

OS lab5实验报告

实验准备

将之前实验的代码添加到 LAB5 源码中。

proc.c 的 alloc_proc 函数填充部分：

代码块

```
1 // LAB4原有
2 proc->state = PROC_UNINIT; // 初始化进程状态为未初始化
3 proc->pid = -1; // 初始化进程ID为-1（无效ID），后续通过
  get_id()来分配有效pid
4 proc->runs = 0; // 初始化运行次数为0
5 proc->kstack = 0; // 初始化内核栈为0（空指针），后续通过
  setup_kstack()为进程分配实际的内核栈空间
6 proc->need_resched = 0; // 初始化不需要重新调度
7 proc->parent = NULL; // 初始化父进程指针为NULL
8 proc->mm = NULL; // 初始化内存管理结构指针为NULL
9 memset(&(proc->context), 0, sizeof(struct context)); // 初始化上下文结构体，全部设
  为0，为后续保存现场做准备
10 proc->tfs = NULL; // 初始化陷阱帧为NULL
11 proc->pgdir = boot_pgdir_pa; // 初始化页目录表基址为boot_pgdir_pa
12 proc->flags = 0; // 初始化进程标志为0
13 memset(proc->name, 0, PROC_NAME_LEN + 1); // 初始化进程名称数组全部清零，后续通过
  set_proc_name()设置具体的进程名称
14 list_init(&(proc->list_link));
15 list_init(&(proc->hash_link));
16
17 // LAB5新增
18 proc->wait_state = 0; // 设置进程的等待状态为0，表示进程当前不在等待状态
19 proc->cptr = NULL; // 初始化子进程指针为NULL
20 proc->optr = NULL; // 初始化较年长兄弟进程(older sibling pointer)指针为
  NULL
21 proc->yptr = NULL; // 初始化较年轻兄弟进程(younger sibling pointer)指针为
  NULL
```

proc.c 的 do_fork 函数填充部分：（在LAB4基础上做了改动）

代码块

```
1 // 1) 分配进程控制块
```

```

2  if ((proc = alloc_proc()) == NULL) {
3      goto fork_out;
4  }
5
6  // LAB5: 设置父进程, 并确保父进程的 wait_state 为
7  proc->parent = current;
8  // 确保当前进程的等待状态为0 (不在等待子进程)
9  // assert(current->wait_state == 0); // 可选的断言检查
10
11 // 2) 分配子进程的内核栈
12 if (setup_kstack(proc) != 0) {
13     goto bad_fork_cleanup_proc;
14 }
15
16 // 3) 复制/共享内存管理信息
17 if (copy_mm(clone_flags, proc) != 0) {
18     goto bad_fork_cleanup_kstack;
19 }
20
21 // 4) 复制上下文 (含 trapframe), 并设置返回路径
22 copy_thread(proc, stack, tf);
23
24 // 5) 分配唯一 pid
25 bool intr_flag;
26 local_intr_save(intr_flag); // 关中断, 保证原子性
27 {
28     proc->pid = get_pid();
29
30     // 挂到哈希表, 便于 O(1) 按 pid 查找
31     hash_proc(proc);
32
33     // LAB5: 使用 set_links 建立进程关系并插入链表
34     set_links(proc);
35 }
36 local_intr_restore(intr_flag); // 恢复中断
37
38 // 6) 唤醒子进程, 使其可调度 (会把 state 置为 PROC_RUNNABLE)
39 wakeup_proc(proc);
40
41 // 7) 父进程得到子进程的 pid 作为返回值
42 ret = proc->pid;

```

`proc.c` 的 `proc_run` 函数:

代码块

```

1  void proc_run(struct proc_struct *proc)
2  {
3      // 只有当要运行的进程不是当前进程时, 才需要执行切换
4      if (proc != current)
5      {
6          /* 关闭本地中断, 并保存当前中断标志状态 */
7          unsigned long irq_flags;
8          local_intr_save(irq_flags);
9
10         /* 保存原来的当前进程指针, 将 current 切换为目标进程 */
11         struct proc_struct *prev = current;
12         current = proc;
13
14         /* 记录该进程被调度运行的次数 (用于统计或调度算法) */
15         current->runs++;
16
17         /* 切换页表, 加载新进程的地址空间 */
18         lsatp(current->pgdir);
19
20         /*
21          * 进行上下文切换:
22          * 保存 prev 的寄存器上下文, 恢复 current 的寄存器上下文。
23          * 当 CPU 再次切换回 prev 时, switch_to 函数会从这里继续执行。
24          */
25         switch_to(&prev->context, &current->context);
26
27         /* 恢复之前保存的中断标志状态 */
28         local_intr_restore(irq_flags);
29     }
30 }

```

trap.c 的填充部分: 在 LAB3 的基础上做了改动, 删去了 sbi_shutdown(), 同时设置了调度标志, 保证程序能正常执行直至结束。

代码块

```

1  case IRQ_S_TIMER:{
2      /* LAB5 GRADE    YOUR CODE :  */
3      /* 时间片轮转:
4      *(1) 设置下一次时钟中断 (clock_set_next_event)
5      *(2) ticks 计数器自增
6      *(3) 每 TICK_NUM 次中断 (如 100 次), 进行判断当前是否有进程正在运行, 如果有则
    标记该进程需要被重新调度 (current->need_resched)
7      */
8
9      // (1) 设置下一次时钟中断

```

```

10         clock_set_next_event();
11
12         // (2) ticks 计数器自增
13         ticks++;
14
15         // (3) 每 TICK_NUM 次中断, 标记进程需要重新调度
16         if (ticks % TICK_NUM == 0) {
17             if (current != NULL) {
18                 current->need_resched = 1;
19             }
20         }
21         break;
22     }

