Lab 3: Copy-on-Write Fork for xv6 - Alejandro Vargas (avarg116)

The fork() system call in the xv6 kernel is time consuming and wastes memory. It copies the entire parent process's user space memory into the child even if pages are not utilized. Copy-on-Write (COW) fork makes memory management more efficient by allocating and copying physical memory pages for the child until actually needed.

Step 1. Implement page reference counter

```
struct {
  struct spinlock lock;
  struct run *freelist;
  int ref_count[(PHYSTOP >> 12)]; //ref_count array use 12 right step (3.1)
} kmem;

void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
  for(int i= 0; i < (PHYSTOP >> 12); i++) {
    kmem.ref_count[i]= 0;
  }
}
```

struct: the array ref_count is used to store the ref count for each physical page. The decision to use >> 12 is due to 4096 bytes being the page size in many systems (textbook 3.1). In kinit, we initialize the ref count for all pages to 0.

```
  int index = (uint64)r >> 12;
  kmem.ref_count[index] = 1;
```

kalloc: after allocating a new page, the reference count is set to 1.

```
acquire(&kmem.lock);
int index= (uint64)r >> 12;
kmem.ref_count[index] -= 1; //decrement ref_count
int reference = kmem.ref_count[index];
release(&kmem.lock);

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

kfree: the ref count is decremented. The page is added to the free list if it reaches 0 (released). Modification to the ref count should be guarded by a lock, so we decrement after acquiring and before releasing the kmem.lock spinlock.

```
//added step 4
void incref(uint64 pa) {
  //struct run *r = (struct run*)pa;

  acquire(&kmem.lock);
  int index = (uint64)pa >> 12;
  kmem.ref_count[index] += 1;
  release(&kmem.lock);
}
```

incref: new function to increment the ref count of physical page. This ensures that the physical page will only be freed when the last PTE referencing it is no longer in use. We guard by a lock here too. In the defs.h file, incref is declared under kalloc.c to call the function in other files.

Step 2. Fix uvmcopy() function

```
//step 1: remove new page alloc
/*
if((mem = kalloc()) == 0)
  goto err;
memmove(mem, (char*)pa, PGSIZE);
*/

//step 3: clear PTE_W
flags |= PTE_COW; //step 4: privilege flag COW
flags &= ~PTE_W;

if(mappages(new, i, PGSIZE, pa, flags) != 0){ //step 2: map parent's physical page to child's virtual
  //kfree(mem);
  goto err;
}
uvmunmap(old, i, PGSIZE, 0);
if(mappages(old, i, PGSIZE, pa, flags) != 0){
  //kfree(mem);
  goto err;
}

incref(pa); //added step 5
```

First, I removed the new page allocation for the child process (//comment mem = kalloc() and memmove). Instead, we map the parent's physical page to child's virtual page using mappages. Flags is modified to include the COW flag and clear the write permission (PTE_W) from both the parent and child PTEs. This indicates the pages are COW mapped. The xv6 book chapter 3.1 tells us about the structure of RISC-V page tables and the flags for PTE (PTE_V, PTE_R, etc). It implies that we need to add a custom flag for Copy-on-Write. In the riscv.h file, I defined a privilege flag PTE_COW to record whether a PTE is COW mapping. The last step is to increment the page ref count using incref.

Step 3. Fix usertrap() function

```
} else if(r_scause () == 15) { //step 1:

uint64 start_va = PGROUNDDOWN(r_stval());
pte_t *pte = walk(p->pagetable, start_va, 0); //PTE for faulting page

if (pte == 0 || !(*pte & PTE_V) || !(*pte & PTE_U) || !(*pte & PTE_COW)) {
  p->killed = 1;
} else  { //page fault to COW page:
  uint flags = PTE_FLAGS(*pte);
  flags |= PTE_W;
  flags &= ~PTE_COW;

  //step 2:
  char *mem = kalloc(); //allocate new physical page
  if (mem == 0) {
    p->killed = 1;
  } else {
    char *pa = (char *)PTE2PA(*pte);
    memmove(mem, pa, PGSIZE); //duplicate content of COW page

    //step 3 remap:
    uvmunmap(p->pagetable, start_va, PGSIZE, 0);
    kfree(pa);
    if (mappages(p->pagetable, start_va, PGSIZE, (uint64)mem, flags) != 0) {
      //kfree(mem);
      p->killed = 1;
    }
  }
}
```

First, the (r_scause() == 15) condition checks if the cause of the trap is a writing page fault. Next, we check if the page is marked as COW and kill the process for invalid page fault conditions. A new physical page is allocated using kalloc() and duplicates the content of COW page into a new physical page using memmove(). The faulting virtual page is unmapped and frees the old physical page with kfree(pa). Finally, it is mapped to the new page with updated PTE_W flag.

Step 4. Fix copyout() function

```
//MODIFICATIONS
pte_t *pte = walk(pagetable, va0, 0);

if((pte == 0) || (*pte & PTE_U) == 0 || (*pte & PTE_V) == 0) {
  return -1;
}

if (pte && (*pte & PTE_COW)) { //if COW page
  char *mem = kalloc(); //allocate new page
  if (mem == 0)
    return -1;

  flags = PTE_FLAGS(*pte);
  flags |= PTE_W;
  flags &= ~PTE_COW;

  char *pa = (char *)PTE2PA(*pte);
  memmove(mem, (void*)pa, PGSIZE); //copy COW page content to new one

  uvmunmap(pagetable, va0, PGSIZE, 0);
  kfree((void*)pa);
  //remap
  if (mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0) {
    //kfree(mem); //decrement
    return -1;
  }
  //pte_t *new_pte = walk(pagetable, va0, 0); //make new page writible
  //*new_pte |= PTE_W;
  //*pte &= ~PTE_COW;
  //pa0 = (uint64)mem; //update pA0 -> new physical addr
}
pa0 = PTE2PA(*pte) ;
if(pa0 == 0)
  return -1;
```

The COW page is handled similarly to usertrap function. If the virtual address belongs to a COW page, it is allocated and copied. Then, we remap just as before. After the if condition, data from src is copied to dstva within the current page. The pa0 statement had to move. I added if va0 > MAXVA to fix a usertest error (virtual address does not exceed).

Testing

To read time.txt, timeout in grade-lab-cow was changed to 250 to extend the running time.

```
running cowtest:
$ make qemu-gdb
(14.8s)
  simple: OK
  three: OK
  file: OK
usertests:
$ make qemu-gdb
OK (213.4s)
time: OK
Score: 100/100
[avarg116@sledge cs179f-fall23]$
```