# Lab 6: Copy-on-Write Fork in xv6
# Report

Tejesh Raut 140050008
Gangesh Gudmalwar 140050058

October 2, 2016

**To run the program:** Clone the git repo of xv6 and then patch our changes into it by entering the following commands:

```
git clone https://github.com/guilleiguaran/xv6.git
patch -p0 -i file.patch
cd xv6
make
make qemu
```

Then to run testcase enter following command in qemu:

```
testcow
```

**Project Report:**

## 1 Adding getNumFreePages system call

**Add the variable numFreePages to implement the given system call in kalloc.c :**
In kalloc.c add the entry in struct kmem:

```
 int numFreePages;
```

**Adding given system calls**
Add following code in defs.h:

```
int getNumFreePages(void);
```

Add following code in sysproc.c:

```
int
sys_getNumFreePages(void)
{
```

```
    return getNumFreePages();
}
```

Add the following declaration along with other system calls in syscall.c:

```
extern int sys_getNumFreePages(void);
```

Add following fields in the same file like other system calls in syscall.c:

```
[SYS_getNumFreePages] sys_getNumFreePages,
```

Add the following lines in usys.S:

```
SYSCALL(getNumFreePages)
```

Add following lines in syscall.h:

```
#define SYS_getNumFreePages 22
```

Add the function body of getNumFreePages in kalloc.c

```
// Returns the number of free pages.
int
getNumFreePages(void)
{
   if(kmem.use_lock)
      acquire(&kmem.lock);
   int r = kmem.numFreePages;
   if(kmem.use_lock)
      release(&kmem.lock);
   return (r);
}
```

Initialize numFreePages in kinit1

```
kmem.numFreePages = 0;
```

In kfree

```
kmem.numFreePages = kmem.numFreePages + 1; // A new node is added to
    freelist so increase the number of free pages
```

In kalloc

```
kmem.numFreePages = kmem.numFreePages - 1 ; // A node is popped out
    from the freelist so decrease the number of free pages.
```

## 2 Reinstalling of page table

Whenever the flags are changed in copyuvm function the page table must be reinstalled using:

```
lcr3(v2p(pgdir)); // reinstall the page table
```

## 3 Keeping track of the reference count of pages

In vm.c add the declaration of array and lock:

```
struct spinlock lock;
char pg_refcount[PHYSTOP >> PGSHIFT]; // array to store refcount
```

In inituvm function:

```
acquire(&lock);
pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >> PGSHIFT] +
    1 ;
release(&lock);
```

In allocuvm function:

```
acquire(&lock);
pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >> PGSHIFT] +
    1 ;
release(&lock);
```

In deallocuvm free the page only when no other page table is pointing it:

```
acquire(&lock);
if(--pg_refcount[pa >> PGSHIFT] == 0)// if no other page table is
    pointing to this page remove it
{
  char *v = p2v(pa);
  kfree(v);
}
release(&lock);
```

In copyuvm function when a process is forked the refcount of that permanent address should be incremented:

```
 acquire(&lock);
 pg_refcount[pa >> PGSHIFT] = pg_refcount[pa >> PGSHIFT] + 1; //
     increase reference count of that permanent page.
 release(&lock);
```

# 4 Change of copyuvm function

Make the pagetable unwritable and then assign the same permanent addresses to the new page table

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  //char *mem; //No need to allocate new memory

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    *pte &= ~PTE_W; // make this page table unwritable
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    // No need of page allocation
    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) // map the
        child's page table to same permanent addresses
      goto bad;
    acquire(&lock);
    pg_refcount[pa >> PGSHIFT] = pg_refcount[pa >> PGSHIFT] + 1; //
        increase reference count of that permanent page.
    release(&lock);
  }
  lcr3(v2p(pgdir)); // reinstall the page table
  return d;

  bad:
  freevm(d);
  lcr3(v2p(pgdir)); // reinstall the page table
  return 0;
}
```

# 5 Adding trap handler to handle pagefaults

**To show error on page fault** In defs.h

```
void       pagefault(uint err_code);
```

In trap.c:

```
case T_PGFLT:
   pagefault(tf->err);
   break;
```

In vm.c

```
void pagefault(uint err_code)
{
   cprintf("Pagefault occured");
   return;
}
```

# 6 Adding trap handling function to make copy of user memory

In vm.c

```
void pagefault(uint err_code)
{
   uint va = rcr2();
   uint pa;
   pte_t *pte;
   char *mem;
   if(va >= KERNBASE)
   {
      cprintf("pid %d %s: Illegal memory access on CPU %d due to
          virtual address 0x%x is mapped to kernel code. So killing
          the process\n", proc->pid, proc->name, cpu->id, va);
      proc->killed = 1;
      return;
   }
   if((pte = walkpgdir(proc->pgdir, (void*)va, 0))==0)
   {
      cprintf("pid %d %s: Illegal memory access on CPU %d due to
          virtual address 0x%x is mapped to NULL pte. So killing the
          process\n", proc->pid, proc->name, cpu->id, va);
      proc->killed = 1;
      return;
```

```
}
if(!(*pte & PTE_P))
{
   cprintf("pid %d %s: Illegal memory access on CPU %d due to
       virtual address 0x%x is mapped to pte which is not present.
       So killing the process\n", proc->pid, proc->name, cpu->id,
       va);
   proc->killed = 1;
   return;
}
if(!(*pte & PTE_U))
{
   cprintf("pid %d %s: Illegal memory access on CPU %d due to
       virtual address 0x%x is mapped to pte which is not
       accessible to user. So killing the process\n", proc->pid,
       proc->name, cpu->id, va);
   proc->killed = 1;
   return;
}
if(*pte & PTE_W)
{
   panic("Unknown page fault due to a writable pte");
}
else
{
   pa = PTE_ADDR(*pte);
   acquire(&lock);
   if(pg_refcount[pa >> PGSHIFT] == 1)
   {
      release(&lock);
      *pte |= PTE_W;
   }
   else
   {
      if(pg_refcount[pa >> PGSHIFT] > 1)
      {
         release(&lock);
         if((mem = kalloc()) == 0)
         {
            cprintf("pid %d %s: Pagefault due to out of memory",
                proc->pid, proc->name);
            proc->killed = 1;
            return;
         }
         memmove(mem, (char*)p2v(pa), PGSIZE);
         acquire(&lock);
         pg_refcount[pa >> PGSHIFT] = pg_refcount[pa >> PGSHIFT] -
             1;
         pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >>
             PGSHIFT] + 1;
```

```
            release(&lock);
            *pte = v2p(mem) | PTE_P | PTE_W | PTE_U;
         }
         else
         {
            release(&lock);
            panic("Pagefault due to wrong ref count");
         }
      }
      lcr3(v2p(proc->pgdir));
   }
}
```

# 7 Test case testcow

```
#include "types.h"
#include "stat.h"
#include "user.h"

int i = 3;
int main(void)
{
   int pid;
   pid = fork();
   if(pid == 0)
   {
      printf(1,"Number of free pages in child 1 before changing
          variable are: %d\n", getNumFreePages());
      i = 4;
      printf(1,"Number of free pages in child 1 after changing
          variable due to copy are: %d\n", getNumFreePages());
   }
   else
   {
      wait();
      pid = fork();
      if(pid == 0)
      {
         printf(1,"Number of free pages in child 2 before changing
             variable are: %d\n", getNumFreePages());
         i = 4;
         printf(1,"Number of free pages in child 2 after changing
             variable due to copy are: %d\n", getNumFreePages());
      }
      else
      {
```

```
        printf(1,"Number of free pages in parent are: %d\n",
            getNumFreePages());
        wait();
      }
    }
    exit();
}
```

**Output of above test case is :**

```
$ testcow
Number of free pages in child 1 before changing variable are: 56710
Number of free pages in child 1 after changing variable due to copy
    are: 56709
Number of free pages in parent are: 56710
Number of free pages in child 2 before changing variable are: 56710
Number of free pages in child 2 after changing variable due to copy
    are: 56709
```