```

另外, 在 `trap.c` 中添加处理页面错误的代码: (以通过两个 `fault` 样例)

代码块

```

1  static int
2  pgfault_handler(struct trapframe *tf) {
3      uintptr_t addr = tf->tval;
4
5      if (current == NULL) {
6          print_trapframe(tf);
7          panic("page fault in kernel!");
8      }
9
10     if (current->mm == NULL) {
11         print_trapframe(tf);
12         panic("page fault in kernel thread!");
13     }
14
15     // 对于非法访问 (如 faultread 访问地址 0), 返回错误
16     // 让 exception_handler 杀死进程
17     return -E_INVAL;
18 }
19
20 void exception_handler(struct trapframe *tf)
21 {
22     int ret;
23     switch (tf->cause)
24     {
25         // ... 其他 case 保持不变 ...
26
27         case CAUSE_FETCH_PAGE_FAULT:
28             if ((ret = pgfault_handler(tf)) != 0) {

```

```

29         cprintf("Instruction page fault\n");
30         print_trapframe(tf);
31         if (current != NULL) {
32             do_exit(-E_KILLED);
33         } else {
34             panic("kernel page fault");
35         }
36     }
37     break;
38
39     case CAUSE_LOAD_PAGE_FAULT:
40         if ((ret = pgfault_handler(tf)) != 0) {
41             cprintf("Load page fault\n");
42             print_trapframe(tf);
43             if (current != NULL) {
44                 do_exit(-E_KILLED);
45             } else {
46                 panic("kernel page fault");
47             }
48         }
49         break;
50
51     case CAUSE_STORE_PAGE_FAULT:
52         if ((ret = pgfault_handler(tf)) != 0) {
53             cprintf("Store/AMO page fault\n");
54             print_trapframe(tf);
55             if (current != NULL) {
56                 do_exit(-E_KILLED);
57             } else {
58                 panic("kernel page fault");
59             }
60         }
61         break;
62
63     // ... 其他 case 保持不变 ...
64 }
65 }

```

练习一

代码如下：

代码块

```

1      /* --- begin inserted code --- */
2      /*

```

```

3  * (6) 建立用户态 trapframe 和初始寄存器
4  *
5  * 目标:
6  *   - sp 指向用户栈顶 (USTACKTOP)
7  *   - epc 指向 ELF 的入口 (elf->e_entry)
8  *   - status: 清 SPP (返回后进入用户态), 置 SPIE (sret 返回后允许中断)
9  *
10 * /
11 /* set user stack pointer */
12 tf->gpr.sp = (uintptr_t)USTACKTOP;
13
14 /* set program entry point (sepc) */
15 tf->epc = (uintptr_t)elf->e_entry;
16
17 /* set return value for exec in user mode (argc = 0) */
18 tf->gpr.a0 = 0;
19
20 /* Adjust sstatus: clear SPP (so sret goes to user-mode), set SPIE (enable
   interrupts after sret) */
21 tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;

```

设计实现过程

1. `load_icode` 首先为当前进程创建一个新的 `mm_struct`，并调用 `pgdir_alloc_page/vmm_map/phdr` 等逻辑，按 ELF 的 program header 把代码段、数据段映射到进程的页表中。同时分配用户栈（一般大小为若干页），栈顶地址固定为 `USTACKTOP`。
2. 遍历 ELF 的 program segments（`PT_LOAD`），把数据按 `p_vaddr` 复制到用户页，同时清空 bss 区域。这保证了用户程序进程的虚拟空间布局与 ELF 所要求的一致。
3. 正确初始化 trapframe
 - 设置用户栈指针 SP `tf->gpr.sp = USTACKTOP;`
 - 设置 EPC = ELF 入口地址 `tf->epc = elf->e_entry;`
这样用户进程第一次执行的就是 ELF `_start`。
 - 设置 `trapframe.status`
`tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;`
 - `SPP = 0` ⇒ `sret` 之后进入用户态
 - `SPIE = 1` ⇒ 允许用户态中断在返回后开启
4. 在 `load_icode` 末尾，执行以下代码保证 MMU 使用新页表运行：

代码块

```
1 write_csr(satp, MAKE_SATP(mm->pgdir));
2 sfence.vma;
```

执行经过

当一个用户进程被创建完成后，ucore 的调度器会把其标记为 `RUNNABLE`。当调度器选择它运行时，该进程进入 `RUNNING` 状态，随后经历以下步骤，最终执行用户程序的第一条指令：

1. 调度器选择并切换到该进程（switch_to）

调度器调用 `proc_run(next)`，该函数执行：

- 切换当前 `current` 指针
- 切换页表（`satp`）
- 调用 `switch_to` 切换内核栈、保存/恢复上下文

2. 内核准备返回用户态执行（在系统调用/时钟中断结束处）

返回前内核会把该进程的 trapframe restore 到 CPU：

- `sepc ← tf->epc`（用户程序入口）
- `sscratch ← 内核栈地址`
- `sstatus` 根据 `tf->status` 设定（`SPP=0` 表示要进入用户态）

3. 执行 sret，CPU 权限从 S 模式切换到 U 模式

`sret` 指令做三件事：

- a. 把 `SPP=0` 表示的目标模式切换到用户态
- b. 把 `sepc` 的值加载到 PC（执行开始地址）
- c. 根据 SPIE 设置用户态的中断使能

