Project Report: Copy on Write (COW)

Harrison Williams(861168815), Aadhar Chauhan(862546563)

Feb 11, 2025

1 Modifications to xv6 Source Code for COW

1.1 Step 1: Modify fork() Behavior - uvmcopy()

The xv6 operating system creates processes using the `fork()` system call, which duplicates the parent process's address space. The default implementation copies memory pages by allocating new memory for the child process.

To implement Copy-on-Write (COW), modifications were made to `uvmcopy()` in `vm.c`. Instead of allocating new memory, the process page table entries were iterated over to obtain virtual addresses and convert them to physical addresses using `PTE2PA()` from `riscv.h`. The write bit `PTE_W` was cleared, and the reserved software bit `PTE_RSW` was set to make the pages read-only. These adjustments inf (Figure 1) allow parent and child processes to share physical memory efficiently.

```
pa=PTE2PA(*pte); //get phyical address of PTE in parent
flags= PTE_FLAGS(*pte); //extract flags/bits for the PTE in parent

//disable write and enable COW if the page was originally writable
if (flags & PTE_W) {
  *pte &= ~PTE_W;   //disable parent write
  *pte |= PTE_RSW;  //mark as COW page for parent
  flags &= ~PTE_W;  //update flags for child write bit
  flags |= PTE_RSW; //update flag as a COW page
}
```

Figure 1

The updated page table entries were mapped using `mappages()` (Figure 2) to associate the child's address space with the same physical frames as the parent.

```
//map pa to the child's page table rather than kalloc() and copying parent
if (mappages(new, i, PGSIZE, pa, flags) != 0)
  goto err;
```

Figure 2

1.2 Step 2: Handling Page Faults - usertraps()

If a process attempts to write to a read-only shared page, a page fault occurs. To handle this situation, modifications were made to the `usertraps()` function in `trap.c`. By default, `usertraps()` does not handle Copy-on-Write page faults, so additional logic was implemented to detect and process them.

```
//Proj2_ COW fault handler
}else if(r_scause() == 15){  //page fault exception handler 15=page
  uint64 va = r_stval();  //fetch virtual address of the fault
```

Figure 3

A new condition was introduced to check whether the page fault value stored in the `scause` register equals 15 (Figure 3). Before allocating a new memory page, validity checks are performed to ensure the faulting address and page attributes meet the required conditions (Figure 4).

```
//checks before allocating for new page
if(va >= MAXVA){ //check validity of va
  p->killed = 1;
  exit(-1);
}
if((pte=walk(pgTbl,va,0)) == 0){ //walk page table for va faulting
  p->killed = 1;
  exit(-1);
}
if((*pte & PTE_V) == 0 || (*pte & PTE_U) == 0){ //check for valid bit
  p->killed = 1;
  exit(-1);
}

if (!(*pte & PTE_RSW)) {
  printf("usertrap: page fault on non-COW page at %p\n", va);
  p->killed = 1;  //kill the process not a COW fault
  exit(-1);
}
```

Figure 4

Since the `walk()` function in `vm.c` is needed within `trap.c`, it is declared as an `extern` function. The fault handler then allocates a new physical page using `kalloc()`, copies the original page data, updates the page table entry flags, and finally deallocates the old physical page using `kfree()`(Figure 5).

With these changes, xv6 can now handle Copy-on-Write faults without causing segmentation faults.

```
//allocate new physical page
oldPA = PTE2PA(*pte); //get old page physical address
if((newPage= kalloc())==0){
  p->killed = 1;
  exit(-1);
}

//copy old page to new page and update PTEs
memmove(newPage, (char *)oldPA, PGSIZE);
flags = PTE_FLAGS(*pte); //fetch current flags
*pte = PA2PTE((uint64)newPage); //set new physical page
*pte |= flags; // restore flags
*pte |= PTE_W; //set write bit

kfree((void*)oldPA); //free old physical page
```

Figure 5

```
acquire(&pa_ref_lock);
pa_ref_count[pa/PGSIZE]++;;
release(&pa_ref_lock);
```

Figure 6

1.3 Step 3: Reference Counting - uvmcopy() and kalloc.c

Multiple processes can share the same physical frame in memory, so it is necessary to implement reference counting to manage shared frames efficiently. When a process terminates, its pages must not be deallocated if other processes still reference them.

```
extern struct spinlock pa_ref_lock;
extern int pa_ref_count[1<<20];
```

Figure 7

A reference counter tracks the number of processes using a physical frame (Figure 6). This count is incremented in the `fork()` process when memory is shared and decremented when a process releases its pages. Only when the reference count reaches zero is the physical page in memory freed.

A spinlock is needed to protect the the critical section of reference counter updates to ensure consistency (Figure 7). The reference count is stored in an array `pa_ref_count[PHYSTOP/PGSIZE]`. During system initialization in `kinit()`, the spinlock is initialized, and the reference counters are set to 1 in `freerange()` (Figure 8).

```
kinit()
{
  initlock(&pa_ref_lock, "pa_ref_lock"); //proj2- init ref counter lock
```

```
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE){
    //initially set counter to 1
    acquire(&pa_ref_lock);
    pa_ref_count[((uint64)p)/PGSIZE]=1;
    release(&pa_ref_lock);
```

Figure 8

```
kfree(void *pa)
{
  //decrease ref. counter
  acquire(&pa_ref_lock);
  pa_ref_count[((uint64)pa)/PGSIZE]--;
  release(&pa_ref_lock);

  if(pa_ref_count[(uint64)pa/PGSIZE] ==0){
```

