In this lab assignment, **Copy On Write** feature is implemented in the xv6 OS.
The following files had to be changed for implementing it, about which I describe in detail in each of the sections.

1. *kernel/kalloc.c*
2. *kernel/riscv.h*
3. *kernel/vm.c*
4. *kernel/trap.c*
5. *kernel/defs.h*

1 kalloc.c

To keep track of the reference counts for each page, an `ref_count` array of size `((PHYSTOP - KERNBASE) / PGSIZE) + 1` is created and is indexed from address by the mapping `ref_index(a) = (a - KERNBASE) / PGSIZE`. Though user pages exist only after `end`, array is created for pages from `KERNBASE` to `PHYSTOP` to be of static size. The array is defined inside `struct kmem` and uses its spinlock to serialise modification access to it.

kernel/kalloc.c

```
struct {
  struct spinlock lock;
  struct run *freelist;
  int ref_count[(PHYSTOP-KERNBASE) / PGSIZE + 1];
} kmem;
```

Functions are defined to increment the reference count for an address and also get the reference count given the address. The prototypes of these functions are added to *kernel/defs.h*.

kernel/kalloc.c

```
void
kaddref(void* pa)
{
  acquire(&kmem.lock);
  kmem.ref_count[ref_index(pa)]++;
  release(&kmem.lock);
}

uint
kgetref(void* pa)
{
  return kmem.ref_count[ref_index(pa)];
}
```

The `kfree()` function is modified to decrement the reference count when it is called and only when the count falls to zero (or below), the page is released to `freelist`.

kernel/kalloc.c

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

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

  uint idx = ref_index(pa);
  acquire(&kmem.lock);
  kmem.ref_count[idx]--;
  release(&kmem.lock);

  if(kmem.ref_count[idx] <= 0){
    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    kmem.ref_count[idx] = 0;
    release(&kmem.lock);
  }
}
```

The `kalloc()` function initialises reference count of the page procured from the `freelist` to 1.

kernel/kalloc.c

```
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    kmem.ref_count[ref_index(r)] = 1;
  }
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

2 riscv.h

A reserved bit in PTE is used to denote COW page.

kernel/vm.c/walkpte()

```
#define PTE_COW (1L << 8) // Bit for COW - Lab 6
```

3 vm.c

A new function `walkpte()` is added that walks the pagetable for the given virtual address using the `walk()` function, checks if the PTE is valid and accessible by user space like `walkaddr()` and returns the PTE.

kernel/vm.c/walkpte()

```
pte_t *
walkpte(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;

  if(va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
  if(pte == 0)
    return 0;
  if((*pte & PTE_V) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;

  return pte;
}
```

Next, the `uvmcopy()` function is modified so that instead of creating a new page for the child process, it simply maps the PTEs of the child process to the pages used by parent process. The COW bit is set to indicate that the page is used for COW which will be useful for identifying these types of page faults later. The Write bit is unset for the page to generate a fault when there is an attempt to write to it.

kernel/vm.c/uvmcopy()

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint newflags;
  // char *mem;

  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);
    newflags = PTE_FLAGS(*pte);
    // if((mem = kalloc()) == 0)
    //   goto err;
    // memmove(mem, (char*)pa, PGSIZE);
    newflags |= PTE_COW;
    newflags &= (~PTE_W);

    if(mappages(new, i, PGSIZE, pa, newflags) != 0){
      // kfree(mem);
      goto err;
    }
    else{
      *pte = (*pte | PTE_COW) & ~PTE_W;
      kaddref((void *)pa);
    }
  }
  return 0;

 err:
```

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

`copyout()` is a function that writes to a user page from memory in kernel space. So it is modified to allocate a new page for the user process if the page to which writing is attempted is a COW page. The process's previous page corresponding to the address is unmapped, contents are copied from previous page to new page and finally new page is mapped to that address. Also, the reference count is decremented for the earlier mapped page. If the reference count for the earlier page falls to 1, then its COW bit is cleared and Write bit is set since the previous page is only mapped to a single process now.

kernel/vm.c/copyout()

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;
  pte_t *pte;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pte = walkpte(pagetable, va0);
    if(pte == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;

    if(*pte & PTE_COW){
      uint flags = PTE_FLAGS(*pte);
      uint64 pa = PTE2PA(*pte);
      char* mem;
      if((mem = kalloc()) == 0){
        return -1;
      }
      memmove(mem, (char*)pa, PGSIZE);

      uvmunmap(pagetable, va0, 1, 0);
      kfree((void*)pa);
      if(kgetref((void *)pa) <= 1){
        *pte = (*pte | PTE_W) & ~PTE_COW;
      }

      if(mappages(pagetable, va0, PGSIZE, (uint64)mem,
                  (flags | PTE_W) & ~PTE_COW) != 0){
        return -1;
      }
      pa0 = (uint64)mem;
    }
    else pa0 = PTE2PA(*pte);

    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

4 trap.c

In the `usertrap()` function, a code section is added if `r_scause() == 15` which denotes load page faults. The code added checks if the page that generated the fault is a COW page and if that is the case, it allocates a new normal (non-COW) page, unmaps the process from earlier page and maps it to the new page after copying contents from earlier page to the new page. Also, the reference count is decremented for the earlier mapped page. If the reference count for the earlier page falls to 1, then the COW bit is cleared and Write bit is set as in `copyout()`.

kernel/trap.c/usertrap()

```
else if(r_scause()==15){   // 15: load page fault
    uint64 vpage_head = PGROUNDDOWN(r_stval());
    pte_t *pte;
    if((pte = walkpte(p->pagetable, vpage_head)) == 0){
      printf("usertrap(): scause=15 stval=%p page not found\n", r_stval());
      p->killed = 1;
      goto err;
    }

    if(*pte & PTE_COW){
      uint flags = PTE_FLAGS(*pte);
      uint64 pa = PTE2PA(*pte);
      char* mem;
      if((mem = kalloc()) == 0){
        printf("usertrap(): scause=15 kalloc failed during COW for process\n");
        p->killed = 1;
        goto err;
      }
      memmove(mem, (char*)pa, PGSIZE);

      uvmunmap(p->pagetable, vpage_head, 1, 0);
      kfree((void*)pa);
      if(kgetref((void *)pa) <= 1){
        *pte = (*pte | PTE_W) & ~PTE_COW;
      }

      if(mappages(p->pagetable, vpage_head, PGSIZE, (uint64)mem,
                  (flags | PTE_W) & ~PTE_COW) != 0){
        printf("usertrap(): scause=15 mappages failed during COW for process\n");
        p->killed = 1;
        goto err;
      }
    }

    else goto uerr;

  } else {
    uerr:
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }
  err:
  if(p->killed)
    exit(-1);
```

5 defs.h

All function prototypes and directives which are required in external programs are added here.

kernel/defs.h

```
// index corresponding to the address in ref_counts array of kmem
#define ref_index(a) (((uint64)(a) - KERNBASE) / PGSIZE)

pte_t*          walkpte(pagetable_t, uint64);

void            kaddref(void *);
uint            kgetref(void *);
```

The end result of these changes is that xv6 runs normally as before, and clears all `usertests` and `forktest`, just that pages are not all duplicated during `fork()` like before, but duplicated only when required. This saves a lot of memory and time especially when there are a lot of processes running and created, and they share libraries (or code) which are mostly not written into.