此时 CPU 正式离开内核态。

4. 用户程序开始执行自己的第一条指令

`PC = elf->e_entry`

然后用户程序在它的虚拟地址空间中执行：

- 使用自己的用户栈（`USTACKTOP`）
- 所有地址访问都受页表保护
- 若进程执行系统调用/异常，则再次进入 trap 流程

这标志着 ucore 成功完成了一次从“创建用户进程 → 调度 → 执行用户代码”的完整过程。

练习二

```

1  // (1) 获取源页面的内核虚拟地址
2  void *src_kvaddr = page2kva(page);
3
4  // (2) 获取目标页面的内核虚拟地址
5  void *dst_kvaddr = page2kva(npage);
6
7  // (3) 复制页面内容（整页复制，大小为 PGSIZE）
8  memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
9
10 // (4) 建立目标进程的虚拟地址 start 到新物理页 npage 的映射
11 ret = page_insert(to, npage, start, perm);

```

`copy_range` 实现采用“逐页复制”的方式，将父进程用户空间中的有效页面复制到子进程中。实现过程中，函数首先对传入的地址范围进行合法性检查，然后使用 `get_pte` 按页扫描父进程页表，跳过未映射区域以提升效率。对于每一个有效页，函数通过 `pte2page` 获取其对应的物理页，并调用 `alloc_page` 为子进程分配新的物理页。利用 `page2kva` 将物理页转换为可访问的内核虚拟地址后，使用 `memcpy` 完整复制父页内容。在复制完成后，使用 `page_insert` 将新页正确映射到子进程页表中，继承父页的用户态权限。

核心流程如下：

1. 参数合法性检查

代码块

```

1  assert(start % PGSIZE == 0 && end % PGSIZE == 0);
2  assert(USER_ACCESS(start, end));

```

1. `start` 与 `end` 必须按页对齐，否则无法按页复制。
2. 复制范围必须在用户空间，否则可能破坏内核地址空间。

2. 逐页扫描父页表项

代码块

```

1  pte_t *ptep = get_pte(from, start, 0);
2  if (ptep == NULL) {
3      start = ROUNDDOWN(start + PTSIZE, PTSIZE);
4      continue;
5  }

```

- 使用 `get_pte(from, start, 0)` 查询父页表项，不创建页表。

- 若 pte 不存在，对应的一级页表中不存在该段，则可直接跳到下一个 PTSIZE（4MB）区域，提升效率。

3.判断该页是否有效（PTE_V）

代码块

```
1  if (*ptep & PTE_V) {
```

- 仅当父进程该页有效（被实际使用）时才需要复制，否则跳过。

4.为子进程确保页表存在

代码块

```
1  nptep = get_pte(to, start, 1);
2  if (nptep == NULL) {
3      return -E_NO_MEM;
4  }
```

- 子进程对应的二级页表必须存在，否则需要创建。
- 若创建失败则返回内存不足。

5.获取父页的权限与对应 Page 结构

代码块

```
1  uint32_t perm = (*ptep & PTE_USER);
2  struct Page *page = pte2page(*ptep);
3  assert(page != NULL);
```

- `perm` 继承用户态权限位（读写、用户可访问等）。
- `pte2page` 得到物理页对应 Page 结构，以便获取内核映射地址。

6.为子进程分配新物理页

代码块

```
1  struct Page *npage = alloc_page();
2  assert(npage != NULL);
```

- 在非 COW 的实现中必须为子进程分配新物理页。

- 若分配失败直接断言。

7.计算源页与目标页的内核虚拟地址

代码块

```
1 void *src_kvaddr = page2kva(page);
2 void *dst_kvaddr = page2kva(npage);
```

8.拷贝整页数据

代码块

```
1 bool intr_flag;
2 local_intr_save(intr_flag);
3
4 memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
5
6 local_intr_restore(intr_flag);
```

1. `memcpy` 需拷贝 4KB 整页内容。
2. 使用 `local_intr_save()` 与 `restore()` 保护临界区，避免中断导致页内容不一致。
3. 保证复制过程原子性。

9.将新物理页插入子页表中

代码块

```
1 int ret = page_insert(to, npage, start, perm);
2 assert(ret == 0);
```

- `page_insert` 完成从虚拟地址到物理页的映射。
- 若插入失败（如已有映射），需断言终止。
- 权限继承父进程。

10.移动到下一页

代码块

```
1 start += PGSIZE;
```

- 每次循环复制一页，递增地址继续扫描。

练习三

fork 的执行流程

用户态调用：`int pid = fork();`，随后流程为：

(1) 用户态 → trap（系统调用）→ 内核态

- 执行 `ecall`
- trap 进入内核的 `exception_handler`
- 内核解析 `syscall number = SYS_fork`

(2) 内核态执行 `do_fork`

主要步骤：

- 创建子进程的 `proc_struct`
- 复制父进程的内核栈、`trapframe`、线程上下文
- 复制父进程页表并建立 CoW 或直接复制内存（不同实验版本差异）
- 将子进程的 `trapframe->a0` 设置为 0
→ 让子进程的 `fork` 返回值变为 0
- 修改父进程的返回值为 子 `pid`
- 把子进程放入可调度队列

(3) 内核返回用户态（通过 `sret`）

返回父进程 → `fork` 返回子 `pid`

子进程第一次被调度时 → `fork` 返回 0

exec 的执行流程

用户态调用：`exec("hello");`，流程为：

(1) 用户态 `ecall` → 进入内核

`syscall number = SYS_exec`

(2) 内核态执行 `do_execve / load_icode`

关键操作：

- 丢弃当前进程旧的地址空间 mm
- 为进程创建新的 mm_struct
- 根据 ELF program headers:
 - 建立代码段/数据段页表映射
 - 加载文件内容到用户虚拟地址
- 分配新的用户栈
- **重建 trapframe** (设置 sp、epc=ELF entry、清 SPP)
- 切换页表 satp

