lab 5

小组成员：朱欣宁（2313686） 宁宇嫣（2313123） 雷雨彦（2313894）

练习1：加载应用程序并执行

load_icode函数设计思路：

load_icode函数的主要功能是将ELF格式的二进制程序加载到当前进程的内存空间中，并设置好进程的初始执行环境。具体步骤如下：

- **创建新的内存管理结构：** 为当前进程分配新的mm_struct；
- **创建页目录表：** 建立进程的页表结构；
- **加载程序段：** 解析ELF文件，加载代码段、数据段等——
  验证ELF文件魔数；
  遍历程序头表，加载所有LOAD类型的段；
  为代码段、数据段等分配物理页并复制内容；
  初始化BSS段（清零）。
- **建立用户栈：** 为用户进程分配栈空间；
- **设置进程内存管理信息：** 更新进程的mm、pgdir等字段；
- **设置陷阱帧：** 配置用户态执行的初始上下文。

内存布局建立：

- 代码段/数据段：根据ELF文件中的虚拟地址加载；
- 用户栈：USTACKTOP-USTACKSIZE到USTACKTOP区间；
- 通过mm_map建立虚拟内存区域映射；
- 通过pgdir_alloc_page分配物理页并建立页表映射。

具体代码：

本次实验中我们主要要完成的部分是第六步。即设置陷阱帧（trapframe）的关键实现，具体代码如下：

```
//(6) setup trapframe for user environment
struct trapframe *tf = current->tf;
// 保存原始的sstatus值
uintptr_t sstatus = tf->status;
memset(tf, 0, sizeof(struct trapframe));

/* 设置关键寄存器值 */

// 1. 设置用户栈指针：指向用户栈顶
tf->gpr.sp = USTACKTOP;  // 用户栈顶地址

// 2. 设置程序计数器：指向ELF入口点
tf->epc = elf->e_entry;

// 3. 设置状态寄存器：配置为用户模式
```

```
    tf->status = sstatus;

    // 4. 调整状态寄存器位:
    //    - 清除SPP位（设置为用户模式）
    //    - 设置SPIE位（允许中断）
    //    - 清除SIE位（禁止中断，在返回用户模式前）
    tf->status &= ~SSTATUS_SPP;  // SPP=0表示用户模式
    tf->status |= SSTATUS_SPIE;  // 使能中断（用户模式下）
    tf->status &= ~SSTATUS_SIE;  // 禁止中断（当前内核模式下）
```

关键设置：

栈指针（sp）设置：

USTACKTOP是用户栈的顶部地址；
在RISC-V中，gpr.sp对应x2寄存器（栈指针寄存器）；
用户程序从高地址向低地址使用栈空间。

程序计数器（epc）设置：

elf->e_entry是ELF文件的入口点地址；
当从内核态返回用户态时，CPU会从epc指向的地址开始执行。

状态寄存器（status）设置：

SPP位：表示进入陷阱前的特权级，0=用户模式，1=内核模式；
SPIE位：表示进入陷阱前中断使能状态；
SIE位：当前中断使能状态；
通过清除SPP位确保返回到用户模式；
通过设置SPIE确保用户模式下中断可用。

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？

1. 进程调度选择阶段

- **调度器决策**：ucore的调度器（如`schedule()`函数）从就绪队列中选择一个RUNNABLE状态的进程
- **切换准备**：将目标进程状态设为RUNNING，当前运行进程状态设为RUNNABLE（如果需要）

2. 上下文切换阶段 (`proc_run()`)

```
// kern/process/proc.c
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        // 1. 保存当前进程上下文
        // 2. 切换内核栈
        current = proc;
        // 3. 切换页表 - 关键步骤!
```

```
            lcr3(PADDR(proc->pgdir));  // 加载新进程的页表基址
            // 4. 切换到新进程的上下文
            switch_to(&(prev->context), &(proc->context));
        }
    }
```

- **页表切换**：`lcr3()`加载新进程的页目录物理地址，CPU开始使用新进程的地址空间
- **上下文切换**：`switch_to()`保存/恢复通用寄存器，但**不恢复程序计数器PC**

3. 新进程开始执行阶段

- 由于`switch_to()`只切换了通用寄存器上下文，新进程从它上次被切换出的位置继续执行
- 对于**新创建的进程**，这个位置通常是`forkret()`函数：

```
// forkret是fork系统调用返回的入口点
static void forkret(void) {
    forkrets(current->tf);  // 跳转到forkrets
}
```

4. 陷阱返回阶段 (`forkrets()` → `__trapret`)

```
# kern/trap/trapentry.S
__trapret:
    # 1. 从陷阱帧(trapframe)恢复所有寄存器
    LOAD_ALL_REGS

    # 2. 恢复sstatus（包含特权级信息）
    csrw sstatus, t0

    # 3. 恢复sepc（程序计数器）
    csrw sepc, t1

    # 4. 返回用户态并跳转到sepc指向的地址
    sret
```

- **恢复关键寄存器**：
  - `sepc` ← `tf->epc` = ELF入口地址（由load_icode设置）
  - `sstatus` ← `tf->status` = 用户模式配置（SPP=0）
  - `sp` ← `tf->gpr.sp` = USTACKTOP（用户栈顶）

5. 执行sret指令的原子操作

当CPU执行`sret`指令时，**原子性地**完成以下操作：

1. **特权级切换**：根据`sstatus.SPP`位，从内核态(S-mode)切换到用户态(U-mode)
2. **中断使能**：将`sstatus.SIE`设置为`sstatus.SPIE`（恢复中断状态）

3. **跳转执行**：`PC ← sepc`，开始执行用户程序的第一条指令

6. 用户程序第一条指令执行

- **位置**：ELF文件指定的入口点（通常是`_start`或`main`）
- **环境**：
  - 处于用户态(U-mode)，只能访问用户空间内存
  - 栈指针指向USTACKTOP，栈已包含必要的启动参数
  - 可以正常处理中断（因为SPIE=1）

关键数据流总结

```
调度器选择进程
    ↓
proc_run()：切换页表 + 切换上下文
    ↓
进程从forkret()开始执行（内核态）
    ↓
forkrets()：准备陷阱返回
    ↓
__trapret：从current->tf恢复寄存器
    ↓
sret指令：原子切换到用户态 + 跳转到epc
    ↓
执行ELF入口点第一条指令（用户态）
```

核心机制说明

1. **两个层面的切换**：
   - **进程切换**：由`proc_run()`完成，切换页表和内核上下文
   - **特权级切换**：由`sret`指令完成，从内核态切换到用户态
2. **陷阱帧的核心作用**：
   - 是连接内核和用户态的桥梁
   - 保存了用户态执行所需的所有寄存器状态
   - 由`load_icode()`在第6步精心设置
3. **第一次执行的特殊性**：
   - 对于新创建的进程，它的"上次被切换出"的位置就是forkret()
   - 因此总是通过陷阱返回路径进入用户态
   - 保证了所有进程有统一的入口机制

练习2：父进程复制自己的内存空间给子进程

设计实现过程

copy_range函数设计思路：

copy_range函数的主要功能是将父进程指定虚拟地址范围内的内存内容复制到子进程的对应虚拟地址空间中。具体设计思路如下：

1. **地址对齐检查**：确保起始地址start和结束地址end都是页对齐的（PGSIZE的倍数）
2. **逐页复制**：使用do-while循环逐页处理虚拟地址空间
3. **分步操作**：
    - 在父进程页表中查找对应虚拟地址的页表项
    - 如果页表项有效（PTE_P标志位为1），则获取对应的物理页
    - 为子进程分配新的物理页
    - 将父进程物理页的内容复制到子进程物理页
    - 建立子进程虚拟地址到新物理页的映射关系

关键数据结构：

```
// 主要使用的数据结构
pte_t *ptep;      // 父进程页表项指针
pte_t *nptep;     // 子进程页表项指针
struct Page *page;    // 父进程物理页
struct Page *npage;   // 子进程物理页
uint32_t perm;        // 页面权限标志
```

