Lab5 Copy on-write

一、环境搭建

本实验在 xv6-labs-2021 的 `cow` 分支中进行。首先执行 `git fetch` 和 `git checkout cow` 切换分支，然后使用 `make clean` 清理旧的构建文件，确保实验环境干净、无缓存。编译和运行环境依旧使用 Ubuntu 22.04，并在 QEMU 模拟器中运行 xv6。编译过程中未引入额外工具链，仅在 xv6 源码中进行修改。

二、实验目的

本实验旨在通过实现 Copy-on-Write（COW）机制，优化 xv6 中 `fork()` 系统调用在内存拷贝方面的性能开销。通过延迟复制内存页，仅在子进程或父进程对页面进行写操作时才真正执行物理内存复制，从而提高资源利用效率。通过该实验进一步理解虚拟内存、页表标志位、页错误处理、引用计数等核心机制。

三、实验内容

- 修改 `uvmcopy()` 函数：
  - 不再直接复制物理内存页，而是将父进程页表中的每一页映射到子进程页表中；
  - 清除原有 PTE 中的 `PTE_W` 位，并设置一个自定义的 `PTE_COW` 标志，表明该页为共享的 Copy-on-Write 页；
  - 更新引用计数数组中对应物理页的引用计数值。
- 修改 `usertrap()` 中页错误处理逻辑：
  - 检查是否为 store 页错误，且对应 PTE 设置了 `PTE_COW`；
  - 若满足条件，分配一个新的物理页，并将原页面内容复制到新页面；
  - 修改页表项，清除 `PTE_COW`，设置 `PTE_W`，并更新为新的物理页地址；
  - 同时更新原页面引用计数，必要时释放页面。
- 实现物理页引用计数机制：
  - 在 `kalloc.c` 中定义全局整型数组 `refcnt[]`，数组大小根据系统物理页数量计算；
  - 为每次 `kalloc()` 分配的新页设置引用计数为 1；
  - 在 `uvmcopy()`、`uvmunmap()`、COW 处理逻辑中根据情况对引用计数进行增加或减少；
  - `kfree()` 仅在引用计数为 0 时才真正释放物理页。
- 修改 `copyout()` 函数：
  - 在将数据拷贝到用户页时检查是否为 COW 页；
  - 若是，执行与 page fault 时相同的拷贝逻辑，确保不会直接写共享页。

四、实验结果分析

执行 `cowtest` 程序，原本无法通过的 `simple` 测试可以成功运行并显示 `simple: ok`。随后 `three` 和 `file` 子测试也均能输出 `ok` 并通过所有测试项。最终输出 `ALL COW TESTS PASSED`，验证了 Copy-on-Write 实现的正确性。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$
$
$
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
```

同时运行 `usertests`，系统通过了包括内存、fork、exec 等在内的全部测试项，输出 `ALL TESTS PASSED`，表明 COW 实现未引入额外错误，系统稳定性良好。

```
== Test running cowtest ==
$ make qemu-gdb
(5.2s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(79.9s)
    (Old xv6.out.usertests failure log removed)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
```

五、实验中遇到的问题及解决方法

在实现 `uvmcopy()` 时初始只修改了子进程的页表 PTE，未同步更新父进程页表的 `PTE_W` 标志，导致子进程写入触发 page fault 后拷贝新页，但父进程仍可写原页，破坏了共享页面的一致性。修复方式为在 COW 设置时同时修改父子进程对应 PTE。

另一个问题出现在引用计数管理上，最初未对 `uvmunmap()` 中页表项为 COW 的页面进行引用计数的减少，导致部分页面无法释放，造成内存泄漏。补充逻辑后，每次 `uvmunmap()` 均检查是否为 COW 页并正确递减计数。

六、实验心得

本实验较之前的实验更贴近真实操作系统中的内存优化实践。通过实现 Copy-on-Write，不仅提升了 `fork()` 系统调用的效率，还引导我深入理解页表的标志位设计与软硬件协作机制。尤其是在处理 page fault 与引用计数时，锻炼了我对边界情况与资源管理细节的思考能力。整个实现过程虽有挑战，但收获颇丰，为进一步探索现代操作系统中的虚拟内存管理机制打下坚实基础。

附：实验部分源码

修改 `kalloc.c` 添加物理页引用计数

```
// 引用计数数组
int refcnt[PHYSTOP / PGSIZE];

void* kalloc(void) {
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if (r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if (r) {
    memset((char*)r, 5, PGSIZE);
    refcnt[(uint64)r / PGSIZE] = 1;  // 设置初始引用为1
  }
  return (void*)r;
}

void kfree(void *pa) {
  if (refcnt[(uint64)pa / PGSIZE] > 1) {
    refcnt[(uint64)pa / PGSIZE]--;
    return;
  }
  refcnt[(uint64)pa / PGSIZE] = 0;

  memset(pa, 1, PGSIZE);
  struct run *r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

修改 `vm.c` 的 `uvmcopy()` 实现 COW

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz) {
  for (uint64 i = 0; i < sz; i += PGSIZE) {
    pte_t *pte = walk(old, i, 0);
    if (!pte || !(*pte & PTE_V)) return -1;
```

```
    uint64 pa = PTE2PA(*pte);
    *pte &= ~PTE_W;              // 清除写权限
    *pte |= PTE_COW;             // 设置COW标志（自定义宏）

    if (mappages(new, i, PGSIZE, pa, PTE_FLAGS(*pte)) != 0)
      return -1;

    refcnt[pa / PGSIZE]++;       // 增加引用计数
  }
  return 0;
}
```

修改 `trap.c` 的 `usertrap()` 添加 COW 页错误处理

```
extern char trampoline[];

void usertrap(void) {
  struct proc *p = myproc();

  if ((r_scause() & 0xFFF) == 13 || (r_scause() & 0xFFF) == 15) {
    uint64 va = r_stval();
    va = PGROUNDDOWN(va);

    pte_t *pte = walk(p->pagetable, va, 0);
    if (!pte || !(*pte & PTE_V) || !(*pte & PTE_COW)) {
      p->killed = 1;
      return;
    }

    uint64 pa = PTE2PA(*pte);
    char *mem = kalloc();
    if (mem == 0) {
      p->killed = 1;
      return;
    }

    memmove(mem, (char*)pa, PGSIZE);
    *pte = PA2PTE((uint64)mem) | PTE_FLAGS(*pte);
    *pte &= ~PTE_COW;
    *pte |= PTE_W;

    refcnt[pa / PGSIZE]--;
    sfence_vma();
    return;
  }
}
```

修改 `copyout()` 支持写时复制

```
int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len) {
  while (len > 0) {
```

```
    uint64 va0 = PGROUNDDOWN(dstva);
    pte_t *pte = walk(pagetable, va0, 0);
    if (!pte || !(*pte & PTE_V)) return -1;

    if ((*pte & PTE_COW) || !(*pte & PTE_W)) {
      uint64 pa = PTE2PA(*pte);
      char *mem = kalloc();
      if (!mem) return -1;

      memmove(mem, (char*)pa, PGSIZE);
      *pte = PA2PTE((uint64)mem) | PTE_FLAGS(*pte);
      *pte &= ~PTE_COW;
      *pte |= PTE_W;
      refcnt[pa / PGSIZE]--;
    }

    uint64 n = PGSIZE - (dstva - va0);
    if (n > len) n = len;
    memmove((void*)(PTE2PA(*pte) + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva += n;
  }
  return 0;
}
```