INDIAN INSTITUTE OF TECHNOLOGY MADRAS

Operating Systems : Lab 6

SATHVIK JOEL K

CS19B025

November 6, 2021

CONTENTS

1 SUMMARY

When a process forks a child, all the code, data and stack variables are copied to the child processes address space and the child continues its execution. However, most of the pages remain unaltered (For example, the pages belonging to the code section). Hence, it is better to keep a single copy of pages in physical memory and make both parent and child processes' virtual addresses point to the same memory. A new copy of the page is deferred until there is a write to the shared page. This is reffered to as the **COW** (copy on write)

2 UVMCOPY

- This function copies all the pagetables and the pages of the parent to the child.
- In COW implementation new pages are not allocated rather the old pages are mapped in the child process
- While mapping the write bit of the process is unset
- The PTE_C bit is set to provide identification when handling page faults
- The reference to those pages is also incremented since now two processes (parent and child) will be pointing the page

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  . . .
    pa = PTE2PA(*pte);
    flags = (PTE_FLAGS(*pte) | PTE_C) & (~PTE_W);
    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      goto err;
    }
    *pte = PA2PTE(pa) | flags;
    incref((void *)pa);
  }
  return 0;
  . . .
}
```

3 HANDLING PAGE FAULT

Modify the usertrap to handle page load faults. Whenever the **r_scause()** returns 15, this means that there is a page load fault. This might potentially be due to COW modification, so this should be handled if this is the case.

To identify it we will first do some primary checks and then call **pgfault_handler** this will be the handling function.

```
. . .
        else if(r_scause() == 13 || r_scause() == 15){
    uint64 va = r_stval();
    if(va >= MAXVA) {
      p->killed = 1;
    } else if(va >= p->sz) {
      p->killed = 1;
    } else if(va < p->trapframe->sp && va >= PGROUNDDOWN(p->trapframe->sp) - PGSIZE) {
      p->killed = 1;
    } else if(pgfault_handler(p->pagetable, va) != 0) {
      p->killed = 1;
    }
}
. . .
```

4 KALLOC.C

- First, add an array in the kmem structure
- The length of this array is PHYSTOP/PGSIZE. Essentially this is the number of pages. So, each element in this array has a number correponding to each page
- This number is the number of references the page has.
- Initially in **kinit** all the elements in this array are set to zero
- Later on the elements are incremented and decremented accordingly
- **kfree** is also modified so that it only actually frees the page when there are no references to the page

```
struct {
  struct spinlock lock;
  struct run *freelist;
  int count[PHYSTOP/PGSIZE];  // ref cnts
} kmem;


void
kfree(void *pa)
{
```

```
. . .
  acquire(&kmem.lock);
  if((--kmem.count[REFINDEX(pa)]) <= 0) {
    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);
    r = (struct run*)pa;
    r->next = kmem.freelist;
    kmem.freelist = r;
  }
  release(&kmem.lock);
}
```

5 PGFAULT_HANDLER

- This is the handler executed when one of the programme wants to write to the pages that were shared by multiple processes.
- The arguemnts to this function are the pagetable of the process and the virtual address where the fault occured
- First, the **walk** function is invloked to get the phusical address corresponding to the virtrual address
- The entries are checked to see if the page is valid and is user has access to it
- Then check if reference count is 1, this means that the page is pointed only by this process, in this case there is no need to copy the page. The PTE_C bit is unset and also write access is given.
- If this is not the case, then a new page is allocated and the contents are copied.
- PTE_C bit is unset and write bit is set for the new pagecreated.

```
int pgfault_handler(pagetable_t pagetable, uint64 va) {
if(va >= MAXVA) return -1;

pte_t *pte = walk(pagetable, va, 0);
if(pte == 0)
  return -1;
if((*pte & PTE_V) == 0)
  return -1;
if((*pte & PTE_U) == 0)
  return -1;

// check if refcnt is 1. unset cow page if refcnt = 1, alloc a new page otherwise
uint64 mem, pa;
```

```
  pa = (uint64)PTE2PA(*pte);
  acquire(&kmem.lock);
  if(kmem.count[REFINDEX(pa)] == 1) {
    *pte &= ~PTE_C;
    *pte |= PTE_W;
    release(&kmem.lock);
    return 0;
  }
  release(&kmem.lock);

  if((mem = (uint64)kalloc()) == 0) return -1;
  memmove((void *)mem, (void *)pa, PGSIZE); // copy contents
  kfree((void *)pa); // decrease reference, and free pa if necessary

  // modify mappings and unset cow page
  *pte = PA2PTE(mem) | PTE_FLAGS(*pte);
  *pte &= ~PTE_C; // unset PTE_C
  *pte |= PTE_W;  // set PTE_W

  return 0;
}
```