(3) sret 返回用户态

返回到新 ELF 程序的入口 (_start)

wait 的执行流程

用户态调用: `wait(&status);`

(1) 用户态 → ecall → 内核态

syscall = SYS_wait

(2) 内存态执行 do_wait

逻辑:

- 在进程表中寻找某个子进程, 其 state == ZOMBIE
- 若找到:
 - 回收它的内存、页表、内核栈
 - 获取 exit code 并写入用户提供的 status
 - 返回子进程 pid
- 若没找到:
 - 将当前进程置为 SLEEPING
 - 让调度器切换到其他进程

(3) 内核返回用户态

- 父进程再次被唤醒, wait 返回子 pid

exit 的执行流程

用户态调用: `exit(0);`

(1) 用户态 → ecall → 内核态

syscall = SYS_exit

(2) 内核态执行 do_exit

操作：

- 设置当前进程状态为 **ZOMBIE**
- 记录退出码 exit code
- 释放用户地址空间（mm）
- 唤醒父进程（可能在 wait）
- 调度器切换到其它进程（进程不再返回用户态）

(3) 该进程不会再返回用户态

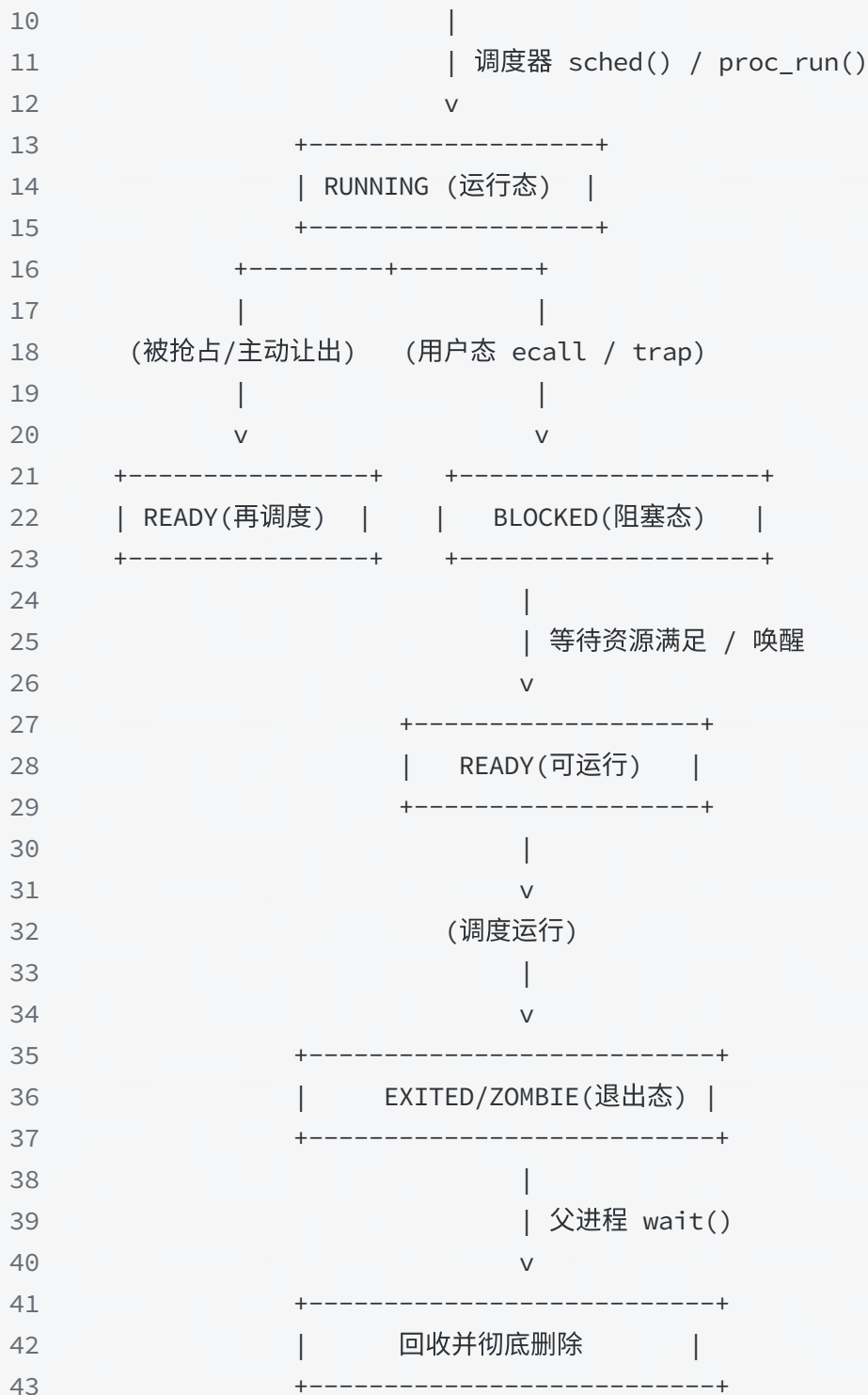
直到父进程 wait 回收它。

用户态和内核态对比

系统调用	用户态部分	内核态部分
fork	fork() 调用 + 返回值处理	创建子进程、复制页表、复制 trapframe、子进程 a0=0
exec	exec("file") 调用	创建新地址空间、加载 ELF、设置 trapframe、切页表
wait	用户传递 status 地址	查找 zombie、回收资源、睡眠父进程、唤醒父进程
exit	exit(code)	设置僵尸状态、唤醒父进程、释放地址空间

ucore 用户进程的生命周期





Challenge1: COW机制

我们在练习二代码的基础上实现COW机制：

代码块

```

1  int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
    share)
2  // (1) 参数说明
3  // to: 子进程的页目录指针
4  // from: 父进程的页目录指针

