Programming Assignment 3 Part B

Binghan Geng

A20482350

bgeng1@hawk.iit.edu

Yu Li

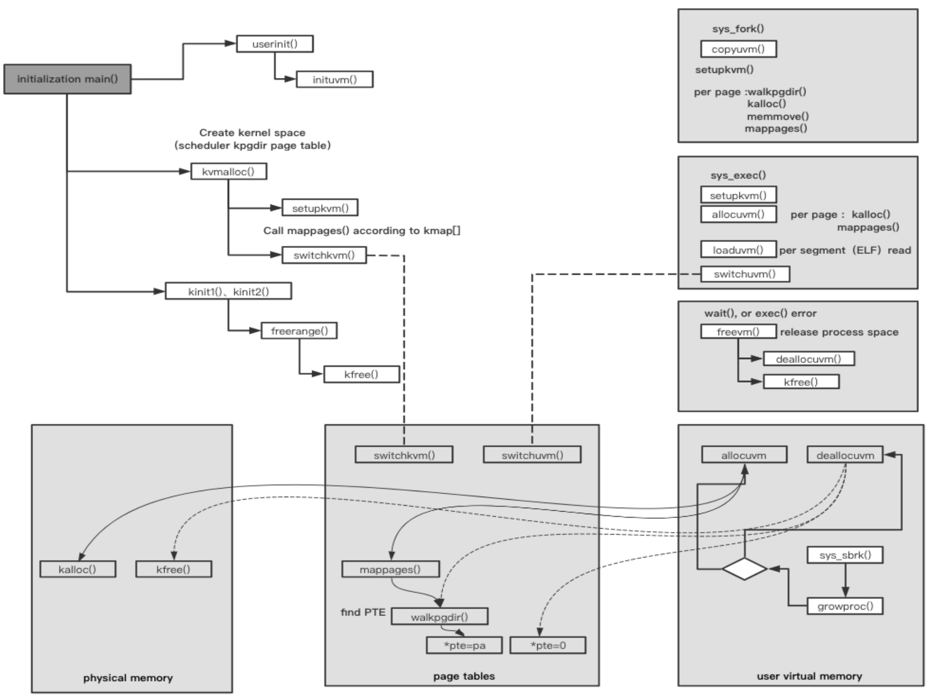
A20496405

yli385@hawk.iit.edu

1. System calls to share a memory page

1.1 Overview：xv6 memory management

XV6 memory management is divided into two stages: initialization and runtime.It involves physical page management, virtual memory space management, virtual and real mapping page tables. On top of these three operations, xv6 implements memory management initialization operations and processing dynamic runtime operations.The following figure is the whole process of memory management organized according to xv6, and the call link of the code.



1.2 Implementation of xv6 memory sharing

1.2.1 The basic program structure

The simplified scheme of process mapping shared memory page is as shown in the figure below, process A, B, C realize data reading and writing through the mapping relationship between virtual address and physical address.