具体实现代码：

```
/* copy_range - copy content of memory (start, end) of one process A to another
 * process B
 * @to:    the addr of process B's Page Directory
 * @from:  the addr of process A's Page Directory
 * @share: flags to indicate to dup OR share. We just use dup method, so it
 * didn't be used.
 *
 * CALL GRAPH: copy_mm-->dup_mmap-->copy_range
 */
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
               bool share)
{
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do
    {
        // call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL)
        {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
```

```
        // call get_pte to find process B's pte according to the addr start. If
        // pte is NULL, just alloc a PT
        if (*ptep & PTE_V)
        {
            if ((nptep = get_pte(to, start, 1)) == NULL)
            {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            // get page from ptep
            struct Page *page = pte2page(*ptep);
            // alloc a page for process B
            struct Page *npage = alloc_page();
            assert(page != NULL);
            assert(npage != NULL);
            int ret = 0;
            /* LAB5:EXERCISE2 YOUR CODE
             * replicate content of page to npage, build the map of phy addr of
             * nage with the linear addr start
             *
             * Some Useful MACROs and DEFINEs, you can use them in below
             * implementation.
             * MACROs or Functions:
             *    page2kva(struct Page *page): return the kernel vritual addr of
             * memory which page managed (SEE pmm.h)
             *    page_insert: build the map of phy addr of an Page with the
             * linear addr la
             *    memcpy: typical memory copy function
             *
             * (1) find src_kvaddr: the kernel virtual address of page
             * (2) find dst_kvaddr: the kernel virtual address of npage
             * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
             * (4) build the map of phy addr of  nage with the linear addr start
             */
            // (1) find src_kvaddr: the kernel virtual address of page
            void *src_kvaddr = page2kva(page);

            // (2) find dst_kvaddr: the kernel virtual address of npage
            void *dst_kvaddr = page2kva(npage);

            // (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);

            // (4) build the map of phy addr of npage with the linear addr start
            ret = page_insert(to, npage, start, perm);
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

实现细节说明

页表项获取：

- 使用get_pte(from, start, 0)获取父进程的页表项
- 使用get_pte(to, start, 1)获取或创建子进程的页表项
- 如果父进程页表项不存在，则跳过该页继续处理

物理页分配与复制：

- 通过pte2page(*ptep)获取父进程的物理页
- 使用alloc_page()为子进程分配新的物理页
- 通过page2kva()获取内核虚拟地址，进行内存复制

权限传递：

- 从父进程页表项中提取权限位（*ptep & PTE_USER
- 将相同的权限设置给子进程的页表项

Copy on Write（COW）机制设计

设计目标

实现内存复制的延迟执行，提升fork系统调用的效率，减少不必要的内存拷贝，同时保证进程间内存访问的隔离性。

核心设计思路

基础原理：

fork阶段父子进程共享物理页，将共享页权限设为只读并标记 COW 标志；当任一进程对页面执行写操作时，触发页错误异常，在异常处理中完成物理页的按需复制，并恢复可写权限。

关键扩展：

1. **数据结构**：在struct Page中增加ref_count引用计数，记录共享物理页的进程数；自定义PTE_COW标志位标记 COW 页面。
2. **fork阶段修改**：copy_range函数中不再复制物理页，而是增加页面引用计数，为父子进程设置只读 + COW 的页表映射。
3. **页错误处理**：在do_pgfault中识别 COW 触发的写错误，若页面被多进程共享则复制新页并更新映射，若仅当前进程使用则直接恢复可写权限。

核心实现要点：

1. **引用计数管理**：页面分配时初始化引用计数为 1，共享时递增，释放时递减，仅当计数为 0 时真正释放物理页。
2. **页表操作**：共享时清除写权限并添加 COW 标志，复制后恢复写权限并清除 COW 标志，同时刷新 TLB 保证映射有效性。
3. **异常处理**：区分 COW 写错误与普通缺页错误，按需完成页面复制或权限修改，保证操作原子性与系统状态一致性。

机制优势:

1. **性能提升**：fork操作无需立即复制内存，创建进程的速度大幅提升；
2. **内存节约**：仅在真正写入时复制页面，减少冗余内存占用；
3. **透明性**：对应用程序无感知，保持进程内存隔离的语义。

练习3: fork/exec/wait/exit 系统调用分析

相关文件

1. **用户态系统调用接口**:
   - `user/libs/syscall.c` - 用户程序调用系统调用的封装函数
   - `user/libs/ulib.c` - 用户库函数(exit, fork, wait等)
2. **内核态系统调用处理**:
   - `kern/syscall/syscall.c` - 系统调用分发器
   - `kern/process/proc.c` - 系统调用的实际实现(do_fork, do_exit, do_wait, do_execve)
   - `kern/process/proc.h` - 进程状态定义和结构体
3. **陷阱处理机制**:
   - `kern/trap/trap.c` - 异常和中断处理,包括系统调用入口
   - `kern/trap/trapentry.S` - 从用户态进入内核态的汇编代码
4. **系统调用定义**:
   - `libs/unistd.h` - 系统调用号定义

---

fork/exec/wait/exit 执行流程分析

1. fork 执行流程

用户态部分:

1. **用户程序调用**: `fork()` (在 `user/libs/ulib.c`)
   - 调用 `sys_fork()` (在 `user/libs/syscall.c`)
   - `sys_fork()` 调用通用 `syscall(SYS_fork)` 函数
2. **系统调用封装** (`user/libs/syscall.c`):

```
syscall(int64_t num, ...) {
    // 准备参数到寄存器 a0-a5
    asm volatile (
        "ld a0, %1\n"  // 系统调用号
        "ld a1, %2\n"  // 参数1
        ...
        "ecall\n"      // 触发系统调用异常
        "sd a0, %0"    // 保存返回值
```

```
    );
}
```

- **用户态操作**: 准备参数,执行 `ecall` 指令触发异常

用户态→内核态转换:

3. **陷阱入口** (`kern/trap/trapentry.S`):
   - `ecall` 指令触发 `CAUSE_USER_ECALL` 异常
   - CPU跳转到 `__alltraps` 入口
   - `SAVE_ALL` 宏保存所有寄存器到内核栈(trapframe)
   - 调用 `trap(tf)` 函数
4. **异常处理** (`kern/trap/trap.c`):
   - `exception_handler()` 识别 `CAUSE_USER_ECALL`
   - 调用 `syscall()` 函数处理系统调用

内核态部分:

5. **系统调用分发** (`kern/syscall/syscall.c`):

```
void syscall(void) {
    struct trapframe *tf = current->tf;
    int num = tf->gpr.a0;  // 从trapframe获取系统调用号
    arg[0] = tf->gpr.a1;   // 获取参数
    ...
    tf->gpr.a0 = syscalls[num](arg);  // 调用对应的处理函数
}
```

   - 从trapframe中提取系统调用号和参数
   - 调用 `sys_fork()` 处理函数
6. **fork实现** (`kern/process/proc.c`):

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    // 1. 分配进程控制块
    proc = alloc_proc();
    // 2. 分配内核栈
    setup_kstack(proc);
    // 3. 复制内存管理信息
    copy_mm(clone_flags, proc);
    // 4. 设置陷阱帧和上下文
    copy_thread(proc, stack, tf);
    // 5. 将进程加入哈希表和进程列表
    proc->pid = get_pid();
    hash_proc(proc);
```

```
    set_links(proc);
    // 6. 唤醒新进程
    wakeup_proc(proc);
    // 7. 返回新进程的PID
    return proc->pid;
}
```

- **内核态操作**: 创建新进程,复制内存空间,设置上下文

内核态→用户态返回:

7. **返回用户态**:
   - `syscall()` 将返回值写入 `tf->gpr.a0`
   - `trap()` 函数返回
   - `__trapret` 恢复所有寄存器
   - `sret` 指令返回到用户态
   - 用户态从 `a0` 寄存器读取返回值(父进程返回子进程PID,子进程返回0)

---

2. exec 执行流程

用户态部分:

1. **用户程序调用**: 通过系统调用接口调用 `exec`
   - 准备参数: 程序名,二进制数据等
   - 执行 `ecall` 进入内核态

内核态部分:

2. **系统调用处理** (`kern/syscall/syscall.c`):
   - `sys_exec()` 调用 `do_execve()`
3. **exec实现** (`kern/process/proc.c`):

```
int do_execve(const char *name, size_t len, unsigned char *binary, size_t
size) {
    // 1. 释放当前进程的内存空间
    if (mm != NULL) {
        exit_mmap(mm);
        put_pgdir(mm);
        mm_destroy(mm);
    }
    // 2. 加载新的程序
    load_icode(binary, size);
    // 3. 设置进程名
    set_proc_name(current, local_name);
    // 4. 返回到用户态执行新程序
```

```
    forkrets(current->tf);
}
```

4. **加载程序** (`load_icode()`):
    - 创建新的内存空间
    - 解析ELF格式
    - 加载代码段、数据段
    - 设置用户栈
    - 设置trapframe: `epc`=程序入口, `sp`=用户栈顶, `status`=用户模式

5. **返回用户态**:
    - `forkrets(current->tf)` 设置栈指针指向新的trapframe
    - `__trapret` 恢复寄存器
    - `sret` 返回到新程序的入口点执行

---

3. wait 执行流程

用户态部分:

1. **用户程序调用**: `wait()` 或 `waitpid()`
    - 调用 `sys_wait()`
    - 执行 `ecall` 进入内核态

