LAB6 REPORT - COPY ON WRITE
CS19B043 - T. BHANU SHASHANK

1)

a) Modified the uvmcopy() to map the parent's physical pages to child
b) Cleared the PTE_W Bits in both parent and child
c) Finally setting PTE_C bits.

Note: RSW (1ll << 8) and PCOUNTACCESS(va) are defined in riscv.h file.

Code : (vm.c file)

```
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){                          //For each page frame.
    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);
    flags = PTE_FLAGS(*pte);
    flags &= ~(PTE_W);                                        // clear the write bit
    flags |= RSW;                                             // set the RSW bits to indicate a COW page
    *pte = PA2PTE(pa) | flags;
    if(mappages(new, i, PGSIZE, pa, flags) != 0)
    {
      printf("page mapping error in old\n");
      goto err;
    }
    pcount[PCOUNTACCESS(pa)]++;                             //increasing pcount to store number of owner's
  }
  return 0;
 err:
  uvmunmap(new, 0, i, 1);
  return -1;
}
```

Explanation :

a) For each page frame of both parent and child, clear the write bit.
b) Set the RSW bit to indicate it is Copy on the write page.
c) Increasing the pcount to store the number of owner's.

2) Modified usertrap() to recognize page faults. When a page-fault occurs on a COW page, allocate a new page with kalloc(), copy the old page to the new page, and install the new page in the PTE with PTE_W set in trap.c file.

Code : (usertrap() in trap.c file)

```
else if(r_scause() == 15){                                              // Store/AMO page fault
    uint64 faulting_va, pa;
    pte_t *pte; uint flags; char *mem;
    faulting_va = r_stval();
    if((faulting_va >= MAXVA) || ((pte = walk2(p->pagetable,faulting_va,0))==0)){
       printf("page mapping does not exist\n");
       p->killed = 1;
       exit(-1); }
    if(((*pte & PTE_V) ==0) || ((*pte & PTE_U) == 0)){
        printf("page mapping does not exist\n");
        p->killed = 1;
        exit(-1); }
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if(*pte & RSW) {   // COW page check.
      if(pcount[PCOUNTACCESS(pa)]==1){       //If there is single owner then just set the write bit
         *pte &= ~(RSW);
         *pte |= PTE_W;
      }
      else{
        if((mem = kalloc()) == 0){                                              //allocate memory
          printf("no free pages\n");
          p->killed = 1;
          exit(-1);
        }
        memmove(mem, (char *)pa, PGSIZE);                     //copy old page to new page
        flags &= ~(RSW);                                                      //clear RSW bits
        flags |= PTE_W;                                                       //set write bit
        uvmunmap(p->pagetable, PGROUNDDOWN(faulting_va), 1, 1);
 if(mappages(p->pagetable, PGROUNDDOWN(faulting_va), PGSIZE, (uint64)mem, flags) != 0){
          kfree(mem);
          printf("page mapping error\n");
          p->killed = 1;
          exit(-1); }
      }
    }
    else {
      printf("invalid page fault\n");
      p->killed = 1;
      exit(-1);}
```

Explanation:

a) Which type of interrupt is stored in the scause register if scause value is 15 (page-fault).
b) Checking if there is a single owner of a COW page, then it's cow and non-writable flags can be reset and it need not obtain a copy of the page.
c) Else allocate new page and copy old page to new page with set PTE_W bit in PTE.
d) Followed by sanity checks.

3) Each physical page is free-ed when the last PTE reference to it goes away, by implementing reference counts in kalloc.c.

Code:

```
extern int pcount[PCOUNTACCESS(PHYSTOP)];
void kfree(void *pa){
  struct run *r;
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");
  if(((uint64)pa >= KERNBASE) && pcount[PCOUNTACCESS((uint64)pa)] > 0) {
    pcount[PCOUNTACCESS((uint64)pa)]--;       // Decreasing the owner’s by one
    if(pcount[PCOUNTACCESS((uint64)pa)] > 0) {
      return;
    }
  }
  // Free-up the page if it last PTE reference.
  memset(pa, 1, PGSIZE);                              // Fill with junk to catch dangling refs.
  r = (struct run*)pa;
  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}

void *kalloc(void){
  struct run *r;
  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);
  if(r){
    if((uint64)r >= KERNBASE) {
       pcount[PCOUNTACCESS((uint64)r)] = 1;      // Intialize the pcount with one
    }
    memset((char*)r, 5, PGSIZE); // fill with junk
  }
  return (void*)r;}
```

Explanation:

a) Checking if it is the last PTE reference or not.
b) If it is the last PTE reference then it is free-ed and added to kfree list.
c) Else don’t change anything.
d) And second function kalloc initializes the pcount to one,since it is allocationing first time.

4) Modified copyout() to use the same scheme as page faults when it encounters a COW page.

Code: (additional code in copyout function in vm.c file)

```
  if((va0 >= MAXVA) || ((pte = walk(pagetable,va0,0))==0)){
     printf("invalid page\n");
     return -1;
   }
   if(((*pte & PTE_V) ==0) || ((*pte & PTE_U) == 0)){
     printf("invalid page\n");
     return -1;
   }
  pa0 = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);
  if(*pte & RSW) {                                          // COW page
    if((mem = kalloc()) == 0)                              //allocating memory
    {
     printf("no free pages\n");                           //no free pages present.
     return -1;
    }
    memmove(mem, (char *)pa0, PGSIZE);
    flags &= ~(RSW); //clear RSW bits
    flags |= PTE_W; //set write bit
    uvmunmap(pagetable, va0, 1, 1);
    if(mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0)
    {
     kfree(mem);
     printf("page mapping error\n");
     return -1;
    }
  }
```

Explanation:

a) Checking where it is COW page or not using the RSW bit.
b)Allocate the new page ,copy the old page and set the flag bit such that it is writable(PTE_W).
c)There are some sanity checks in between.
d)Here implementation is similar to page-fault handled in user trap() in trap.c.