```

```

5 // start/end: 要复制的虚拟地址范围
6 // share: 是否共享 (在COW中始终为true)
7 {
8     // (2) 地址检查: 确保地址是页对齐的, 且位于用户地址空间
9     assert(start % PGSIZE == 0 && end % PGSIZE == 0);
10    assert(USER_ACCESS(start, end));
11
12    // (3) 核心循环逻辑: 逐页处理父进程地址空间中的每个页面
13    do {
14
15        // (3_1) 获取父进程中该虚拟地址的页表项, 如果父进程没有映射此页, 就跳过整个页表
16        pte_t *ptep = get_pte(from, start, 0), *nptep; //ptep和nptep都是指向页表
项的指针
17        if (ptep == NULL) {
18            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
19            continue;
20        }
21
22        // (3_2) 处理有效页面
23        if (*ptep & PTE_V) { //检查父进程页面是否有效
24            if ((nptep = get_pte(to, start, 1)) == NULL) { //为子进程创建页表项
(参数1表示如不存在则创建)
25                return -E_NO_MEM; //error_没内存
26            }
27
28            struct Page *page = pte2page(*ptep); //从页表项指针获取对应的物理页管理
结构
29
30            if (*ptep & PTE_W) {
31                // ===== 关键修改: 只对可写页面设置COW =====
32                page_ref_inc(page); //增加物理页面的引用计数
33
34                // 计算新的PTE值(去掉W,加上COW)
35                pte_t pte_flags = (*ptep) & 0x3FF; // 保留父进程页表项的低10位
(标志位)
36                pte_t new_pte = ((*ptep) & ~((pte_t)0x3FF)) | // 保留高位的物理
页号 (PPN)
37                    ((pte_flags & ~PTE_W) | PTE_COW);
38                // 修改标志位: ~PTE_W表示清楚写权限, 这是触发COW的关
键, PTE_COW是自定义的COW标志位
39
40                // ===== 重要: 不修改父进程,只修改子进程 =====
41                *nptep = new_pte; //只设置子进程的页表项, 父进程的页表项保持不变, 仍
保持写权限
42                // 这样父进程可以继续写入, 而子进程的写操作会报错
43
44                // 只刷新子进程的TLB

```

```

45         tlb_invalidate(to, start);
46
47     } else {
48         // 只读页面(TEXT段)直接共享
49         page_ref_inc(page);
50         *nptep = *ptep; // 子进程的页表项指针就等同于父进程的
51         // 这里不用设置COW, 因为这些页面本来就不能写
52     }
53 }
54 start += PGSIZE; //页对齐的地址递增
55 } while (start != 0 && start < end);
56
57 return 0;
58 }

```

总体概述

`copy_range()` 函数负责在进程 `fork()` 时复制父进程的地址空间到子进程。基于练习二代码的关键改进是：对于可写页面，不是立即复制物理页面，而是**先共享物理页面并标记为COW，等到实际写入时才触发页错误进行真正的复制**。

COW的工作流程总结

1. Fork时：

- 可写页面：父子共享，子进程页表项去掉写权限+添加COW标志
- 只读页面：父子共享，完全复制页表项

2. 子进程尝试写入时：

- CPU检测到无写权限 → 触发缺页异常 `page fault`
- 页错误处理程序看到COW标志 → 执行真正的复制：
 - i. 分配新物理页面
 - ii. 复制原页面内容
 - iii. 更新子进程页表项：设置写权限，清除COW标志
 - iv. 原页面引用计数减1