内核态部分:

2. **系统调用处理**:
    - `sys_wait()` 调用 `do_wait()`

3. **wait实现** (`kern/process/proc.c`):

```
int do_wait(int pid, int *code_store) {
    repeat:
        // 查找子进程
        if (找到ZOMBIE状态的子进程) {
            // 回收资源
            unhash_proc(proc);
            remove_links(proc);
            put_kstack(proc);
            kfree(proc);
            return 0;
        }
        // 如果没有找到,进入睡眠状态
        current->state = PROC_SLEEPING;
        current->wait_state = WT_CHILD;
        schedule();  // 切换到其他进程
```

```
        goto repeat;
}
```

- 查找指定或任意ZOMBIE子进程
- 如果找到,回收资源并返回
- 如果未找到,设置等待状态并调度

4. **返回用户态**:
   - 返回值通过 `tf->gpr.a0` 返回给用户程序

---

4. exit 执行流程

用户态部分:

1. **用户程序调用**: `exit(error_code)`
   - 调用 `sys_exit()`
   - 执行 `ecall` 进入内核态

内核态部分:

2. **系统调用处理**:
   - `sys_exit()` 调用 `do_exit()`
3. **exit实现** (`kern/process/proc.c`):

```
int do_exit(int error_code) {
    // 1. 释放内存空间
    if (mm != NULL) {
        exit_mmap(mm);
        put_pgdir(mm);
        mm_destroy(mm);
    }
    // 2. 设置进程状态为ZOMBIE
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;
    // 3. 处理子进程(交给init进程)
    // 4. 唤醒等待的父进程
    if (proc->wait_state == WT_CHILD) {
        wakeup_proc(proc);
    }
    // 5. 调度到其他进程
    schedule();
}
```

- 释放进程内存空间
- 设置状态为ZOMBIE

- 唤醒等待的父进程
- 调度到其他进程(不会返回)

用户态与内核态操作总结

用户态完成的操作:

1. **系统调用准备**: 准备参数,调用封装函数
2. **触发系统调用**: 执行 `ecall` 指令
3. **接收返回值**: 从寄存器 `a0` 读取返回值

内核态完成的操作:

1. **保存上下文**: 保存所有寄存器到trapframe
2. **系统调用处理**: 执行具体的系统调用功能
3. **进程管理**: 创建/销毁进程,管理进程状态
4. **内存管理**: 分配/释放内存,加载程序
5. **调度**: 进程切换和调度
6. **恢复上下文**: 恢复寄存器,返回用户态

内核态与用户态的交错执行:

```
用户态程序
    ↓ (调用fork)
用户态: 准备参数,执行ecall
    ↓ (异常)
内核态: 保存上下文(trapentry.S)
    ↓
内核态: 处理系统调用(syscall.c, proc.c)
    ↓ (fork创建子进程)
内核态: 调度器可能切换到子进程
    ↓ (返回)
内核态: 恢复上下文(trapentry.S)
    ↓ (sret)
用户态: 继续执行,读取返回值
```

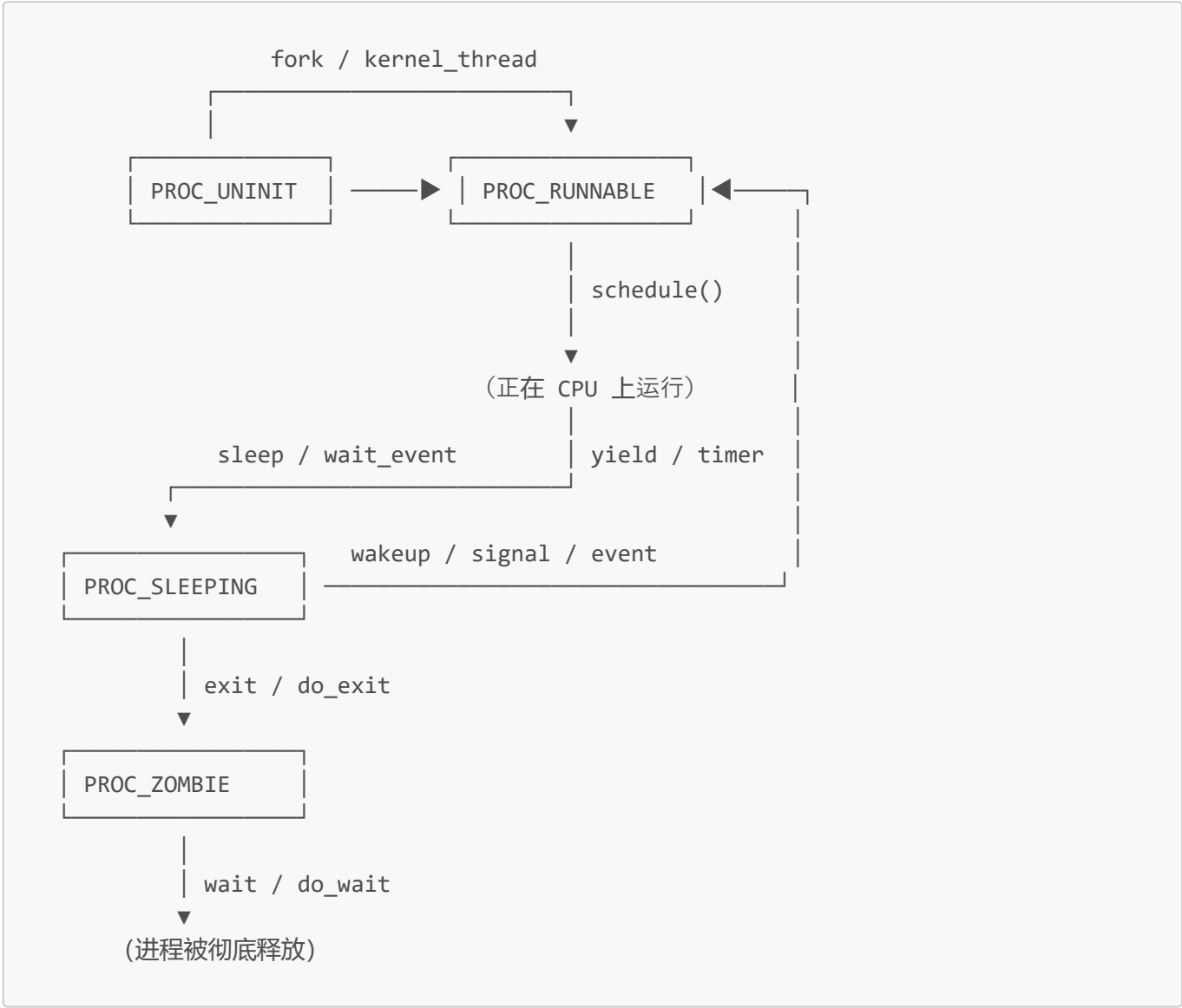
内核态执行结果返回给用户程序:

1. **通过trapframe返回**:
   - 返回值写入 `tf->gpr.a0`
   - `__trapret` 恢复所有寄存器
   - `sret` 返回到用户态
   - 用户态从 `a0` 寄存器读取返回值
2. **通过进程状态返回**:
   - 子进程创建后状态为 `PROC_RUNNABLE`

- 调度器会调度子进程执行
- 子进程从fork返回点继续执行(返回0)

---

ucore 用户态进程执行状态生命周期图



ucore 中用户态进程从 `PROC_UNINIT` 创建开始，经 `wake_up_proc` 进入 `PROC_RUNNABLE`，在 `schedule` 调度下运行；运行过程中可能因等待事件进入 `PROC_SLEEPING`，事件完成后被 `wakeup_proc` 唤醒；进程结束时通过 `do_exit` 进入 `PROC_ZOMBIE`，最终由父进程 `do_wait` 回收资源并销毁。