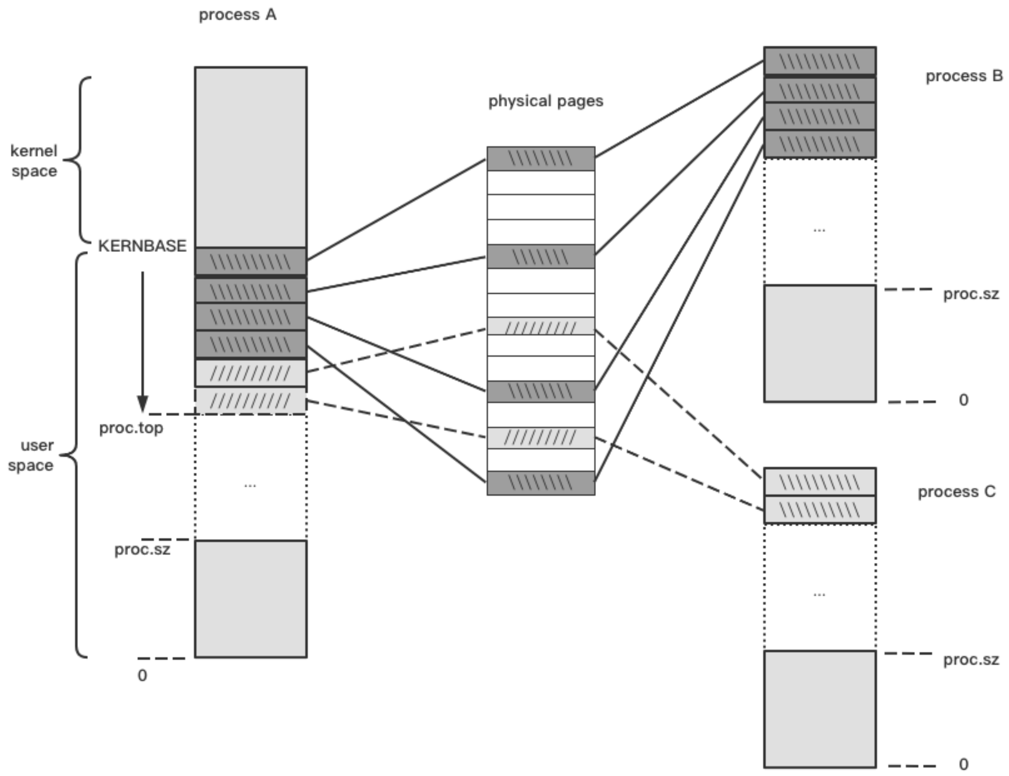


Figure: Program shared memory diagram

1.2.2 The basic data structure

The program have **A maximum of shared memory areas**, using array of record structure,each member is a "sharemempage" structure, the members inside include the reference count (the number of times the process is mapped), the physical page mapped in this interval (code 1-1. line 390-394).

A maximum of physical pages corresponding to the key, using array of integer "page_nums[NKEY]", cooperate with sharemempage(code 1-1. line 398).

A Lock "shmlock", using spinlock struct for exclusive alloc/dealloc memory(code 1-1. line 396).

```c
struct sharemempage
{
    int refcount;            //Reference count for each key
    void* physaddr[MAX_SHM_PGNUM];      //Corresponding to the physical address of per page
};

struct spinlock shmlock;                    //Locks for exclusive access
struct sharemempage shmarray[NKEY];         //Array of Share memory page record
int page_nums[NKEY];    // Number of physical pages corresponding to the key
```

code 1-1 vm.c

The program also defines 3 variables in proc structure (proc.h): **top,** is used to mark the upper limit of the user space of the process, originally KERNELBASE, but the shared content is allocated at the top of the user space, recording this upper limit can be used normally sbrk(). **shm_key[NKEY],** is a array, which shared memory is used at which position. **shm_va[NKEY],** virtual address space of used pages.

```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  uint top;                    //Current top of VA (the high end of the process's adddress space)
  uint shm_keys[NKEY];         //Keys that the process has called GetSharedPage
  void* shm_va[NKEY];          //Virtual address space of used pages

};
```

code 1-2 proc.h

1.2.3 Implementation details

Setp1: Add a new system call void * GetSharedPage(key, num_pages).

The parameter **key**, is used to specify which of the shared memory areas to obtain, **num_pages** represents the number of the shared memory pages. This system call returns the virtual address corresponding to the shared memory pages.

When the process calls GetSharedPage(key, num_pages), it is divided into two cases for processing according to whether the shared memory already exists:

1. If the shared memory area corresponding to shmarray[key] has not been created yet, initialize shmarray[key] according to the size, allocate the corresponding physical memory, establish a page table mapping, and return the virtual address of the memory.

2. If the shmarray[key] shared memory area already exists, the new page table entry pte is bound to the physical address of shmarray[key] pages, and the corresponding virtual address is returned. At this time, the incoming size parameter is ignored.

