Lab 5 Copy-on-Write Fork for xv6

Implement copy-on write(hard)

实验目的

本实验的目标是在操作系统中实现写时复制（Copy-on-Write）技术，以优化 fork 系统调用的性能。写时复制是一种页面管理技术，它在 fork 调用时，只复制父进程的页表，而不复制物理内存中的内容。这样可以推迟实际的页面复制，节省了不必要的时间和内存开销，提高了程序的执行效率。

实验步骤

1. 阅读源代码和文档

在开始实验之前，首先仔细阅读操作系统的源代码和相关文档，特别是关于 fork 系统调用的实现和页表管理的部分。

2. 添加 PTE_COW 标志

在 kernel/riscv.h 文件中添加一个新的 PTE（Page Table Entry）标志 PTE_COW，用于标记写时复制页面。

```
#define PTE_COW (1L << 8)  // copy on write page
```

3. 页面引用计数

在 kernel/kalloc.c 文件中，为每个页面维护一个引用计数。在页面分配时，将页面的引用计数初始化为 1。

```
uint8 page_count[PHYSTOP / PGSIZE + 1];  // 页面引用计数

// 在页面分配时将引用计数置为 1
void *kalloc(void)
{
  // ...其他代码...
  page_count[(uint64)r / PGSIZE] = 1;
}
```

4. 修改 fork 实现

在 kernel/vm.c 文件中，修改 uvmcopy() 函数的实现。删除实际物理页面复制的语句，只复制父进程的页表，并将父子页表项（PTE）的写权限去除，并标记为写时复制页（加上 PTE_COW 标记）。同时，需要增加物理页面的引用计数。

```

```

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;

  for (i = 0; i < sz; i += PGSIZE) {
    if ((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if ((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    *pte = *pte & (~PTE_W); // 去除写权限
    *pte = *pte | PTE_COW;  // 标记为写时复制页
    flags = PTE_FLAGS(*pte);
    if (mappages(new, i, PGSIZE, pa, flags) != 0) {
      goto err;
    }
    incref(pa); // 增加物理页面的引用计数
  }
  return 0;

err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

5. 处理页面错误

在 `kernel/trap.c` 文件中，修改 `usertrap()` 函数，添加对页面错误的处理。当页面错误是写错误时（`sscause` 寄存器值为 `15`），调用 `owfault()` 函数进行写时复制处理。

```
void usertrap()
{
  // ...其他代码...
  else if (r_scause() == 15) {
    // page fault (write)
    struct proc* p = myproc();
    if (cowfault(p->pagetable, r_stval()) < 0) {
      p->killed = 1;
      exit(-1);
    }
  }
}
```

6. 实现 `cowfault()`

在 `kernel/vm.c` 文件中，实现 `cowfault()` 函数，处理写时复制引发的页面错误。根据页面引用计数判断是否需要分配新的物理页面，并恢复进程的 `PTE` 写权限。

```
int cowfault(pagetable_t pagetable, uint64 rval) {
  if (rval >= MAXVA) {
    return -1;
  }
  pte_t* pte = walk(pagetable, rval, 0);
  if (pte == 0 || !(*pte & PTE_COW)) {
    printf("not cow page\n");
    return -1;
  }

  // 引用计数大于等于2，需要分配新的物理页面
  if (page_count[PTE2PA(*pte) / PGSIZE] >= 2) {
    page_count[PTE2PA(*pte) / PGSIZE]--;
    char* mem;
    if ((mem = kalloc()) == 0) {
      return -1;
    }
    memmove(mem, (char*)PTE2PA(*pte), PGSIZE);
    // 设置 PTE 为新的物理页面
    uint flag = PTE_FLAGS(*pte) | PTE_W;
    flag = flag & (~PTE_COW);
    (*pte) = PA2PTE((uint64)mem) | flag;
  } else {
    // 引用计数为1，直接恢复写权限即可
    (*pte) |= PTE_W;
    (*pte) &= (~PTE_COW);
  }
  return 0;
}
```

实验中遇到的问题和解决办法

1. 问题：在实现写时复制的过程中，页面引用计数的正确维护非常重要。如果引用计数没有被正确增加或减少，可能会导致页面在不应该释放的情况下被释放，从而引发系统崩溃或数据错误。

- 解决办法：为了确保页面引用计数的正确维护，我在页面分配时将页面的引用计数初始化为 1。在复制页面或进行写时复制时，通过增加页面的引用计数来防止页面被意外释放。在页面不再被使用时，需要及时减少页面的引用计数，并在引用计数为 `0` 时释放页面。

2. 问题：在写时复制过程中，需要正确判断页面是否需要进行复制。如果没有正确判断，可能会导致页面被错误地共享，从而影响进程间的独立性和数据完整性。

- 解决办法：为了正确处理写时复制，我在页面错误处理函数（`cowfault()`）中进行了如下判断：

- 首先，判断触发页面错误的虚拟地址是否有效，并检查页面是否被标记为写时复制页面（使用 `PTE_COW` 标志）。

- 如果页面不是写时复制页面，表示出现了异常情况，我选择终止进程（通过设置 `killed` 标志并调用 `exit()` 函数）。

- 如果页面是写时复制页面，并且引用计数大于等于 2，表示页面正在被多个进程共享，此时需要为新的进程分配一个新的物理页面，并将原页面的内容复制到新页面。 如果引用计数为 `1`，则表示当前进程是页面

3. 问题：在处理页面错误时，需要确保进程的状态与页面错误的处理一致，否则可能导致进程出现错误状态或数据不一致。

- 解决办法：为了保持进程的状态一致性，我在处理页面错误时，会根据具体情况将进程标记为被杀死（`killed` 标志为 `1`）并调用 `exit()` 函数终止进程。这样可以确保页面错误处理与进程状态的一致性，并且避免了可能的错误状态和数据不一致。

实验心得

通过本次实验，我深入理解了写时复制技术在操作系统中的应用以及其带来的性能优化。在实现写时复制的过程中，我学会了如何处理页面引用计数、页面分配与释放，并且学会了在内核代码中增加新的系统调用。这些知识和经验对于我理解操作系统的内核原理以及进行系统级编程都非常有帮助。

在实验过程中，我遇到了一些问题，如页面引用计数的正确处理、页面权限的设置等，但通过查阅资料和与同学讨论，最终成功解决了这些问题。同时，在进行代码修改时，我也学会了备份原始代码，确保实验的安全性。

评分

```
== Test running cowtest ==
$ make qemu-gdb
(13.6s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(286.9s)
    (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
```