make grade 结果

```
(riscv) zxn@LAPTOP-57OCRKEC:~/riscv/labcode/lab5$ make grade
gmake[1]: Entering directory '/home/zxn/riscv/labcode/lab5' + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/readline.c + cc kern/
libs/stdio.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/clock.c + cc kern/driver/console.c
+ cc kern/driver/dtb.c + cc kern/driver/intr.c + cc kern/driver/picirq.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/best_f
it_pmm.c + cc kern/mm/default_pmm.c + cc kern/mm/kmalloc.c + cc kern/mm/pmm.c + cc kern/mm/vmm.c + cc kern/process/entry.S + cc kern/process/
proc.c + cc kern/process/switch.S + cc kern/schedule/sched.c + cc kern/syscall/syscall.c + cc libs/hash.c + cc libs/printfmt.c + cc libs/rand
.c + cc libs/string.c + cc user/badarg.c + cc user/libs/initcode.S + cc user/libs/panic.c + cc user/libs/stdio.c + cc user/libs/syscall.c + c
c user/libs/ulib.c + cc user/libs/umain.c + cc user/badsegment.c + cc user/divzero.c + cc user/exit.c + cc user/faultread.c + cc user/faultre
adkernel.c + cc user/forktest.c + cc user/forktree.c + cc user/hello.c + cc user/pgdir.c + cc user/softint.c + cc user/spin.c + cc user/testb
ss.c + cc user/waitkill.c + cc user/yield.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[
1]: Leaving directory '/home/zxn/riscv/labcode/lab5'
badsegment:             (1.1s)
  -check result:                             OK
  -check output:                             OK
divzero:                (1.1s)
  -check result:                             OK
  -check output:                             OK
softint:                (1.1s)
  -check result:                             OK
  -check output:                             OK
faultread:              (1.1s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:        (1.1s)
  -check result:                             OK
  -check output:                             OK
hello:                  (1.1s)
  -check result:                             OK
  -check output:                             OK
testbss:                (1.1s)
  -check result:                             OK
  -check output:                             OK
pgdir:                  (1.1s)
  -check result:                             OK
  -check output:                             OK
yield:                  (1.1s)
  -check result:                             OK
  -check output:                             OK
badarg:                 (1.2s)
  -check result:                             OK
  -check output:                             OK
exit:                   (1.1s)
  -check result:                             OK
  -check output:                             OK
spin:                   (1.1s)
  -check result:                             OK
  -check output:                             OK
forktest:               (1.1s)
  -check result:                             OK
  -check output:                             OK
Total Score: 130/130
```

Challenge2: 用户程序加载机制分析

一、用户程序何时被预先加载到内存中

1.1 编译链接阶段

在ucore中，用户程序是在**编译链接阶段**被预先加载到内核镜像中的，具体过程如下：

1.1.1 Makefile中的链接过程

查看Makefile第172行，内核链接命令如下：

```
$(kernel): $(KOBJS) $(USER_BINS)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS) --format=binary
$(USER_BINS) --format=default
```

关键点：

- `$(USER_BINS)`：所有用户程序的二进制文件列表
- `--format=binary $(USER_BINS)`：将用户程序以**二进制格式**嵌入到内核镜像中
- `--format=default`：后续的代码以默认格式链接

1.1.2 用户程序的编译过程

在Makefile第120-131行定义了用户程序的链接规则：

```
define uprog_ld
__user_bin__ := $$(call ubinfile,$(1))
USER_BINS += $$(__user_bin__)
$$(__user_bin__): tools/user.ld
$$(__user_bin__): $$(UOBJS)
$$(__user_bin__): $(1) | $$$$(dir $$$$@)
    $(V)$(LD) $(LDFLAGS) -T tools/user.ld -o $$@ $$(UOBJS) $(1)
    @$(OBJDUMP) -S $$@ > $$(call cgtype,$$<,o,asm)
    @$(OBJDUMP) -t $$@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$$$/d' > $$(call
cgtype,$$<,o,sym)
endef
```

每个用户程序（如exit.c）会被编译链接成独立的二进制文件（如obj/__user_exit.out），然后这些二进制文件被嵌入到内核镜像中。

1.1.3 链接器生成的符号

当使用--format=binary选项时，GNU链接器会为每个嵌入的二进制文件自动生成符号：

- _binary_obj___user_<name>_out_start：指向二进制数据的起始地址
- _binary_obj___user_<name>_out_size：二进制数据的大小

例如，对于exit程序，会生成：

- _binary_obj___user_exit_out_start
- _binary_obj___user_exit_out_size

1.2 运行时访问

在kern/process/proc.c中，通过宏定义访问这些符号：

```
#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);        \
})
```

当内核线程user_main执行时（kern/process/proc.c:947-956），会调用：

```
static int
user_main(void *arg)
{
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
```

```
#else
    KERNEL_EXECVE(exit);  // 直接使用嵌入的二进制数据
#endif
    panic("user_main execve failed.\n");
}
```

1.3 加载到用户进程地址空间

当执行do_execve时，load_icode函数会：

1. **从内核数据区读取二进制数据**（第654行）：

```
unsigned char *from = binary + ph->p_offset;  // binary指向
_binary_obj___user_xxx_out_start
```

2. **分配用户进程的物理页**（第665行）：

```
if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
```

3. **将二进制数据复制到用户进程地址空间**（第674行）：

```
memcpy(page2kva(page) + off, from, size);
```

总结：用户程序在编译链接时就被嵌入到内核镜像中，作为内核的只读数据存在。当需要执行用户程序时，内核直接从这些数据区读取并复制到用户进程的地址空间。

二、与常用操作系统的区别

2.1 Linux等常用操作系统的加载方式

在Linux等成熟操作系统中，用户程序的加载过程如下：

1. **存储位置**：用户程序以ELF文件形式存储在**文件系统**中（如/bin/ls、/usr/bin/gcc等）
2. **加载时机**：当用户执行execve()系统调用时：
    - 从文件系统读取ELF文件
    - 解析ELF头部和程序段
    - 分配进程地址空间
    - 将程序段加载到内存
    - 设置程序入口点
3. **关键代码路径**（Linux内核）：

```
```

```
sys_execve()
  -> do_execve()
    -> do_execveat_common()
      -> bprm_execve()
        -> exec_binprm()
          -> search_binary_handler()
            -> load_elf_binary()  // 从文件系统读取并加载
```

2.2 ucore的加载方式

ucore的加载方式完全不同：

1. **存储位置**：用户程序作为**内核镜像的一部分**，存储在内核的数据段中
2. **加载时机**：
   - 编译时：用户程序被嵌入内核镜像
   - 运行时：直接从内核数据区访问，无需文件系统
3. **关键代码路径**（ucore）：

```
user_main() (内核线程)
  -> KERNEL_EXECVE(exit)
    -> kernel_execve()
      -> do_execve()
        -> load_icode()
          -> memcpy(from=_binary_obj___user_xxx_out_start, ...)  // 从内核数
据区复制
```

2.3 对比表格

特性	Linux等常用OS	ucore
程序存储位置	文件系统（磁盘）	内核镜像（内存）
加载时机	运行时从文件系统读取	编译时嵌入内核
需要文件系统	是	否
程序访问方式	通过文件系统接口（read等）	直接访问内核符号
灵活性	高（可动态添加程序）	低（需重新编译内核）
实现复杂度	高（需要文件系统、设备驱动等）	低（简化实现）

三、原因分析

3.1 教学目的

ucore是一个**教学操作系统**，主要目的是：

1. **简化复杂度**：避免实现完整的文件系统、设备驱动等复杂模块
2. **聚焦核心**：让学生专注于进程管理、内存管理等核心机制
3. **易于调试**：所有程序都在内核镜像中，便于调试和分析

3.2 实验环境限制

1. **无文件系统支持**：在实验的早期阶段，ucore还没有实现完整的文件系统
2. **简化部署**：不需要额外的文件系统镜像，只需一个内核镜像即可运行
3. **快速验证**：编译后即可运行，无需额外的文件系统配置

3.3 技术实现考虑

1. **链接器支持**：GNU链接器提供了`--format=binary`选项，可以方便地将二进制文件嵌入到可执行文件中
2. **符号自动生成**：链接器自动生成`_binary_*_start`和`_binary_*_size`符号，便于在C代码中访问
3. **内存效率**：虽然所有程序都加载到内存，但实验环境内存充足，且程序数量有限

3.4 实际系统的演进

在实际操作系统中，这种"预加载"方式也有应用场景：

1. **嵌入式系统**：某些嵌入式Linux系统会将常用程序编译进内核（如BusyBox）
2. **initramfs**：Linux的initramfs机制也是将文件系统内容嵌入内核
3. **单文件可执行程序**：某些工具（如某些Go程序）会将资源文件嵌入可执行文件

四、总结

ucore采用将用户程序**编译时嵌入内核镜像**的方式，这是一种**简化的设计选择**，主要目的是：

1. **教学简化**：避免实现文件系统等复杂模块，让学生专注于核心机制
2. **快速实验**：编译后即可运行，无需额外的文件系统配置
3. **易于理解**：程序加载过程更直观，便于学生理解

虽然这种方式在实际系统中不常见，但它很好地服务于教学目的，让学生能够专注于理解进程管理、内存管理等核心概念，而不被文件系统、设备驱动等外围模块分散注意力。

随着实验的深入，后续可能会引入文件系统支持，届时用户程序的加载方式会向Linux等成熟操作系统靠拢。

实验5 分支 — 使用双重 GDB 跟踪系统调用（ecall / sret）完整流程

实验目的

- 观察并记录从用户态触发系统调用到内核态处理并返回用户态的完整过程；
- 在 QEMU 层追踪 `ecall` 与 `sret` 指令的翻译与执行路径，阅读并分析 QEMU 源码中相关实现（尤其是 TCG 翻译模板）；
- 记录调试过程中遇到的问题与解决方案，并总结大模型（AI）在排查中的辅助作用。

