同济大学操作系统课程设计 - Xv6 - Lab: Copy-on-Write Fork for xv6

【实验目的】

虚拟内存通过间接性实现灵活管理，比如将页表项 (PTE) 标记为无效或只读，导致页面故障 (page faults)。在 `xv6` 的 `fork()` 系统调用中，写时复制 (Copy-On-Write, COW) 是一种优化策略，它推迟了物理内存页的分配，直到进程真正需要写入数据时才进行复制。

COW `fork()` 的实现中，父进程和子进程共享物理页面，并将 PTE 标记为不可写。当进程试图写入这些共享页面时，CPU 会触发页面故障，内核检测到这一情况后分配新页面，复制数据，并更新 PTE 为可写。这样，内存的实际分配和复制仅在必要时进行。

内存释放时，由于多个进程可能共享同一物理页，只有当所有引用都消失时才会释放该页。这种机制提高了内存使用效率，并加快了进程创建速度。

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

1 Implement copy-on write(hard)

要启动实验室，需要先切换到 cow分支：

```
$ git fetch
$ git checkout cow
$ make clean
```

1. 在 `kernel/riscv.h` 中设置新的 PTE 标志位
   - 添加 `PTE_RSW` 标志位，用于标识页面是否使用了 COW 机制：

     ```
     #define PTE_RSW (1L << 8) // RSW
     ```

2. 修改 `uvmcopy()` 函数以支持 COW
   - 在 `kernel/vm.c` 中，修改 `uvmcopy()` 函数，避免直接复制物理页面。相反，子进程的页表将指向父进程的物理页面。

- 父子进程的 PTE 中的 `PTE_W` 标志应被清除，并且标志位设置为 `PTE_COW`，以表示这些页面是写时复制页面。

```c
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_W;
    flags = PTE_FLAGS(*pte);
    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      goto err;
    }
    adjustref(pa, 1);
  }
  return 0;

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

3. 修改 `usertrap()` 处理页面错误
   - 在 `kernel/trap.c` 文件中，修改 `usertrap()` 函数，以便在发生写页面错误时（即 `r_scause` 寄存器的值为 `15`），调用 `cowfault()` 函数来处理写时复制页面的创建。

4. 为物理页面增加引用计数
   - 在 `kernel/kalloc.c` 中，为每个物理页面维护引用计数。在页面分配时，将引用计数初始化为 `1`，在 `freerange()` 函数中为所有物理页面设置引用计数为 `1`。

```c
if(r)
{
  memset((char *)r, 5, PGSIZE);
  int idx = PA2INDEX(r);
  if (cowcount[idx] != 0) {
    panic("kalloc: cowcount[idx] != 0");
  }
  cowcount[idx] = 1;
}
return (void*)r;
```

   - 在 `kfree()` 函数中，减少引用计数，只有在引用计数减至 `0` 时，才真正释放物理页面。

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  acquire(&kmem.lock);
  int remain = --cowcount[PA2INDEX(pa)];
  release(&kmem.lock);

  if (remain > 0) {
    return;
  }
```

5. 修改 `copyout()` 处理 COW 页面
    - 在 `kernel/vm.c` 中修改 `copyout()` 函数，使其能够处理 COW 页面。遇到 COW 页面时，该函数会像处理页面错误一样处理，调用 `cow_alloc()` 函数分配新的物理页面。

- 定义 `cow_alloc()` 函数来分配新的物理页面并复制数据。

6. 编译并运行 xv6
    - 使用 `make qemu` 命令编译并运行 xv6 系统。
7. 运行 `cowtest` 测试 COW 实现
    - 在 xv6 的命令行中输入 `cowtest`，以验证 COW 机制的正确性:

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab5$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$
```

8. 运行 `usertests` 进行全面测试
    - 最后，在命令行中输入 `usertests`，以确保 COW 机制没有影响到其他:

```
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test killstatus: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

通过以上步骤，COW `fork()` 的实现得以完成，成功地延迟了物理内存的分配，直到实际写入时才进行，从而提高了内存使用效率。

【分析讨论】

在学习操作系统课程时，书本中的理论知识有时显得抽象难懂。通过本次实验，我将理论付诸实践，逐步实现了 `fork()` 系统调用的 COW 机制。在实验过程中，我体会到了如何通过精心设计的机制来优化系统性能，并更好地理解了操作系统中内存管理的原理和实践方法。

写时复制的机制在许多现代操作系统中广泛应用。通过延迟物理内存的分配，COW 机制能够显著减少内存使用，并提高 `fork()` 系统调用的效率。这种延迟分配的思想，不仅仅在操作系统中适用，在其他资源管理场景中也能找到类似的应用场景。通过此次实验，我认识到了这种资源优化思路的广泛性和实际意义。

实验过程中，我深刻体会到关注细节的重要性。比如，在为物理页面增加引用计数时，我需要考虑如何确保只有在所有进程都不再引用该页面时才释放它。这种对细节的把控是操作系统稳定运行的保障。任何一个小错误，都可能导致系统的不稳定或资源泄漏。

通过本次实验，我不再仅仅停留在对操作系统的表面理解，而是深入到内核机制中，真正理解了系统调用的运作原理。这种理解让我更加自信地面对操作系统的其他内容，也为我今后在计算机领域的进一步学习和研究打下了坚实的基础。

总的来说，这次实验不仅是对书本知识的巩固，更是一次实战演练。通过实现 COW `fork()` 机制，我不仅加深了对操作系统内存管理的理解，还提升了自己的编码能力、问题解决能力和团队协作能力。对于未来的学习和工作，这次实验都将是一笔宝贵的经验。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
/uart.o kernel/spinlock.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
 /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/yzz/Desktop/xv6/xv6-2021/lab5'
== Test running cowtest ==
$ make qemu-gdb
(9.3s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(294.2s)
    (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
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab5$
```