3. 父进程写入时：

- 正常写入（父进程页表项仍有写权限）
- 如果子进程还未写入：父子继续共享，子进程的COW状态不变
- 如果子进程已写入：父子使用不同的物理页面，互不影响

其他函数的修改

代码块

```
1 // 页面错误总处理函数
2 static int pgfault_handler(struct trapframe *tf) {
3     uintptr_t addr = tf->tval; // 发生错误的虚拟地址
4     uint32_t error_code = tf->cause; // 错误原因
5
6     if (current == NULL) { //表示内核线程缺页（不可能啊！）所以直接panic
7         print_trapframe(tf);
8         panic("page fault in kernel!"); // current是当前运行进程的PCB指针
9     }
10
11     if (current->mm == NULL) { //表示内核线程缺页（不可能啊！）所以直接panic
12         print_trapframe(tf);
13         panic("page fault in kernel thread!"); // current->mm是进程的内存管理结构
14     }
15
16     // 检查是否是写时复制触发的页面错误
17     pde_t *pgdir_va = current->mm->pgdir; // 进程页目录
18     pte_t *ptep = get_pte(pgdir_va, addr, 0); // 获取页表项
19
20     if (ptep != NULL && (*ptep & PTE_V)) { // 页表项存在且有效
21
22         // 检查COW标志 且 是写操作引起的缺页错误
23         if ((*ptep & PTE_COW) && (error_code == CAUSE_STORE_PAGE_FAULT)) {
24             int ret = do_cow_page_fault(current->mm, error_code, addr);
25             return ret;
26         }
27     }
28
29     cprintf("page fault at 0x%08x: %c/%c\n", addr,
30            (error_code == CAUSE_LOAD_PAGE_FAULT) ? 'R' : 'W',
31            (tf->status & SSTATUS_SPP) ? 'K' : 'U');
32
33     return -E_INVAL; // 不是COW错误，返回无效参数错误
34 }
35
36 // COW页面错误处理函数——执行真正的COW复制操作
37 static int do_cow_page_fault(struct mm_struct *mm, uint32_t error_code,
38                             uintptr_t addr) {
39     int ret = 0;
40     pte_t *ptep = NULL;
41     uintptr_t la = ROUNDDOWN(addr, PGSIZE); // la = 页面对齐的地址
```

```

42     ptep = get_pte(mm->pgdir, la, 0);           // ptep是指向虚拟地址 la 对应的页表
项的指针
43     if (ptep == NULL || !(*ptep & PTE_V)) {
44         return -E_INVALID;
45     }
46
47     pte_t old_pte_value = *ptep;                // 保存原页表项
48
49     struct Page *old_page = pte2page(*ptep);    // 从页表项的值 → 对应物理页的Page结
构指针
50     int ref_count = page_ref(old_page);         // 获取引用计数
51
52     // (1) 只有一个进程引用,直接恢复写权限 (其他进程已经执行了COW, 只剩当前进程使用)
53     if (ref_count == 1) {
54         pte_t new_pte = (old_pte_value | PTE_W) & ~PTE_COW;
55         *ptep = new_pte;
56
57         // 强制刷新TLB
58         asm volatile("sfence.vma zero, %0" :: "r"(la) : "memory");
59         asm volatile("fence" ::: "memory");
60
61         return 0;
62     }
63
64     // (2) 多个进程共享,需要真正复制页面
65     struct Page *new_page = alloc_page();       // 分配新物理页
66     if (new_page == NULL) {
67         return -E_NO_MEM;
68     }
69     uintptr_t src_kva = (uintptr_t)page2kva(old_page); // 复制页面内容
70     uintptr_t dst_kva = (uintptr_t)page2kva(new_page);
71     memcpy((void*)dst_kva, (void*)src_kva, PGSIZE); // 复制4KB数据
72     page_ref_dec(old_page); //原页面引用减一
73
74     // (3) 更新页表项
75     uint32_t perm = (old_pte_value & (PTE_U | PTE_R | PTE_X)) | PTE_W; // 计算新
权限
76     set_page_ref(new_page, 1); // 修改PTE
77     *ptep = pte_create(page2ppn(new_page), PTE_V | perm);
78
79     // 强制刷新TLB, 确保操作顺序
80     asm volatile("sfence.vma zero, %0" :: "r"(la) : "memory");
81     asm volatile("fence" ::: "memory");
82
83     return 0;
84 }

```

这两个函数共同实现了COW机制：

1. `pgfault_handler`：识别器

- 快速判断是否COW错误
- 不是COW错误就交给标准处理
- 是COW错误就交给 `do_cow_page_fault`。

2. `do_cow_page_fault`：执行器

- 根据引用计数选择策略
- 管理物理页复制和引用计数
- 更新页表映射

测试样例说明

为测试所编写的COW机制，我们引入了两个新的测试样例：`cowtest.c` 和 `dirtycow_test.c`。

`cowtest.c`

代码块

```
1 // user/cowtest.c
2 #include <stdio.h>
3 #include <ulib.h>
4
5 #define TEST_SIZE 10
6
7 int main(void) {
8     cprintf("COW Test Starting...\n");
9
10    static int data[TEST_SIZE]; // 测试数据结构, 10个int = 40字节, 通常一个页面
    // 4KB, data数组只占很小部分
11
12    for (int i = 0; i < TEST_SIZE; i++) { // 父进程初始化数组
13        data[i] = i * 100; // [0, 100, 200, ..., 900]
14    }
15    cprintf("Parent: initialized data\n");
16    // 此时:
17    // 物理页A: [0, 100, 200, 300, 400, 500, 600, 700, 800, 900, ...]
18    // 父进程PTE: 指向物理页A, 有写权限(W=1)
19
20    int pid = fork(); // COW机制在这里生效, fork()函数是系统调用, 创建子进程
21
22    // 物理页A: [0, 100, 200, ..., 900] ← 被两个进程共享
23    //           ↗           ↖
24    // 父进程PTE(W=1)       子进程PTE(W=0, COW=1)
```

```

25 // 引用计数: page.ref = 2
26
27 if (pid == 0) { //OS内核硬性保证父进程 pid > 0, 子进程 pid == 0, 如果出错则 pid
    < 0
28 // 子进程: 只读操作, 不触发COW
29 cprintf("Child: reading parent's data...\n");
30 int sum = 0;
31 for (int i = 0; i < TEST_SIZE; i++) {
32     sum += data[i]; // 只是读取, 页表项W=0但R=1, 允许读取
33 }
34 // sum = 0+100+200+...+900 = 4500 ✓
35 cprintf("Child: sum before write = %d\n", sum);
36
37 if (sum != 4500) {
38     cprintf("Child: ERROR - sum should be 4500\n");
39     exit(-1);
40 }
41
42 // 子进程写入操作 - 触发COW
43 cprintf("Child: writing data (trigger COW)...\n");
44 for (int i = 0; i < TEST_SIZE; i++) {
45     data[i] = i * 200;
46 }
47 /*
48 1. 子进程执行: data[0] = 0 * 200 = 0
49 2. CPU检测: 页表项PTE_W=0, 无写权限
50 3. 触发页面错误 (Page Fault)
51 4. 缺页处理程序检查: PTE_COW=1 → 是COW页面
52 5. 执行COW处理:
53     a. 分配新物理页B
54     b. 复制A的内容到B
55     c. 更新子进程PTE指向B, 设置W=1, 清除COW
56     d. 原页面A引用计数减为1
57 6. 恢复执行, 写入成功
58 */
59 /*
60 写入后的内存状态:
61 物理页A: [0, 100, 200, ..., 900] ← 仅父进程使用, ref=1
62 物理页B: [0, 200, 400, ..., 1800] ← 子进程独享, ref=1
63 */
64
65 // 再次读取, 求和验证子进程的写入结果
66 sum = 0;
67 for (int i = 0; i < TEST_SIZE; i++) {
68     sum += data[i];
69 }
70 // sum = 0+200+400+...+1800 = 9000 ✓