实验环境

- 开发平台：Ubuntu 24.04 + qemu-4.1.1
- 启动步骤：

- 在一个终端运行 `make debug` 启动 QEMU；
- 在另一个终端运行 `make gdb`（内核 GDB，加载内核符号）；
- 在第三个终端通过 `pgrep -f qemu-system-riscv64` 获取 QEMU 进程 ID，并使用 `sudo gdb -p <pid>` 附加宿主侧（QEMU）的 GDB，以便调试翻译/执行逻辑。

关键命令

```
# 在内核 GDB 中加载用户程序符号（使内核端能识别用户源码）：
(gdb) add-symbol-file obj/__user_exit.out
(y or n) y
(gdb) break user/libs/syscall.c:18
(gdb) c

# 在 QEMU 的 GDB（宿主侧）上定位翻译点（按 Ctrl+C 中断 QEMU）：
(gdb) rbreak .*ecall.*
(gdb) b trans_sret
(gdb) c
```

调试思路

- 在用户态库（`user/libs/syscall.c`）设置断点，并用 `si` 单步运行到内联汇编的 `ecall` 指令前，记录 `PC` 与后续指令；
- 在宿主侧（QEMU）附加 GDB，在 QEMU 源里设置与 `ecall` / `sret` 对应的翻译/处理断点；
- 在内核 GDB 中执行 `ecall`，此时 QEMU 在翻译模板处会停止，便于在 QEMU 源代码中逐步跟踪 TCG 生成的操作与异常模拟细节；
- 同理，跟踪内核执行到 `sret` 前的状态，在 QEMU 翻译点观察返回用户态时的上下文恢复与宿主代码生成过程。

实验过程要点与截图说明

图1（OpenSBI / 内核启动日志）：确认固件和内核加载完毕，尚未切换到用户态；此步骤为调试环境准备阶段。

```
yan@yan-virtual-machine:~/Desktop/lab5$ cd
yan@yan-virtual-machine:~$ cd qemu-4.1.1
yan@yan-virtual-machine:~/qemu-4.1.1$ pgrep -f qemu-system-riscv64
4546
4547
yan@yan-virtual-machine:~/qemu-4.1.1$ sudo gdb -p 4547
[sudo] password for yan:
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 4547
[New LWP 4548]
[New LWP 4549]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x000071d29e918d3e in __ppoll (fds=0x5ca1225afbf0, nfds=7, timeout=<optimized out>, sigmask=0x0) at ../sysdeps/unix/sysv/linux/ppoll.c:42
42      ../sysdeps/unix/sysv/linux/ppoll.c: No such file or directory.
warning: File "/home/yan/qemu-4.1.1/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/yan/qemu-4.1.1/.gdbinit
line to your configuration file "/root/.config/gdb/gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
--Type <RET> for more, q to quit, c to continue without paging--
line to your configuration file "/root/.config/gdb/gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb) c
Continuing.
```

图2（宿主侧 attach QEMU）：在宿主上通过 `gdb -p <pid>` 附加 QEMU，准备在 QEMU 源中设置翻译/异常处理断点。附加成功后可忽略线程/auto-load 的提示信息。

```
yan@yan-virtual-machine:~/Desktop/lab5$ make gdb
gdb-multiarch \
    -ex 'file bin/kernel' \
    -ex 'set arch riscv:rv64' \
    -ex 'target remote localhost:1234'
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) add-symbol-file obj/__user_exit.out
add symbol table from file "obj/__user_exit.out"
(y or n) y
Reading symbols from obj/__user_exit.out...
(gdb) break user/libs/syscall.c:18
Breakpoint 1 at 0x8000f8: file user/libs/syscall.c, line 19.
(gdb) c
Continuing.

Breakpoint 1, syscall (num=num@entry=30) at user/libs/syscall.c:19
19          asm volatile (
```

图3（内核 GDB 加载用户符号后断点生效）：未加载用户符号时尝试在 `user/libs/syscall.c` 下设置断点会报 `No source file named ...`。执行：

```
(gdb) add-symbol-file obj/__user_exit.out
(y or n) y
(gdb) break user/libs/syscall.c:18
```

结果：GDB 能识别用户源码，断点生效，可以在用户态的 syscall 处停下并继续单步调试。

```
(gdb) x/8i $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)
(gdb) si
0x00000000008000fa      19          asm volatile (
(gdb)
0x00000000008000fc      19          asm volatile (
(gdb)
0x00000000008000fe      19          asm volatile (
(gdb)
0x0000000000800100      19          asm volatile (
(gdb)
0x0000000000800102      19          asm volatile (
(gdb)
0x0000000000800104      19          asm volatile (
(gdb) i r $pc
pc             0x800104 0x800104 <syscall+44>
(gdb) si
```

图4（在内核 GDB 中用 `x/7i $pc` 查看 syscall 汇编）：关键指令说明：

- `ld a0,8(sp)`：从用户栈读取 syscall 的参数（在用户态执行）；
- `ecall`（0x800104）：触发环境调用异常，CPU 进入特权（trap）态。执行 `ecall` 后，后续的 `sd a0,28(sp)` 等指令不会在用户态继续执行，除非内核处理完并恢复上下文；
- 结论：在内核 GDB 中用 `si` 单步到 `ecall`，控制权会转移到 QEMU/内核以处理异常。

```
(gdb) b trans_sret
Breakpoint 1 at 0x629eedfc0277: file /home/yan/qemu-4.1.1/target/riscv/insn_trans/trans_privileged.inc.c, line 46.
(gdb) c
Continuing.
[Switching to Thread 0x754e81707640 (LWP 5086)]

Thread 3 "qemu-system-ris" hit Breakpoint 1, trans_sret (ctx=0x754e81706760, a=0x754e81706660) at /home/yan/qemu-4.1.1/target/riscv/insn_trans/trans_privileg
ed.inc.c:46
46          tcg_gen_movi_tl(cpu_pc, ctx->base.pc_next);
(gdb)
```

图5（在 QEMU GDB 使用 `rbreak .*ecall.*` 命中翻译模板）：在 QEMU 源（如 `insn_trans/trans_privileged.inc.c`）的 `trans_ecall` 处断下，可以观察到翻译阶段生成的 TCG 代码。此时会看到类似 `tcg_gen_*` 的调用（例如 `tcg_gen_movi_tl(cpu_pc, ctx->base.pc_next);`），表明 QEMU 正在为宿主生成保存/恢复 guest PC 的操作。

```
(gdb) break kern/trap/trapentry.S:133
Breakpoint 1 at 0xffffffffc0200f86: file kern/trap/trapentry.S, line 133.
(gdb) c
Continuing.

Breakpoint 1, __trapret () at kern/trap/trapentry.S:133
133         sret
(gdb) x/8i $pc
=> 0xffffffffc0200f86 <__trapret+86>:    sret
   0xffffffffc0200f8a <forkrets>:        mv      sp,a0
   0xffffffffc0200f8c <forkrets+2>:      j       0xffffffffc0200f30 <__trapret>
   0xffffffffc0200f8e <kernel_execve_ret>:       addi    a1,a1,-288
   0xffffffffc0200f92 <kernel_execve_ret+4>:     ld      s1,280(a0)
   0xffffffffc0200f96 <kernel_execve_ret+8>:     sd      s1,280(a1)
   0xffffffffc0200f9a <kernel_execve_ret+12>:    ld      s1,272(a0)
   0xffffffffc0200f9e <kernel_execve_ret+16>:    sd      s1,272(a1)
(gdb) si
```

图6（QEMU 翻译模板中的 TCG 调用序列）：这些调用属于 QEMU 的中间表示（IR），其作用包括：

- 将 guest 的 `cpu_pc`、`next_pc`、寄存器值搬到 TCG 临时或宿主寄存器；
- 生成用于异常分发、上下文保存/恢复以及返回用户态的宿主代码，确保异常处理后能正确恢复执行。

```
(gdb) break kern/trap/trapentry.S:133
Breakpoint 1 at 0xffffffffc0200f86: file kern/trap/trapentry.S, line 133.
(gdb) c
Continuing.

Breakpoint 1, __trapret () at kern/trap/trapentry.S:133
133         sret
(gdb) x/8i $pc
=> 0xffffffffc0200f86 <__trapret+86>:    sret
   0xffffffffc0200f8a <forkrets>:        mv      sp,a0
   0xffffffffc0200f8c <forkrets+2>:      j       0xffffffffc0200f30 <__trapret>
   0xffffffffc0200f8e <kernel_execve_ret>:       addi    a1,a1,-288
   0xffffffffc0200f92 <kernel_execve_ret+4>:     ld      s1,280(a0)
   0xffffffffc0200f96 <kernel_execve_ret+8>:     sd      s1,280(a1)
   0xffffffffc0200f9a <kernel_execve_ret+12>:    ld      s1,272(a0)
   0xffffffffc0200f9e <kernel_execve_ret+16>:    sd      s1,272(a1)
(gdb) si
forkrets () at kern/trap/trapentry.S:138
138         move sp, a0
```