Similarly, we also need to use a series of functions such as the initialization operation function shminit() of the shared memory area. the newly added function is now in the vm.c file and need to be declared in defs.h so that other codes can call them. As shown in code 2-1.

```
void            shminit();
void*           GetSharedPage(uint key, uint num_pages);
int             FreeSharedPage(uint key);
```

code 2-1 defs.h

Setp2: Initialize

Add a shminit method in vm.c to initialize the shared variable array shmarray[].

```
//initialize the shared variable array shmarray[].
void
shminit()
{
  initlock(&shmlock,"shmplock");    //Initializing locks

  for (int i = 0; i < NKEY; i++)      //Initialize shmarray
  {
    shmarray[i].refcount = 0;
    page_nums[i] = 0;
    for(j = 0; j < MAX_SHM_PGNUM; j++) {
      physaddr[j] = 0;
    }
  }

}
```

code 2-2 vm.c

Then call the initialization function in the main() method in main.c, so that the preparation of shared variables can be completed when the system starts.

```c
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    shminit();        // initialize the shared memory page
    userinit();       // first user process
    mpmain();         // finish this processor's setup
}
```

code 2-3 main.c

```c
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->top = KERNBASE;            //Initialize shm, at first shm overlaps with KERNBASE


  for (int i = 0; i < NKEY; i++) {
    p->shm_keys[i] = 0;
  }
```

code 2-4 proc.c

Setp3: Map shared memory to process space

Next, need to implement GetSharedPage(key, num_pages) that maps the shared memory pages area to the process space. As mentioned earlier (NEKY = 8), the parameter key is a subscript (0-7) of the shared memory, and num_pages is the number of pages to be allocated, and returns the address of the shared memory. The implementation needs to be divided into three situations to implement:

1. If the shared memory area has been mapped in the process space, the address is returned directly.

2. If the shared memory area has not been created, the reference count of the system shmarray[key] is still 0, we need to create the memory area (including allocating pages and establishing page table mapping).

The method allocshmex(), the working principle of allocuvm() is the same,the difference is that our memory usage is allocated from high addresses to low addresses,and method deallocshmex() is also the same.

```c
// This method is basically the same as the allcouvm implementation, the difference is
// that our memory usage is allocated from high addresses to low addresses.
int
allocshmex(pde_t *pgdir, uint oldva, uint newva, uint sz,void *phyaddr[MAX_SHM_PGNUM])
{
  char *mem;
  uint a;

  if(oldva > KERNBASE || newva < sz)
    return 0;
  a = newva;
  for (int i = 0; a < oldva; a+=PGSIZE, i++)
  {
    mem = kalloc();         //Assigning physical page frames
    if(mem == 0){
      // cprintf("allocshm out of memory\n");
      deallocshmex(pgdir,newva,oldva);
      return 0;
    }
    memset(mem,0,PGSIZE);
    mappages(pgdir,(char*)a,PGSIZE, pa: (uint)V2P(mem), perm: PTE_W|PTE_U);   //页表映射
    phyaddr[i] = (void *)V2P(mem);
    // cprintf("allocshm : %x\n",a);
  }
  return newva;
}
```

code 2-5 vm.c

```c
// This method is basically the same as the deallocuvm implementation,the difference is
// that our memory usage is dealloced from low addresses to high addresses.
int
deallocshmex(pde_t *pgdir, uint oldva, uint newva)
{
  pte_t *pte;
  uint a, pa;
  if(newva <= oldva)
    return oldva;
  a = (uint)PGROUNDDOWN( a: newva - PGSIZE); //
  for (; oldva <= a; a-=PGSIZE)
  {
    pte = walkpgdir(pgdir,(char*)a, alloc: 0);
    if(pte && (*pte & PTE_P)!=0){
      pa = PTE_ADDR( pte: *pte);
      if(pa == 0){
        panic("kfree");
      }
      *pte = 0;
    }
  }
  return newva;
}
```