Figure 9

The `kfree()` (Figure 9) function is modified to decrement the reference count before freeing a page. If the count is greater than zero, the page is not deallocated. Similarly, `kalloc()` ensures that newly allocated pages from `usertraps()` start with a reference count of 1 (Figure 10).

```
//init counter to 1
acquire(&pa_ref_lock);
pa_ref_count[((uint64)r)/PGSIZE]=1;
release(&pa_ref_lock);
```

Figure 10

1.4 Step 4: Modification of copyout() Function

The `copyout()` function copies data from the kernel space to user space. With Copy-on-Write, multiple processes reference the same pages in memory, requiring modifications to `copyout()` to manage these cases.

```
while(len > 0){
  va = PGROUNDDOWN(dstva); //align va to page boundires

  if(va >= MAXVA || dstva >= MAXVA) //check va are valid in user space
    return -1;
  if((pte = walk(pagetable, va, 0)) == 0) //find PTE in page tabel
    return -1;
  if((*pte & PTE_V) == 0 || (*pte & PTE_U) == 0) //check valid bit and user bit
    return -1;
```

Figure 11

Inside the `copyout()` function, pages are aligned using `PGROUNDDOWN()` (Figure 11), and the page table is accessed using `walk()`. If a write attempt is made to a read-only page, a new physical page is allocated using `kalloc()`. The data is then copied from the old page to the newly allocated one, then page flags are updated, and the old page is freed. The function also ensures that data is copied correctly without exceeding page

```
//handle COW page if page is read-only
if(((*pte) & PTE_W) == 0){
  if((new_pa = kalloc()) == 0) //allocate new physical page
    return -1;
  memmove(new_pa, (char*)pa, PGSIZE); //copy from old to new page
  flags = PTE_FLAGS(*pte);
  *pte = PA2PTE(new_pa);
  *pte |= flags;
  *pte |= PTE_W;

  kfree((void*)pa); //free old page
}else{
  new_pa=(char*)pa; //new page already writable
}
```

Figure 12

boundaries by using the `PGSIZE` (Figure 12) macro and the specified length from the function parameter `len`. The `memmove()` (Figure 13) function is used to perform the actual data copying.

```
//determine size of data to copy
n = PGSIZE - (dstva - va); //amount of data which fits on page
if(n > len)
  n = len; //copy remaining length if it fits in currnet page

//copy data from kernel source buffer to destination page
memmove((void *)(new_pa + (dstva - va)), src, n);
len -= n; //go to next page and repeat process
src += n;
dstva = va + PGSIZE;
```

Figure 13

2 Compiling and Debugging Execution in QEMU

The environment was set up by updating the `.bashrc` file to source `cs202`. Without this, compilation and the auto-grading script failed.

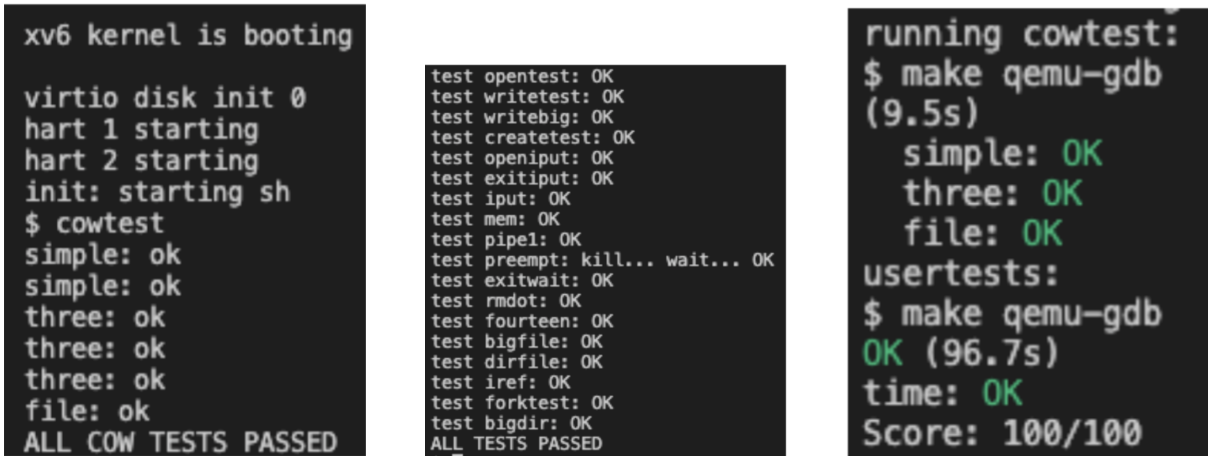
During compilation, an error was encountered in `sh.c` due to the `runcmd()` function being labeled as infinite recursion. The `__attribute__((noreturn))` directive was added to resolve this.

Files `.gdbinit` and `.gdbinit.tmpl-riscv` were lost during Github commits and were restored to enable `make qemu-gdb` and `make grade` to work properly.

3 Execution and Results

The system was booted, and individual tests `cowtest` and `usertest` were successfully executed with the message `ALL TESTS PASSED` at the end of the test.

Running `cowtest` verified that the simple, three, and file tests passed (Figure 14). Running `usertests` verified that all tests in `user/usertests.c` passed (Figure 14). The final test was `make grade`, which executed all tests automatically. The grading script `grade-lab-cow` reported a score of 100/100 (Figure 14).



(a) (b) (c)

Figure 14: Results from COW tests, user tests, and grading script.

Video Demonstration

The project explanation and demonstration video can be accessed on YouTube using the following link:

Project Demonstration Video

Alternatively, here is the raw URL: `https://youtu.be/wZlJz6xOoyk`

The github repo can be accessed via below link

Github_Project2

The link to the gitdiff file is as follows: Gitdiff