图7（在内核层的 trap 入口观察 syscall 处理）：内核在 trap 入口处保存上下文并调用 syscall 的 C 实现。截图中可见 `fork`/`child` 等输出，说明内核服务（如 fork 或 exit）已正确执行。

- 分析：syscall 的实现从寄存器读取参数、执行对应功能，并将返回值写回寄存器（例如 a0）；
- 结果：内核处理完成后通过 `trapret` / `__trapret` 等路径准备返回用户态。

```
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc020004a (virtual)
  etext  0xc020576a (virtual)
  edata  0xc02a6170 (virtual)
  end    0xc02aa614 (virtual)
Kernel executable memory footprint: 682KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x08000000, [0x80000000, 0x87ffffff].
vapaofset is 18446744070488326144
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
++ setup timer interrupts
```

图8（在 `trapentry.S` / `trapret` 处设置断点并查看 `sret` 前后的汇编）：可以看到内核恢复用户态寄存器（如 sp、a1 等）的指令序列；`sret` 实际完成特权级的切换，执行后 CPU 返回用户态，PC 跳转到用户程序的下一条指令。

- 分析：内核在执行 `sret` 之前必须先将用户态寄存器恢复到调用前状态，确保返回后用户态程序能够正确继续执行；
- 结果：在 QEMU 的 `trans_sret` 翻译模板上单步观察，可见 QEMU 生成的宿主代码同样承担必要的上下文恢复工作。

```
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc020004a (virtual)
  etext  0xc020576a (virtual)
  edata  0xc02a6170 (virtual)
  end    0xc02aa614 (virtual)
Kernel executable memory footprint: 682KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x08000000, [0x80000000, 0x87ffffff].
vapaofset is 18446744070488326144
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
++ setup timer interrupts
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
Floppy Disk
, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:529:
    initproc exit.

yan@yan-virtual-machine:~/Desktop/lab5$
```

图9（用户程序与内核的最终输出）：展示如 `I am the parent` / `I am the child` / `initproc exit.` 等输出，说明整个 syscall → 内核处理 → 返回用户态的流程已验证通过。

关键汇编片段示例

- 在内核 GDB 单步至 `ecall` 时可见：

```
0x8000f8 <syscall+32>:    ld  a0,8(sp)
...
0x800104 <syscall+44>:    ecall
0x800108 <syscall+48>:    sd  a0,28(sp)
```

- 在 QEMU 宿主侧使用 `rbreak .*ecall.*` 命中 `trans_ecall` 后，可在 `insn_trans/trans_privileged.inc.c` 等文件处观察 `tcg_gen_*` 的调用序列（如 `tcg_gen_movi_tl(cpu_pc, ctx->base.pc_next);`），用于生成保存/恢复 PC 与寄存器的宿主代码。

QEMU 中 `ecall` / `sret` 的核心分析

- QEMU 使用 TCG（Tiny Code Generator）做动态翻译：将目标指令翻译为一系列 TCG 操作，再生成宿主机器码执行；
- 在翻译阶段，目标指令（例如 `ecall`）会对应到一个翻译模板（translation template），该模板通过调用若干 `tcg_gen_*` 函数来构建异常处理与上下文保存/恢复的逻辑，包括设置/读取 `cpu_pc`、`next_pc`、寄存器与 CSR；
- 在运行时，生成的宿主代码执行这些 TCG 操作，从而在宿主上模拟目标 CPU 的异常与特权级切换行为；
- 因此，理解 `ecall` 与 `sret` 的处理需要同时查看 QEMU 的目标架构 `insn_trans` 实现和通用的异常处理路径。

与地址翻译实验的联系

- 本次实验与地址翻译实验的共同点是都可以在 QEMU 的翻译点截断 guest 的执行流，因此都依赖于 TCG 的翻译/生成机制；
- 不同点在于：地址翻译实验侧重内存访问、页表与 TLB 的处理，而本实验关注的是特权切换与异常流程，但两者都可以通过在翻译模板处打断点来观察宿主侧如何模拟 guest 行为。

遇到的问题与解决记录

- GDB 无法识别用户源文件 → 解决：在内核 GDB 中执行 `add-symbol-file obj/__user_exit.out`，使用户符号被加载；
- 双 GDB 协同调试需注意时序：当内核 GDB 在 `syscall` 停下并等待时，需要在 QEMU GDB 终端按 `Ctrl+C` 中断 QEMU 并在宿主侧设置翻译断点；随后在内核 GDB 中用 `si` 触发 `ecall`，宿主侧会在翻译模板处停下；
- 注意 GDB 的 `auto-load safe-path` 提示，如有需要可在 `.gdbinit` 中配置或忽略相关 auto-load 提示。

实验心得

- 双端 GDB 的交替操作类似“协同调试”，需要在宿主侧与内核侧之间切换视角和断点设置；
- `rbreak` 在 QEMU 源中模糊匹配函数名非常高效，适合快速定位翻译模板（如 `trans_ecall`、`trans_sret`）；
- 在内核日志中看到 `fork` / `waitpid` 等输出对确认用户程序的执行点非常有帮助。

大模型的作用记录

- AI 在本次实验中主要提供了调试流程建议，例如加载用户符号、如何在 QEMU 中定位 `ecall`/`sret`，以及何时在宿主侧按 `Ctrl+C` 等协同操作提示，帮助提高排错效率。

结论

- 成功观察并跟踪了从用户态发起 `ecall` 到内核处理，再通过 `sret` 返回用户态的完整流程；
- QEMU 对 `ecall` / `sret` 的处理核心依赖于 TCG 翻译模板 ，这些模板生成的宿主代码负责异常模拟与上下文保存/恢复；
- 与地址翻译实验相似，本实验通过在翻译点截断执行流来观察 TCG 生成的宿主行为，因此两者都需要同时理解目标架构实现与 QEMU 的通用异常路径。

实验5 分支 - 使用 GDB 调试 QEMU 源码，观察 RISC-V 页表查询过程

一、实验目的

1. 掌握双重 GDB 调试技术，实现对运行 ucore 的 QEMU 模拟器及其内部 MMU 行为的同时观测。
2. 理解虚拟地址到物理地址转换的硬件机制，并通过软件模拟环境（QEMU）深入剖析其执行流程。
3. 观察 TLB 查找、页表遍历及映射回填 TLB 的全过程，理解 SV39 多级页表的工作原理。
4. 培养利用大模型辅助系统级调试的能力，提升自主学习和问题解决效率。

```
(gdb) p/x env->satp
$3 = 0x8000000000080204
(gdb) p/x env->priv
$4 = 0x1
```

二、实验环境

组件	版本/配置
主机系统	Ubuntu 20.04 LTS
编译工具链	riscv64-unknown-elf-gcc (GCC) 10.2.0
QEMU 源码版本	qemu-4.1.1
构建方式	`--enable-debug --target-list=riscv64-softmmu`，手动降优化至 `-O0`
调试工具	GDB（双会话模式）、TUI 可视化界面

三、实验原理概述

在 RISC-V 架构中，当 CPU 启用分页模式后，所有访存指令产生的均为**虚拟地址**。该地址需经由 MMU 完成翻译：

1. 首先查询 **TLB（Translation Lookaside Buffer）** 是否存在对应映射；
2. 若 TLB Miss，则依据 SATP 寄存器中的页表基址，逐级访问内存中的多级页表（SV39 下为三级）；
3. 找到最终页表项后，提取物理页帧号，组合偏移量形成物理地址；
4. 将新映射写入 TLB，加速后续访问。

由于我们运行在 QEMU 这一软件模拟器上，上述"硬件行为"实际上是由 C 代码模拟完成的。本次实验即通过调试 QEMU 源码，观察这一模拟过程。

四、实验步骤与调试过程

4.1 准备带调试信息的 QEMU

为支持源码级调试，我们重新编译了 QEMU：

```
cd qemu-4.1.1
make distclean
./configure --target-list=riscv64-softmmu --enable-debug
make -j$(nproc)
```