code 2-6 vm.c

3. If the system already has corresponding shared memory, just map it to the process space.

```c
void *
GetSharedPage(uint key, uint num_pages) {
  void *phyaddr[MAX_SHM_PGNUM];
  uint addr = 0;
  if (key < 0 || NKEY <= key || num_pages < 0 || MAX_SHM_PGNUM < num_pages)    //Calibration parameters
    return (void *) -1;
  addr = proc->top;
  // If the current process has already mapped the shared memory of the key, return the address directly.
  // Do not support multiple mappings of the same shared memory area in the same process.
  if (proc->shm_keys[key] == 1) {
    return proc->shm_va[key];
  }
  acquire(&shmlock);
  // If the system has not yet created the shared memory corresponding to this key, then allocate the memory and map.
  if (shmarray[key].refcount == 0) {
    addr = allocshmex(proc->pgdir, addr, newva: addr - num_pages * PGSIZE, proc->sz, phyaddr);
    //The new allocshmex() allocates memory and maps it, the same principle as allcouvm()
    if (addr == 0) {
      release(&shmlock);
      return (void *) -1;
    }
    proc->shm_va[key] = (void *) addr;
    shmadd(key, num_pages, phyaddr);    //Fill the new memory area information into the shmarray[8] array
  } else {
    //If the key is not held and the shared memory corresponding to this key is already allocated in the system,
    // then it is mapped directly.
    for (int i = 0; i < num_pages; i++) {
      phyaddr[i] = shmarray[key].physaddr[i];
    }
    num_pages = page_nums[key];
    //The mapshmex method creates a new mapping
    if ((addr = mapshmex(proc->pgdir, addr, newva: addr - num_pages * PGSIZE, proc->sz, phyaddr)) == 0) {
      release(&shmlock);
      return (void *) -1;
    }
    proc->shm_va[key] = (void *) addr;
    shmarray[key].refcount++;            //Reference Count +1
  }
  proc->top = addr;
  proc->shm_keys[key] = 1;
  release(&shmlock);
  return (void *) addr;
}
```

GetSharedPage

code 2-7 vm.c

Setp4: Remove/Unmap the shared memory area: int FreeSharedPage(uint key).

FreeSharedPage(key) is used to release / unmap the shared memory pages, and is also called to release the physical memory to completely destroy the Shared memory. If the refcount of a shared memory area becomes 0, it must be called to release the physical memory to completely destroy the shared memory. And because the key addition is disorderly, if you follow the virtual address released by the key, the calculation is very complicated. So when the memory map of the current process (shm_keys[]) is all zero, all are released at one time.

```c
FreeSharedPage(uint key) {
  if (key < 0 || NKEY <= key) {
    cprintf("key error\n");
    return -1;
  }
  if (proc->shm_keys[key] != 1) {
    return -1;
  }
  acquire(&shmlock);
  proc->shm_keys[key] = 0;

  cprintf("FreeSharedPage: key is %d\n", key);
  struct sharemempage *shmem = &shmarray[key];
  cprintf("FreeSharedPage: refcount is %d\n",shmem->refcount);
  shmem->refcount--;
  cprintf("FreeSharedPage: page_nums is %d\n", page_nums[key]);
  if (shmem->refcount == 0) {
    for (int i = 0; i < page_nums[key]; i++) {
      kfree((char *) P2V( a: shmem->physaddr[i]));        //Recycle physical memory, page by page, frame by frame
    }
  }
  //If the current key without proc are 0, then free the space
  int flag = 0;
  for (int i = 0; i < NKEY; i++) {
    if (proc->shm_keys[key] == 1) {
      flag++;
      break;
    }
  }
  if (flag == 0) {
    // Free the user space memory, because the add is unordered, the calculation of the free virtual
    // address is very complicated, only a one-time free
    deallocshmex(proc->pgdir, proc->top, KERNBASE);
    cprintf("Free the user space memory.\n");
  }
  release(&shmlock);
  return 0;
}
```

code 2-8 vm.c