

练习

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

练习0：填写已有实验

本实验依赖实验2/3/4。请把你做的实验2/3/4的代码填入本实验中代码中有“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行 lab5 的测试应用程序，可能需对已完成的实验2/3/4的代码进行进一步改进。

练习1：加载应用程序并执行（需要编码）

`do_execve`函数调用`load_icode`（位于`kern/process/proc.c`中）来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序。你需要补充`load_icode`的第6步，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好`proc_struct`结构中的成员变量`trapframe`中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的`trapframe`内容。

请在实验报告中简要说明你的设计实现过程。

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。

首先是调度器选中下一个进程指定为这个应用程序，然后调用`proc_run`将`current`进程指向新的进程，并把页表切换到当前页表

并用`switch_to`切换旧的到新的上下文，此时cpu就运行在了新的进程的内核栈上了，但是仍然处于之前老进程遗留的supervisor模式，在执行

```
_trapret();           // 进入 kern/trap/trapentry.s 中的通用返回用户态代码  
_trapret() 汇编序列
```

在这段内联汇编中先从新的trapframe中回复32个通用寄存器，然后将内核的栈指针指向用户分配的栈顶，`csrw sstatus, t0` → 恢复`sstatus`（`SPP=0, SPIE=1, FS=initial/clean`），这时将`epc`指向新用户进程的elf程序程序中的`entry`第一条指令开始执行，特权级：`Supervisor → User`。此时把`SPIE`的值赋回给`PIE`，恢复中断，地址 = `elf->e_entry` → 这就是用户程序的真正第一条指令（通常是程序入口的`_start`或编译器生成的CRT代码）。

练习2：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。

答：

在实现COW机制时，关键是复制内存页表的时机，在dofork阶段不再直接复制页表，而是增加一个指针指向这个物理地址空间，在真正遇到了需要写的时候再去复制页，在这之前父子页表都清写权限（只读）并增加页框引用计数。具体设计如下：

1. copy_range() (或 fork 时页表复制逻辑)：建立共享映射，清写权限，增加引用计数（**不 alloc 不 memcpy**）。
2. 页框管理 (struct Page) 支持引用计数: page_ref_inc(page), page_ref_dec(page)，并在计数到 0 时释放物理页。
3. 页表操作 (page_insert/page_remove) 正确处理引用计数与 PTE 权限变更。
4. 页错误处理 (page fault handler)：检测写缺页且为 COW 页，执行按需复制或直接恢复写权限（当引用计数 == 1）。
5. TLB flush (tlb_invalidate) 与同步。
6. (可选) 支持 MAP_SHARED 的不同语义: shared mapping 不走 COW (或按策略)。
7. 代码实现示例：

```
int copy_range_cow(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));
    do {
        pte_t *ptep = get_pte(from, start, 0);
        pte_t *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        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);
            assert(page != NULL);

            // 1) make both parent and child read-only (clear write bit)
            uint32_t new_perm = perm & ~PTE_W;

            // 2) map child to the same physical page (do NOT alloc or
memcp)
            int ret = page_insert(to, page, start, new_perm);
            if (ret < 0) return ret;

            // 3) update parent's pte to be read-only if it was writable
            if ((*ptep & PTE_W) != 0) {
                *ptep = (*ptep & ~PTE_W); // 保留 PTE_V, U 等位
                // 如果需要维护硬件页表同步，可能调用 mmu更新函数
            }
        }
    } while (start < end);
}
```

```

        }

        // 4) increase refcount
        page_ref_inc(page);
    }
    start += PGSIZE;
} while (start != 0 && start < end);
return 0;
}

```

8. 在写时遇到页错误的处理，就复制页再执行写：

```

int handle_cow_fault(pde_t *pgdir, uintptr_t addr, int fault_write)
{
    pte_t *ptep = get_pte(pgdir, addr, 0);
    if (ptep == NULL || !(*ptep & PTE_V)) return -E_INVAL;

    // 不是写导致的缺页，或页当前就可写，绕过
    if (!fault_write) return -E_INVAL;

    // 如果该页有写权限，那可能不是 COW（直接错误或其它）
    if ((*ptep & PTE_W) != 0) return -E_INVAL;

    struct Page *old = pte2page(*ptep);
    if (old == NULL) return -E_INVAL;

    // 原子读取引用计数（注意并发）
    if (page_ref_count(old) > 1) {
        // 多个引用：实际复制
        struct Page *new = alloc_page();
        if (!new) return -E_NO_MEM;
        memcpy(page2kva(new), page2kva(old), PGSIZE);

        // 更新页表：把当前页表项替换为 new（带写权限）
        int ret = page_insert(pgdir, new, addr, (*ptep & PTE_USER) |
PTE_W));
        if (ret < 0) {
            free_page(new); // 或 dec ref 并释放
            return ret;
        }
        page_ref_dec(old); // old 引用减少
    } else {
        // 只有 1 个引用（即当前进程独占），可以直接把 PTE 标成可写
        *ptep |= PTE_W;
        // 如果有需要，刷新 mmu 或 tlb
    }
    tlb_invalidate(addr);
    return 0;
}

```

练习3: 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)

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：

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?

