4c <__alltraps+12>: addi sp,sp,-288
    0xfffffffffc0200e4e <__alltraps+14>: sd zero,0(sp)
    0xfffffffffc0200e50 <__alltraps+16>: sd ra,8(sp)
    0xfffffffffc0200e52 <__alltraps+18>: sd gp,24(sp)
    0xfffffffffc0200e54 <__alltraps+20>: sd tp,32(sp)
```

此时 PC 已经从用户空间的 `0x800104` 跳到内核的 `__alltraps`，进入 `SAVE_ALL` 宏。说明 `ecall` 触发了异常，硬件按 RISC-V 规范完成了以下动作：

- 将触发异常的用户 PC 保存到 `sepc`；
- 将异常原因写入 `scause`；
- 根据 `stvec` 跳转到内核 trap 入口 `__alltraps`；
- 切换特权级为 S 模式。

在 `__alltraps` 入口，通过 gdb 直接读取相关 CSR：

```
gdb(gdb) info registers sepc scause sstatus
sepc                0x800104 8388868
```

```
scause      0x8      8
sstatus     0x8000000000046020 -9223372036854489056
```

可以看到：

- `sepc = 0x800104`，保存的正是刚才那条 `ecall` 的地址；
- `scause = 0x8`，对应「Environment call from U-mode」（用户态系统调用）；
- `sstatus` 中的 `SPP`、`SPIE` 等位已按 trap 规则更新（例如 `SPP` 记录异常前处于 U 态）。

`SAVE_ALL` 随后会根据 `sscratch` 判断 trap 来源于 U 态还是 S 态，完成「切换到内核栈 + 保存全部通用寄存器 + 保存 `sstatus/sepc/stval/scause` 到 `trapframe`」等工作，为后续 C 语言层的 `trap()` 提供完整上下文。

3. C 语言层的 `trap()` / `trap_dispatch()`：识别 `syscall` 并分发

`__alltraps` 在完成现场保存后，会：

```
asmmove a0, sp      # a0 指向 trapframe
jal trap            # 进入 C 语言的 trap(tf)
```

在 gdb 中打印 `trapframe` 可以看到这次系统调用的寄存器状态：

```
gdb(gdb) p *tf
$1 = {gpr = {zero = 0, ra = 8390000, sp = 2147483456, gp = 0, tp = 0, t0 = 0,
  t1 = 0, t2 = 0, s0 = 0, s1 = 32, a0 = 2, a1 = 0, a2 = 0, a3 = 0, a4 = 0,
  a5 = 0, a6 = 0, a7 = 0, s2 = 0, s3 = 0, s4 = 0, s5 = 0, s6 = 0, s7 = 0,
  s8 = 0, s9 = 0, s10 = 0, s11 = 0, t3 = 0, t4 = 0, t5 = 0, t6 = 0},
  status = 9223372036855062560, epc = 8388868, tval = 0, cause = 8}
(gdb) p tf->cause
$2 = 8
(gdb) p tf->epc
$3 = 8388868
```

将 `epc/cause` 换算为 16 进制更直观：

```
gdb(gdb) p/x tf->epc
$4 = 0x800104
(gdb) p/x tf->cause
$5 = 0x8
```

可以总结为：

- `tf->epc = 0x800104`：正是用户态 `ecall` 指令地址；
- `tf->cause = 8`：User-mode `ecall`；
- `tf->gpr.a0 = 2`：这是用户态传入的系统调用号（本次 `num=2`）。

`trap()` 函数内部会根据 `tf->cause` 调用 `trap_dispatch(tf)/exception_handler(tf)`，进而识别 `CAUSE_USER_ECALL` 并调用内核态的 `syscall()`：

```
cvoid exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_BREAKPOINT:
            ...
            break;
        case CAUSE_USER_ECALL:
            tf->epc += 4; // 返回时跳过 ecall
            syscall();   // 进入内核态 syscall 分发
            break;
        ...
    }
}
```

而内核态 `syscall()` 则从 `trapframe` 中取出系统调用号和参数，查表调用具体的 `sys_xxx`，并将返回值写回 `tf->gpr.a0`，为后续返回用户态做好准备。

4. 从内核通过 `sret` 返回用户态

当内核完成本次系统调用处理 (`trap()` / `exception_handler()` / `syscall()` / `sys_xxx`) 后，控制流会从 C 函数返回到汇编层的 `__trapret`：