```

```

71     cprintf("Child: sum after write = %d\n", sum);
72
73     if (sum != 9000) {
74         cprintf("Child: ERROR - sum should be 9000\n");
75         exit(-2);
76     }
77
78     exit(0);
79 } else if (pid > 0) {
80     int exit_code = 0;
81     waitpid(pid, &exit_code); // 父进程等待子进程结束
82
83     if (exit_code == 0) {
84         cprintf("Child completed successfully\n");
85     } else {
86         cprintf("Child failed with code %d\n", exit_code);
87     }
88
89     cprintf("Parent: checking data after child...\n");
90     // 验证父进程的数据未受影响
91     int sum = 0;
92     for (int i = 0; i < TEST_SIZE; i++) {
93         sum += data[i]; // 读取的是物理页A, 原始数据
94     }
95     // sum = 0+100+200+...+900 = 4500 ✓
96     cprintf("Parent: sum = %d (should be 4500)\n", sum);
97
98     if (sum == 4500 && exit_code == 0) {
99         cprintf("COW Test PASSED!\n");
100     } else {
101         cprintf("COW Test FAILED!\n");
102     }
103 } else {
104     cprintf("fork failed\n");
105     return -1;
106 }
107
108 return 0;
109 }

```

`fork()` 后分裂，父进程和子进程交替输出，流程如下：

代码块

- 1 时间轴：
- 2 t0: 【父进程】输出 "COW Test Starting..."

```
3  t1: 【父进程】输出 "Parent: initialized data"
4  t2: fork()分裂成两个进程
5      └─ 【父进程】继续，但暂时不输出（在waitpid等待）
6      └─ 【子进程】开始执行，输出所有"Child: ..."内容
7  t3: 【子进程】exit(0)结束
8  t4: 【父进程】waitpid返回，继续输出"Child completed successfully..."
9  t5: 【父进程】输出剩余内容
```

测试后的输出如下（由于篇幅问题，这里只显示关键输出）：可以看到输出了正确语句。

代码块

```
1  kernel_execve: pid = 2, name = "cowtest".
2  Breakpoint
3  COW Test Starting...
4  Parent: initialized data
5  Child: reading parent's data...
6  Child: sum before write = 4500
7  Child: writing data (trigger COW)...
8  Child: sum after write = 9000
9  Child completed successfully
10 Parent: checking data after child...
11 Parent: sum = 4500 (should be 4500)
12 COW Test PASSED!
13 all user-mode processes have quit.
14 init check memory pass.
```

dirtycow_test.c

什么是 Dirty COW？

Dirty COW 是Linux内核中的一个著名安全漏洞，该漏洞不是正常的COW功能，而是一个**利用COW机制的竞态条件漏洞**。

漏洞本质：

- 正常COW：写时复制，保护进程间内存隔离
- Dirty COW漏洞：利用内核中的竞态条件，让只读内存页面被恶意修改

Dirty COW攻击步骤：

1. 进程A：通过 `mmap` 映射一个只读文件到内存
2. 进程A： `fork()` 创建进程B（共享该只读页面）
3. 【竞态条件开始】

进程A：不断调用 `madvise(MADV_DONTNEED)` → 告诉内核"这个页面不重要"

进程B：不断写入该页面 → 触发COW处理

（这里值得注意：因为 `mmap` 时用了 `MAP_PRIVATE`，即使原文件是只读的，对私有映射的写入应该触发COW，`fork` 之后进程B的那个页面自动带有COW标志，所以可写！）

4. 关键：如果两个操作恰好同时发生：

- `madvise` 让内核认为页面可以被丢弃
- COW处理正在分配新页面
- 内核可能错误地将写入应用到原只读页面！

5. 结果：只读页面被非法修改！普通用户提升到root权限

代码块

```
1 // user/dirtycow_test.c
2 #include <stdio.h>
3 #include <ulib.h>
4 #include <string.h>
5
6 int main(void) {
7     cprintf("DirtyCOW vulnerability test\n");
8
9     // 使用静态数组代替动态分配，确保在一个页面内
10    static char data[4096];
11
12    // 初始化数据
13    const char *original = "ORIGINAL DATA";
14    for (int i = 0; original[i] != '\0'; i++) {
15        data[i] = original[i];
16    }
17    data[13] = '\0'; // 确保字符串结束
18    // 现在: data[] = "ORIGINAL DATA"
19    // 物理页面P: 包含这个字符串，父进程有写权限
20
21    // fork创建子进程
22    int pid = fork();
23    // fork后: 父子共享页面P
24    // 父进程PTE: W=1
25    // 子进程PTE: W=0, COW=1
26
27    if (pid == 0) {
28        // 子进程尝试多次写入，触发竞态条件
29        for (int i = 0; i < 100; i++) {
30            data[0] = 'M'; // 触发COW
31        }
32    }
33    /*
34     潜在竞态场景时间线:
```

```

34  t0: 子进程第一次写入 → 触发COW
35      CPU: 检测W=0 → 触发缺页 → 进入内核处理
36  t1: 内核开始COW处理:
37      1. 获取原页面锁
38      2. 分配新页面Q
39      3. 复制P→Q
40      4. 更新子进程PTE指向Q
41      5. 释放原页面锁
42  t2: 子进程第二次写入 (循环中)
43      如果发生在t1的步骤4之前:
44      - 可能还在使用旧的映射 (指向P)
45      - 但内核认为正在处理COW
46      - 可能导致未定义行为
47  */
48
49      // 检查是否修改成功
50      if (data[0] == 'M') {
51          cprintf("Child: data modified (expected behavior)\n");
52      }
53
54      exit(0);
55  } else if (pid > 0) {
56      int exit_code = 0;
57      waitpid(pid, &exit_code); // 等待子进程结束
58
59      // 父进程检查数据是否被破坏
60      int is_original = 1;
61      for (int i = 0; original[i] != '\0'; i++) {
62          if (data[i] != original[i]) {
63              is_original = 0; // 数据被破坏
64              break;
65          }
66      }
67
68      if (is_original) {
69          cprintf("Test completed - no corruption should occur\n");
70      } else {
71          cprintf("ERROR: parent data corrupted!\n");
72      }
73  } else {
74      cprintf("fork failed\n");
75      return -1;
76  }
77
78  return 0;
79  }