- **do_fork**中的return syscall(SYS_fork);开始进入内核态, 然后分配一个新的进程控制块, 并按照传统复制, 将父进程的内核栈和上下文的trapframe复制给子进程, 让子进程从同一位置和状态恢复执行, 处理内存也是按照copyrange全复制, 然后把父子进程写进不同的trap frame, 父进程的fork返回子进程的pid, 然后返回用户态, 内核将要运行哪一个进程交给调度器恢复trapframe → 回到用户态。
- **do_exec** 用户程序执行:

```
execve("bin/sh", argv, envp);
```

然后陷入内核态, 清空旧地址空间 (摧毁旧的代码/数据/栈) 从 ELF 加载程序, 加载 .text、.data、堆栈, 建立新的 VMA (虚拟内存区域) 构造新的用户栈 (含参数与环境变量), 重置 trapframe 的 PC = 程序入口点。恢复到用户态内核执行 return-from-trap, 跳到新程序的入口, 用户进程开始运行完全新的一套指令。

- **do_wait** 用户态

```
pid = wait(&status);
```

陷入内核态, 检查是否有已退出的子进程

如果没有: **挂起**当前进程 (sleep/wait_queue)

如果有:

回收子进程资源 (free address space、内核栈)

将退出码写入父进程用户态地址空间中的 status

返回用户态

把子进程的 pid 作为返回值写入 trapframe

用户态 wait() 返回

- **do_exit** 用户态 exit(code)

陷入内核态

内核做: 关闭所有文件、回收内存空间

1. 保存退出码到 PCB
2. 将进程标记为 ZOMBIE, 等待父进程 wait 回收
3. 唤醒父进程 (如果父在 wait)

但是exit不会返回用户态, 直接进入调度器切换到其他进程。

- **综上, 内核态和用户态的交错执行本质上是通过: 用户态代码主动调用系统调用 → 内核代码执行 → 返回用户态继续执行。**
- **而且内核态是通过修改 trapframe 中用户寄存器的内容来把结果返回给程序**

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)

执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。（使用的是qemu-4.1.1）

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

答：

1. ucore 中用户程序的加载时机

在 ucore 实验系统中，用户程序（如 bin/sh、hello 等）并非在系统启动时动态从磁盘读取，而是：

在内核编译阶段，就被直接链接（link）进内核镜像（kernel image）中，作为只读数据段的一部分嵌入内存。

具体过程如下：

用户程序（ELF 格式）被预先编译为二进制文件（如 obj/user/hello）；

通过工具（如 objcopy）将其转换为 C 数组或符号（如 _binary_obj_user_hello_start）；

在 kern/init/init.c 的 user_main() 函数中，

直接以指针形式引用该二进制数据：C 编辑

```
1extern unsigned char _binary_obj_user_hello_start[];
2kernel_execve("hello", _binary_obj_user_hello_start, ...);
```

调用 kernel_execve() → 触发 do_execve() → load_icode() 将这段已在内核内存中的 ELF 数据解析并加载到新进程的用户地址空间。

综上所述：

用户程序在 内核启动前就已静态嵌入内核镜像，运行时直接从内核内存拷贝到用户空间，无需文件系统或磁盘 I/O。

2. 与通用操作系统（如 Linux）的区别

方面	ucore (教学 OS)	通用操作系统 (如 Linux)
加载来源	内核镜像中预嵌的二进制数组	文件系统中的可执行文件 (如 /bin/ls)

方面	ucore (教学 OS)	通用操作系统 (如 Linux)
是否依赖文件系统	否 (无真实文件系统)	是 (需 VFS + 磁盘驱动)
加载时机	内核初始化阶段 (user_main)	进程调用 execve() 时动态加载
灵活性	固定几个测试程序, 无法运行新程序	可执行任意符合格式的 ELF 文件
实现复杂度	极简 (直接 memcpy)	复杂 (需解析 ELF、按需分页、demand paging)

原因:

ucore 采用这种“**预加载嵌入式**”方式, 主要原因有:

1. 简化实验复杂度

避免实现完整的文件系统、块设备驱动、缓冲区管理等模块;

让学生聚焦于 **进程管理、虚拟内存、系统调用** 等核心 OS 概念。

2. 脱离硬件依赖

在 QEMU 模拟环境中, 无需模拟磁盘或 SD 卡;