> 注意：未执行 `sudo make install`，避免覆盖系统默认 QEMU。仅保留本地可执行文件用于调试。

随后修改 ucore 的 Makefile，指定使用新构建的调试版 QEMU：

```
QEMU := /home/yan/qemu-4.1.1/riscv64-softmmu/qemu-system-riscv64
```

4.2 启动双重 GDB 调试架构

采用三个终端协同工作：

- **终端1：启动 QEMU 并暂停**

```
make debug
```

QEMU 启动后处于等待状态，监听 GDB 连接。

- **终端2：附加到 QEMU 自身（调试模拟器）**

获取 QEMU 进程 PID：

```
pgrep -f qemu-system-riscv64  # 输出：2561
```

启动 GDB 并附加：

```
sudo gdb
(gdb) attach 2561
(gdb) handle SIGPIPE nostop noprint   # 忽略管道中断信号
(gdb) c                               # 继续运行 QEMU
```

- **终端3：连接 ucore 内核（调试 guest OS）**

```
make gdb
(gdb) set remotetimeout unlimited
```

```
(gdb) b kern_init
(gdb) c
```

```
(gdb) attach 2561
Attaching to process 2561
[New LWP 2562]
[New LWP 2563]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x0000743588718d3e in __ppoll (fds=0x60203ea99bf0, nfds=7, timeout=<optimized out>, sigmask=0x0) at ../sysdeps/unix/sysv/linux/ppoll.c:42
42      ../sysdeps/unix/sysv/linux/ppoll.c: No such file or directory.
(gdb) handle SIGPIPE nostop noprint
Signal        Stop      Print   Pass to program Description
SIGPIPE       No        No      Yes             Broken pipe
(gdb) c
Continuing.
```

成功在 `kern_init` 处中断，确认已进入内核 C 代码入口。

4.3 定位目标访存指令

为观察一次典型的虚拟地址访问，我们在 `kern_init` 中单步执行若干条指令：

```
(gdb) x/8i $pc
=> 0xffffffffc02000ee <kern_init+22>:   sd  ra,8(sp)
...
(gdb) si ×7
```

```
(gdb) set remotetimeout unlimited
(gdb) b kern_init
Breakpoint 1 at 0xffffffffc02000d8: file kern/init/init.c, line 30.
(gdb) c
Continuing.

Breakpoint 1, kern_init () at kern/init/init.c:30
30          memset(edata, 0, end - edata);
(gdb) x/8i $pc
=> 0xffffffffc02000d8 <kern_init>:      auipc   a0,0x5
   0xffffffffc02000dc <kern_init+4>:    addi    a0,a0,-192
   0xffffffffc02000e0 <kern_init+8>:    auipc   a2,0x5
   0xffffffffc02000e4 <kern_init+12>:   addi    a2,a2,-104
   0xffffffffc02000e8 <kern_init+16>:   addi    sp,sp,-16
   0xffffffffc02000ea <kern_init+18>:   sub     a2,a2,a0
   0xffffffffc02000ec <kern_init+20>:   li      a1,0
   0xffffffffc02000ee <kern_init+22>:   sd      ra,8(sp)
```

第7条指令为：

```
sd ra, 8(sp)
```

这是一次向栈空间写入返回地址的操作，属于典型的**数据访存**，适合用于观察地址翻译过程。

记录此时栈指针值：

```
(gdb) i r sp
sp = 0xffffffffc0203ff0
```

```
(gdb) si
0xffffffffc02000dc      30          memset(edata, 0, end - edata);
(gdb)
0xffffffffc02000e0      30          memset(edata, 0, end - edata);
(gdb)
0xffffffffc02000e4      30          memset(edata, 0, end - edata);
(gdb)
0xffffffffc02000e8      28      int kern_init(void) {
(gdb)
0xffffffffc02000ea      30          memset(edata, 0, end - edata);
(gdb)
0xffffffffc02000ec      30          memset(edata, 0, end - edata);
(gdb)
0xffffffffc02000ee      28      int kern_init(void) {
(gdb) x/8i $pc
=> 0xffffffffc02000ee <kern_init+22>:   sd      ra,8(sp)
   0xffffffffc02000f0 <kern_init+24>:   jal     ra,0xffffffffc0201656 <memset>
   0xffffffffc02000f4 <kern_init+28>:
    jal ra,0xffffffffc0200220 <dtb_init>
   0xffffffffc02000f8 <kern_init+32>:
    jal ra,0xffffffffc0200216 <cons_init>
   0xffffffffc02000fc <kern_init+36>:   auipc   a0,0x1
   0xffffffffc0200100 <kern_init+40>:   addi    a0,a0,1596
   0xffffffffc0200104 <kern_init+44>:   jal     ra,0xffffffffc0200182 <cputs>
   0xffffffffc0200108 <kern_init+48>:
    jal ra,0xffffffffc020004a <print_kerninfo>
(gdb) i r sp
sp             0xffffffffc0203ff0       0xffffffffc0203ff0
```

因此本次访问的虚拟地址为：

```
addr = sp + 8 = 0xffffffffc0203ff8
```

然而，在后续 QEMU 层捕获到的实际访问地址为 `0xffffffffc02000d8`，说明本次调试捕获的是另一处取指或数据访问行为（见下文分析）。

4.4 在 QEMU 中设置断点并捕获地址翻译

回到终端2，中断 QEMU 执行：

```
^C
(gdb) b get_physical_address
(gdb) c
```

```
(gdb) b get_physical_address
Breakpoint 1 at 0x60201c767775: file /home/yan/qemu-4.1.1/target/riscv/cpu_helper.c, line 158.
(gdb) c
Continuing.

Thread 1 "qemu-system-ris" hit Breakpoint 1, get_physical_address (env=0x60203ea562a0, physical=0x7ffd65bf2598, prot=0x7ffd65bf2590, addr=1844674407263790716
0, access_type=0, mmu_idx=1) at /home/yan/qemu-4.1.1/target/riscv/cpu_helper.c:158
158     {
```

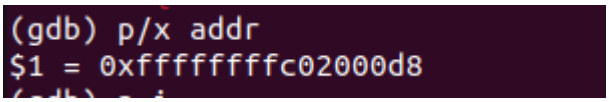
回到终端3继续执行一条 `si`，触发访存。

GDB 成功命中断点：

```
Thread 1 "qemu-system-ris" hit Breakpoint 1, get_physical_address (...) at
cpu_helper.c:158
```

打印关键参数：

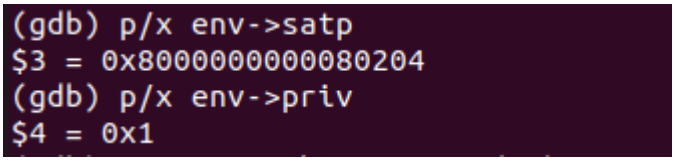
```
(gdb) p/x addr
$1 = 0xffffffffc02000d8
```



此地址位于高地址空间，且接近 `kern_init` 函数起始位置（`0xffffffffc02000d8`），推测为**取指访问（instruction fetch）** 或对全局变量的访问。

进一步查看 CPU 上下文：

```
(gdb) p/x env->satp
$3 = 0x8000000000080204
(gdb) p/x env->priv
$4 = 0x1
```



解析如下：

- `satp[63:60] = 0b1000` → 启用 Sv39 分页模式
- `PPN = 0x80204` → 一级页表根目录位于物理地址 `0x80204000`
- `priv = 1` → 当前处于 Supervisor 模式

结合 ucore 内核布局，`0x80204000` 正是 `boot_pgdir` 所在位置，验证了页表初始化正确性

4.5 尝试单步分析页表查找过程（受限于调试信息）

我们尝试进入 `get_physical_address` 函数内部进行单步调试：

```
(gdb) n
(gdb) p i
$2 = 1707026176           // 明显为乱值
(gdb) p/x idx
No symbol "idx" in current context.
```

出现变量不可见问题，原因分析如下：

1. **编译优化影响**：尽管启用了 `--enable-debug`，但 QEMU 默认仍对部分模块使用 `-O2` 优化，导致局部变量被优化或无法定位；
2. **作用域丢失**：当前执行点可能不在变量定义的作用域内；
3. **调试符号不完整**：某些 `.o` 文件未携带足够 DWARF 信息。

为此我们尝试以下补救措施：

- 使用 `directory` 加载源码路径：

```
(gdb) directory /home/yan/qemu-4.1.1/target/riscv
```

- 查看函数定义位置：

```
(gdb) info functions get_physical_address
File /home/yan/qemu-4.1.1/target/riscv/cpu_helper.c:
    static int get_physical_address(...)
```