```


我们的上述代码不是Dirty COW攻击，只是测试COW机制的竞态条件安全性。该测试的测试输出如下（只显示关键部分）：

代码块

```
1  kernel_execve: pid = 2, name = "dirtycow_test".
2  Breakpoint
3  DirtyCOW vulnerability test
4  Child: data modified (expected behavior)
5  Test completed - no corruption should occur
6  all user-mode processes have quit.
7  init check memory pass.
```

可以看到，我们的COW实现正确处理了多次写入，**没有竞态条件问题**。真实的COW攻击还需要引入另一进程，让其不断调用 `madvise(addr, MADV_DONTNEED)`，与触发COW的进程并发执行，增大竞态窗口。

测试得分截图

基础版：

```
badsegment:          (2.0s)
  -check result:      OK
  -check output:      OK
divzero:             (1.7s)
  -check result:      OK
  -check output:      OK
softint:             (1.6s)
  -check result:      OK
  -check output:      OK
faultread:           (1.8s)
  -check result:      OK
  -check output:      OK
faultreadkernel:     (2.0s)
  -check result:      OK
  -check output:      OK
hello:               (1.8s)
  -check result:      OK
  -check output:      OK
testbss:             (1.8s)
  -check result:      OK
  -check output:      OK
pgdir:               (1.6s)
  -check result:      OK
  -check output:      OK
yield:               (2.0s)
  -check result:      OK
  -check output:      OK
badarg:              (1.7s)
  -check result:      OK
  -check output:      OK
exit:                (1.7s)
  -check result:      OK
  -check output:      OK
spin:                (4.5s)
  -check result:      OK
  -check output:      OK
forktest:            (1.7s)
  -check result:      OK
  -check output:      OK
Total Score: 130/130
```

COW机制版:

```

badsegment:                (2.0s)
  -check result:            OK
  -check output:            OK
divzero:                    (1.1s)
  -check result:            OK
  -check output:            OK
softint:                    (1.5s)
  -check result:            OK
  -check output:            OK
faultread:                  (2.5s)
  -check result:            OK
  -check output:            OK
faultreadkernel:            (2.2s)
  -check result:            OK
  -check output:            OK
hello:                      (1.9s)
  -check result:            OK
  -check output:            OK
testbss:                    (2.6s)
  -check result:            OK
  -check output:            OK
pgdir:                      (1.6s)
  -check result:            OK
  -check output:            OK
yield:                      (2.1s)
  -check result:            OK
  -check output:            OK
badarg:                     (1.5s)
  -check result:            OK
  -check output:            OK
exit:                      (1.9s)
  -check result:            OK
  -check output:            OK
spin:                      (5.0s)
  -check result:            OK
  -check output:            OK
forktest:                   (2.0s)
  -check result:            OK
  -check output:            OK
cowtest:                    (2.1s)
  -check result:            OK
  -check output:            OK
dirtycow_test:              (2.2s)
  -check result:            OK
  -check output:            OK
Total Score: 150/150

```

Challenge2

1.用户程序是何时被预先加载到内存中的？

用户态应用程序在编译内核镜像阶段就已经作为数据段一起打包进内存中。

其过程如下：

1. `make` 过程会把所有 `user/*.c` 编译为 ELF 格式的用户程序。
2. 这些 ELF 程序会被链接脚本安排放置在内核镜像的末尾。
3. 当 QEMU 启动并加载 `kernel` 镜像时，这些用户程序就随内核一起被装入物理内存。

2.与常见操作系统的加载方式有什么区别？

项目	ucore（教学系统）	Linux / Windows（真实 OS）
用户程序存放位置	编译内核时打包在内核镜像内存中	存储在磁盘文件系统中
用户程序加载时间	内核启动时一次性加载到内存	exec 时从磁盘按需加载
exec 的实现	直接从内存中的 ELF 模板复制到用户空间	读取 ELF 文件、建立 VMA、按需页缺页加载
文件系统依赖	不依赖文件系统（lab5 无 FS）	强依赖 FS（ext4, NTFS 等）
程序数量	少量固定测试程序	任意可执行文件
安全隔离/权限	简化实现，不涉及权限管理	完整的权限控制与文件访问控制

ucore 是预加载用户程序到内存，真实 OS 则是按需从磁盘加载可执行文件。

分支任务1（lab2）： gdb 调试页表查询过程

分支任务2（lab5）： gdb 调试系统调用以及返回