所有代码和数据都在内存中, 启动更快、更可靠。

3. 教学目的明确

实验重点是理解 execve 如何建立用户上下文, 而非“如何从磁盘读文件”;

load_icode() 只需处理 **内存中的 ELF 解析**, 不涉及 I/O 调度。

lab5分支: gdb 调试系统调用以及返回

```
heye@DESKTOP-63A0U9G:~/qemu-4.1.1$ sudo gdb
[sudo] password for heye:
GNU gdb (Ubuntu 15.0.50.20240403-0
ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software F
oundation, Inc.
```

```
(gdb) set remotetimeout unlimited
(gdb) add-symbol-file obj/__usr_exit.out
add symbol table from file "obj/__usr_exit.out"
(y or n) y
obj/__usr_exit.out: No such file or directory.
(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...
```

add-symbol-file obj/__usr_exit.out 是 GDB (GNU 调试器) 中的一个命令, 用于在调试时加载额外的符号信息文件。其作用如下:

1. 主要用途

- 当你的程序（如操作系统或内核）动态加载了某个模块、用户程序或二进制文件（如 `__usr_exit.out`），而这些文件的符号（函数名、变量名等）没有包含在主调试信息中时，可以用 `add-symbol-file` 命令手动加载它们的符号表。
- 这样，GDB 就能识别和显示该文件中的函数、变量等符号，便于断点设置、单步调试和变量查看。

```
(gdb) break user/libs/syscall.c:18
Breakpoint 1 at 0x8000f0: file user/libs/syscall
.c, line 19.
(gdb) c
Continuing.
```

```
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]),
      :"memory");
    return ret;
}
```

汇编指令	作用
"ld a0, %1\n"	将参数 num（即系统调用号）加载到寄存器 a0
"ld a1, %2\n"	将 a[0] 加载到 a1
"ld a2, %3\n"	将 a[1] 加载到 a2
"ld a3, %4\n"	将 a[2] 加载到 a3
"ld a4, %5\n"	将 a[3] 加载到 a4
"ld a5, %6\n"	将 a[4] 加载到 a5
"ecall\n"	触发 系统调用异常 ，CPU 从 U-mode 切换到 S-mode，跳转到内核的 trap 处理函数（如 <code>trap_handler</code> ）

汇编指令 作用

"sd a0,
%0\n" 将内核返回值（在 a0 中）保存回变量 ret

它把系统调用号和最多5个参数分别装入 RISC-V 的 a0 到 a5 寄存器，然后执行 ecall 指令陷入内核；内核处理完后，将返回值通过 a0 传回，并存入变量 ret。

简单说：让用户程序安全地请求内核服务（如输出、退出等）。

```
Breakpoint 1, syscall (num=2)
  at user/libs/syscall.c:19
19          asm volatile (
(gdb) x/8i $pc
=> 0x8000f0 <syscall+28>:
    sd  a6,128(sp)
 0x8000f2 <syscall+30>:
    sd  a7,136(sp)
 0x8000f4 <syscall+32>:      sd      t1,32(sp)
 0x8000f6 <syscall+34>:      ld      a0,8(sp)
 0x8000f8 <syscall+36>:      ld      a1,40(sp)
 0x8000fa <syscall+38>:      ld      a2,48(sp)
 0x8000fc <syscall+40>:      ld      a3,56(sp)
 0x8000fe <syscall+42>:      ld      a4,64(sp)
(gdb) si
0x000000000008000f2      19          asm volatile (
(gdb)
```

```
0x80010c <syscall+50>:      addi   sp,sp,-144
 0x80010e <syscall+58>:      ret
 0x800110 <sys_exit>: mv    a1,a0
(gdb) si
0x00000000000800100     19          asm volatile (
(gdb) x/8i $pc
=> 0x800100 <syscall+44>:      ld      a5,72(sp)
 0x800102 <syscall+46>:      ecall
 0x800106 <syscall+50>:      sd      a0,28(sp)
 0x80010a <syscall+54>:      lw      a0,28(sp)
 0x80010c <syscall+56>:      addi   sp,sp,144
 0x80010e <syscall+58>:      ret
 0x800110 <sys_exit>: mv    a1,a0
 0x800112 <sys_exit+2>:      li      a0,1
(gdb) si
0x00000000000800102     19          asm volatile (
(gdb) i n n
```

```
(gdb) i r pc
pc          0x800102 0x800102 <syscall+46>
```

```
(gdb) b kern/trap/trapentry.S:133
Breakpoint 2 at 0xfffffffffc0200fb2: file kern/trap/trapentry.S, line 133.
(gdb) c
Continuing.
```

```
Breakpoint 2, __trapret ()
  at kern/trap/trapentry.S:133
133      sret
```

```
127      # sp should be the same as before "jal trap"
128
129      .globl __trapret
130  <__trapret:
131      RESTORE_ALL
132      # return from supervisor call
133  *     sret
134
135      .globl forkrets
136  <forkrets:
137      # set stack to this new process's trapframe
138      move sp, a0
139      j __trapret
140
141      .global kernel_execve_ret
142  <kernel_execve_ret:
143      // adjust sp to beneath kstacktop of current process
144      addi a1, a1, -36*REGBYTES
```