```
asm.globl __trapret
__trapret:
    RESTORE_ALL
    sret
```

`RESTORE_ALL` 从当前 `trapframe` 恢复 `sstatus`、`sepc` 以及所有通用寄存器，最后恢复 `x2 (sp)`，使得栈指针重新指向用户栈；根据 `sstatus.SPP` 的值决定 `sret` 返回到 U 态还是 S 态。对于本次用户态 `ecall` 而言，`SPP=0`，因此会返回到 U 模式。

在 `gdb` 中，我使用 `finish` 命令让 `trap()` 正常返回，可以看到调用栈和 PC 已经回到用户空间：

```
gdb(gdb) finish
Run till exit from #0  trap (tf=0xfffffffffc04b0ee0) at kern/trap/trap.c:243

Breakpoint 1, syscall (num=2) at user/libs/syscall.c:19
19      asm volatile (
(gdb) bt
#0  syscall (num=2) at user/libs/syscall.c:19
#1  0x0000000000800570 in main () at user/exit.c:9
Backtrace stopped: frame did not save the PC
```

此时的 PC 及附近指令为：

```

gdb(gdb) i r pc
pc                0x8000f8 0x8000f8 <syscall+32>

(gdb) x/20i $pc
=> 0x8000f8 <syscall+32>: ld a0,8(sp)
   0x8000fa <syscall+34>: ld a1,40(sp)
   0x8000fc <syscall+36>: ld a2,48(sp)
   0x8000fe <syscall+38>: ld a3,56(sp)
   0x800100 <syscall+40>: ld a4,64(sp)
   0x800102 <syscall+42>: ld a5,72(sp)
   0x800104 <syscall+44>: ecall
   0x800108 <syscall+48>: sd a0,28(sp)
   0x80010c <syscall+52>: lw a0,28(sp)
   0x80010e <syscall+54>: addi sp,sp,144
   0x800110 <syscall+56>: ret
   ...

```

可以看到：

- 程序已经重新回到了用户态的 `syscall()` 函数中；
- PC 位于 `syscall+32`，紧接在完成参数装载、即将执行 `ecall` 附近；
- 汇编里既包含刚刚触发 trap 的 `ecall` 指令（`0x800104`），也包含后续将返回值写回栈的 `sd a0,28(sp)` 等指令。

结合之前在 `__alltraps` 入口处读取到的：

- `sepc = 0x800104`（用户态 `ecall` 地址）；
- `scause = 8`（User-mode `ecall`）；

可以推断，在 `trap()` 返回之后，内核通过 `trapentry.S` 中的 `RESTORE_ALL` 宏恢复了 `trapframe` 中保存的寄存器和 CSR，并执行了 `sret` 指令：

- `sret` 使用 `sepc` 中保存的用户态 PC 作为返回地址，将 PC 恢复为 `0x800104` 之后的位置；
- 根据 `sstatus` 中的 `SPP` 位把特权级从 S 模式切回 U 模式；
- 于是 CPU 从内核 trap 处理流程返回到用户态的 `syscall()`，继续执行 `ecall` 之后的指令，最终把 `a0` 中的系统调用返回值写入用户栈，并作为 C 函数返回值返回给调用者。

至此，通过 `gdb`，我们完整观察并验证了一次用户态系统调用的典型执行路径：

1. U 态发起 `ecall`；
2. 硬件保存 `sepc/scause` 并跳到 `__alltraps`，`SAVE_ALL` 构造 `trapframe`；
3. C 层 `trap()/exception_handler()/syscall()` 在 S 态根据 `tf->scause` 和参数分发并执行内核服务；
4. `__trapret` 中 `RESTORE_ALL + sret` 恢复上下文，将 CPU 从 S 态重新送回 U 态原程序继续执行。