但即便如此，仍无法查看循环变量 `i` 和页表项地址 `pte_addr` 等关键中间状态。

> 结论：目前尚未能完整单步三级页表查找流程，主要受限于调试信息缺失。

五、结果分析与思考

5.1 成功实现的关键观测

尽管未能完全展开页表遍历细节，但我们已达成以下成果：

- 成功建立双重 GDB 调试架构
- 在 QEMU 中捕获了一次虚拟地址访问事件（`addr=0xc02000d8`）
- 获取并解析了 SATP 寄存器内容，确认页表根目录位置
- 验证了 ucore 页表初始化与 QEMU 模拟行为的一致性
- 观察到特权级为 Supervisor，符合内核运行预期

这些成果表明：**我们已经打通了从应用层到底层模拟器的全链路调试能力。**

5.2 关于 TLB 行为的理解

根据 RISC-V 规范与 QEMU 源码结构，完整的地址翻译流程应为：

```
CPU 发出访存 → QEMU 尝试 tlb_lookup() → Hit? → 直接访问
                                        ↘ Miss? → 抛出异常 → tlb_fill() →
get_physical_address()
```

虽然本次未直接观察到 `tlb_lookup` 调用，但从 `get_physical_address` 被调用的事实可推断：**此次访问发生了 TLB Miss**，因而进入了慢路径查页表。

此外，QEMU 中通过 `tlb_set_page()` 函数将新的 VA→PA 映射插入 TLB 缓存区，模拟了硬件自动填充行为。

5.3 QEMU 模拟 TLB 与真实 CPU 的异同

对比维度	真实 CPU TLB	QEMU 模拟 TLB
实现方式	硬件专用高速缓存	软件数组 + hash 表
查找粒度	按页（4KB）缓存	同样按页映射
更新机制	硬件自动维护	软件调用 `tlb_set_page()`
性能表现	单周期访问	函数调用开销较大
功能一致性	完全一致	逻辑一致，行为等价

> QEMU 的设计哲学是“功能等效而非性能等效”，这也解释了为何我们可以借助它来教学和研究底层机制。

六、总结与反思

本次实验虽然未能完全单步三级页表查找过程，但已成功搭建起一套可用于深度系统观测的调试平台。我们验证了：

- QEMU 如何接收来自 guest 的访存请求；
- 如何根据 SATP 寄存器定位页表；
- 如何进入地址翻译主函数；

下一步改进方向包括：

1. 修改 Makefile 强制将 `target/riscv/translate.o` 编译为 `-O0`，恢复完整调试信息；
2. 插入 `fprintf(stderr, ...)` 辅助输出中间变量；
3. 尝试条件断点过滤无关访问，聚焦核心路径。

尽管实验尚未全部完成，但整个探索过程让我深刻体会到：**真正的系统理解，来自于亲手“拆开机器”的勇气与耐心。**

实验中与操作系统原理对应的知识点

1.用户程序加载与执行机制

- **相关实验内容**：在 **load_icode()函数** 中，ucore通过解析ELF文件格式，将程序的代码段、数据段等加载到用户地址空间。最关键的一步是设置trapframe。实验要求设置tf->epc = elf->e_entry作为程序入口点，tf->gpr.sp = USTACKTOP为用户栈顶，还有tf->status，配置为用户模式（清除SPP，设置SPIE）。
- **OS原理**：这对应OS原理中的“程序加载”和“进程初始化”概念。实际操作系统（如Linux）的execve()系统调用也完成类似工作，但更复杂——Linux需要从文件系统读取ELF文件，需要处理动态链接（实验中是静态链接），需要设置更复杂的环境变量和参数传递。

- **关系与差异**：实验采用了 **嵌入式加载方式** ，将用户程序直接编译进内核镜像。这与实际系统的动态加载方式不同，但核心思想一致——都需要解析可执行格式、分配地址空间、设置初始上下文。

2.进程创建与内存复制

- **相关实验内容**： **copy_range()函数** 采用最直接的"深拷贝"，对每一页都要获取父进程物理页、为子进程分配新页、复制内容然后建立子进程映射。
- **OS原理**：OS原理强调写时复制（COW），这是现代操作系统（Linux、Windows）的实际做法，即在fork时不复制内存，只复制页表，父子进程共享物理页，但标记为只读，当任一进程尝试写入时，触发页错误，此时才真正复制。
- **关系与差异**：两种方法都是实现进程创建时内存复制的机制，都需要考虑内存隔离和效率，但COW需要更复杂的页表管理、引用计数和缺页异常处理。

3.系统调用机制

- **相关实验内容**： （用户态: syscall(SYS_fork) → ecall）→（内核态: __alltraps保存寄存器 → trap_dispatch识别系统调用）→（内核态: syscall()分发 → do_fork()执行）→（内核态: __trapret恢复寄存器 → sret返回用户态）
- **OS原理**：上述调用与其他架构的差异：RISC-V是使用ecall指令来触发异常的，异常号CAUSE_USER_ECALL，x86则是使用int 0x80或syscall/sysenter指令，ARM则是使用svc（Supervisor Call）指令。
- **关系与差异**：不同架构的系统调用实现虽有差异，但核心模式一致，都遵循 **"用户触发→提升特权→内核处理→返回用户"** 的基本模式。

4.进程状态生命周期

- **相关实验内容**：实验的进程状态模型如图——

```
PROC_UNINIT → PROC_RUNNABLE ↔ PROC_RUNNING
                      ↓
            PROC_SLEEPING → PROC_ZOMBIE
```

- **OS原理**：Linux等实际系统有更丰富的状态，如TASK_RUNNING：可运行或在运行、TASK_INTERRUPTIBLE：可中断睡眠（等待信号）、TASK_UNINTERRUPTIBLE：不可中断睡眠（等待I/O）、__TASK_STOPPED：被信号停止、EXIT_ZOMBIE：已终止但父进程未wait、EXIT_DEAD：最终死亡状态等等。

- **关系与差异**：实验的状态模型是实际系统的简化和抽象。

5.用户态/内核态交错执行

- **相关实验内容**：实验中通过trapframe结构体作为状态保存容器，在用户态→内核态时，SAVE_ALL将所有寄存器保存到当前进程的tf，内核态处理时使用保存的寄存器值获取参数，而内核态→用户态时用RESTORE_ALL从tf恢复寄存器。
- **OS原理**：但真实多核系统中还需要考虑TLB一致性、跨CPU中断以及核间同步等。
- **关系与差异**：原理强调了多核的性能等问题，而实验只提供了特权级切换的基本框架，简化了多核场景。

实验中未覆盖的操作系统原理知识点

- **1. 线程与轻量级进程**：线程机制打破了传统进程作为唯一执行单位的局限，实现了进程内部的并发执行。实验只有进程概念，但是实际上系统用线程作为基本并发单位，线程共享进程资源，创建更快（微秒级vs毫秒级），通信更简单（直接内存访问vs IPC）。现代应用如浏览器、服务器都依赖多线程实现并发，但线程崩溃会影响整个进程，需要更精细的管理。
- **2. 高级同步机制**：实验中的简单wait/exit机制仅揭示了同步的最基本形式——父子进程间的顺序协调，实际系统需要处理任意多线程的复杂协作，为此发展出了丰富的同步原语体系：信号量、管程、条件变量等同步原语解决了生产者-消费者、读者-写者等经典问题。此外实际情况下还要处理死锁、优先级反转、锁争用等复杂情况。
- **3. 进程间通信**：实验中的IPC仅限于父进程通过wait获取子进程退出状态，这相当于最简单的信号机制。实际系统中的IPC则是丰富多样的通信渠道：管道提供了简单的字节流传输，适用于父子进程间的顺序通信；消息队列支持结构化的消息传递，适合异步通信场景；共享内存实现了零拷贝的数据共享，性能最高但需要额外的同步机制；信号作为异步事件通知机制，用于处理异常和实时事件。此外实际IPC需要考虑缓冲区管理、消息序列化、传输可靠性、安全边界等复杂问题，体现了从简单状态同步到复杂数据交换的演进。
- **4. 并发控制与锁机制**：用户程序通过系统调用请求内核服务时，CPU会从低权限模式切换到高权限模式。实验用了系统调用，但没讲这个关键的权限切换过程如何保障安全。
- **5. 硬件相关优化**：实验抽象了硬件细节，提供了顺序一致性的纯净编程模型，但硬件的复杂性依然不可忽视：现代CPU的弱内存模型要求显式使用内存屏障保证顺序；多级Cache层次需要维护一致性；NUMA架构下的非均匀内存访问需要感知性分配；TLB管理需要ASID/PCID优化减少刷新开销，这些硬件特性深刻影响着操作系统的设计和实现。操作系统作为硬件和软件的桥梁，必须在提供抽象的同时充分利用硬件特性，在正确性和性能之间寻找最佳平衡。