你想了解 RISC-V 架构下 ecall 和 sret 指令在 QEMU 中的处理流程。下面是简要分析和源码关键流程说明：

1. ecall 指令的处理

ecall 是用户/内核发起系统调用的指令，会触发陷入（trap），QEMU 需要模拟这种异常。

在 QEMU 的 riscv 架构实现中，ecall 相关的处理主要在 target/riscv insn_trans/trans_rvv.c、target/riscv/op_helper.c 和 target/riscv/cpu_helper.c 等文件。

关键流程如下：

1. QEMU 译码到 ecall 指令时，会调用 helper 函数（如 helper_ecall）。
2. helper_ecall 会调用 raise_exception 函数，向 QEMU 的 CPU 结构体注入一个 ECALL 异常（EXCP_ENVCALL）。
3. QEMU 的 CPU 执行主循环（cpu_exec）检测到异常后，会跳转到异常处理流程，模拟硬件 trap 过程，保存上下文，切换到 trap handler。
4. QEMU 会设置 sepc、scause、stval 等寄存器，并跳转到 stvec 指定的异常入口（如 trapentry.S 里的 __alltraps）。
5. **sret 指令的处理**

sret 是从 supervisor trap 返回的指令，恢复 trap 前的上下文。

QEMU 译码到 sret 时，会调用 helper_sret（target/riscv/op_helper.c）。

关键流程如下：

1. helper_sret 检查当前特权级和 sstatus 寄存器，恢复 sepc 到 PC，恢复 sstatus。
2. QEMU 会模拟硬件行为，把 PC 设置为 sepc，sstatus 的 SPP 位恢复到 SIE 等。
3. 继续执行 trap 前的指令流。
4. 关键源码文件与函数

- target/riscv/op_helper.c
 - helper_ecall：处理 ecall，调用 raise_exception。
 - helper_sret：处理 sret，恢复上下文。
 - target/riscv/cpu_helper.c
 - riscv_cpu_do_interrupt：处理异常分发，设置 sepc、scause、stval 等。
 - cpu_exec.c
 - cpu_exec：主循环，检测并分发异常。
4. 调试时常见流程
 - 当用户程序执行 ecall，QEMU 触发异常，进入 trap handler（如 trapentry.S）。
 - handler 处理完后，执行 sret，QEMU 恢复上下文，返回用户态。

总结

QEMU 对 ecall/sret 的处理高度模拟了 RISC-V 的硬件陷入与返回机制，核心流程是“指令译码 → helper 函数 → 异常分发 → trap handler → 上下文恢复 → 继续执行”。调试时可关注 op_helper.c 和 cpu_helper.c 这两个文件的相关 helper 函数实现。

TCG (Tiny Code Generator) 是 QEMU 的动态二进制翻译引擎。QEMU 运行时会将目标架构（如 RISC-V）的指令（包括 ecall、sret 等）翻译为宿主机可执行的中间代码（TCG IR），再进一步生成本地机器码执行。

TCG Translation 关键点

- QEMU 不是直接解释执行每条指令，而是先将目标指令块（如 ecall、sret）翻译为 TCG IR，再编译成本地代码，提高执行效率。
- 对于特殊指令（如 ecall、sret），QEMU 会在 TCG translation 阶段插入调用 helper 函数（如 helper_ecall、helper_sret），这些 helper 负责模拟异常、陷入、返回等硬件行为。
- 这样，QEMU 能高效且准确地模拟目标 CPU 的行为，包括异常处理、上下文切换等。

与双重 GDB 调试的关系

- 在双 GDB 调试实验（如同时调试 QEMU 和被模拟的内核/用户程序）中，QEMU 作为被调试对象，其 TCG translation 过程和 helper 函数的调用都可以被 GDB 跟踪。
- 你可以在 QEMU 的源码（如 helper_ecall）设置断点，观察 QEMU 如何响应 ecall/sret，并进一步跟踪 guest 代码的 trap handler。
- 这种调试方式让你既能看到 guest 代码的执行，也能深入 QEMU 的实现细节，理解指令翻译和异常分发的全过程。

总结

TCG translation 是 QEMU 高效模拟的核心，所有指令（包括 ecall/sret）都要经过这一翻译流程。双重 GDB 调试实验正是利用了 QEMU 的这一机制，既能调试 guest 代码，也能调试 QEMU 的指令翻译和异常处理逻辑，两者密切相关。