CS4375-13948 Fall 2023
Ruben Carmona
rcarmoname@miners.utep.edu

Homework Report
Submitted on Nov 9, 2023

HW 4: Lazy Allocation for xv6

https://github.com/Rcarmonam/myxv6RubenCarmona.git

Task 1. freepmem() system call

```
uint64
sys_freepmem(void){
  int count = kfreepagecount();
  return count*PGSIZE;
}

uint64
sys_memoryuser(void){
  int count = kfreepagecount();
  return count*PGSIZE;
}
```

```
[SYS_freepmem]   sys_freepmem,
[SYS_memoryuser] sys_memoryuser,
```

```
extern uint64 sys_freepmem(void);
extern uint64 sys_memoryuser(void);
```

```
#define SYS_freepmem 23
#define SYS_memoryuser 24
```

```
entry("freepmem");
entry("memoryuser");
```

```
//Adding declaration
int freepmem(void);
int memoryuser(void);
```

```
hart 2 starting
hart 1 starting
init: starting sh
$ free -k
130264
$ free
133390336
$ free -m
127
$ memoryuser 1 4 1 &
$ allocating 0x0000000000000001 mebibytes
malloc returned 0x0000000000003010
free
132300800
$ freeing 0x0000000000000001 mebibytes
free
132300800
$ allocating 0x0000000000000002 mebibytes
malloc returned 0x0000000000103020
free
130199552
$ freeing 0x0000000000000002 mebibytes
free
130199552
$ allocating 0x0000000000000003 mebibytes
malloc returned 0x0000000000003020
free
130199552
$ freeing 0x0000000000000003 mebibytes
free
130199552
$ allocating 0x0000000000000004 mebibytes
malloc returned 0x0000000000303030
free
125997056
$ freeing 0x0000000000000004 mebibytes
free
125997056
```

Above we can find the modified code for the task 1 to be successful. We first had to add the declarations on the correct files for our program compile. We can also find the implementations for freepmem() and memoryuser(), both functions calculate the total free physical memory in the system by calling kfreepagecount() function which presumably returns the number of free pages in the system. By implementing this code, the expected output was successful.

Task 2. Change sbrk() so that it does not allocate physical memory.

```
uint64
sys_sbrk(void)
{
  int n;
  if(argint(0, &n) < 0)
    return -1;
  struct proc *curproc = myproc();
  // Calculate new heap size
  uint64 new_sz = curproc->sz + n;
  // Ensure new_sz is within the process's address space limits
  if (new_sz < curproc->sz || new_sz >= TRAPFRAME) {
    return -1;
  }
  // Update the process's heap size without allocating physical memory
  curproc->sz = new_sz;
  // Return the old end of the heap (before growing)
  return curproc->sz - n;
}
```

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ free -k
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000012c0 stval=0x0000000000004008
panic: uvmunmap: not mapped
```

Above we can find the implementation for sys_sbrk() method where it takes an integer argument n representing the number of bytes to increase the heap size. It first checks if ‘n’ is a valid integer using ‘argint’ function. The error ‘panic: uvunmap: not mapped’ states that there is an issue with unmapping memory regions, meaning that this will get fix on the next task.

Task 3. Handle the load and store faults that result from Task 2

Below we can observe usertrap() method located in trap.c

```
usertrap(void)
{
  int which_dev = 0;
  uint64 addr;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
    //ADDING CHANGES
  } else if(r_scause() == 0xf || r_scause() == 13){

    addr = r_stval();
    if(addr < p -> sz){
      // Allocate a physical memory frame and install the page table mapping
      char *mem = kalloc();
      if (mem == 0) {
        // Handle allocation failure (panic, exit, etc.)
        printf("Out of memory\n");
        p->killed = 1;
      } else {
        // Clear the allocated physical memory
        memset(mem, 0, PGSIZE);
```

```
        // Install the page table mapping for the faulting address
        if((mappages(p->pagetable, addr, PGSIZE, (uint64)mem, PTE_W | PTE_X | PTE_R) != 0)){
          // Handle mapping failure (panic, exit, etc.)
          printf("Failed to map memory\n");
          kfree(mem); // Free the allocated physical memory
          p->killed = 1;
        }
      }
    }else{
      // Invalid access, handle it as appropriate (panic, exit, etc.)
      printf("Invalid memory access at address %p\n", addr);
      p->killed = 1;
    }
  }else{
    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;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

Ruben Carmona

```
hart 2 starting
hart 1 starting
init: starting sh
$ free -k
panic: uvmunmap: not mapped
```

In this task we had to work with trap.c modifying usertrap() that check for various exceptions, including system calls, device interrupts, load, store page defaults, and unexpected exceptions. When a load or storage page faults occurs and the faulting address is within the process's allocated virtual memory, it attempts to allocate a physical memory frame and install the necessary page table mapping. If some reason the mapping fails, it marks the process as killed. In this case there is a panic error indicating an issue with unmapping a virtual address that was not mapped, suggesting a problem with the memory mapping logic in the code. In this specific task I had difficulties trying to figure out the issue with the mapping task, but after carefully reviewing the code I was able to fix the usertrap() issue from task 2 and in the following task which is 4 I was able to fix the mapping panic error.

Task 4. Fix kernel panic and any other errors.
trap.c, vm.c were the files that need to be fixed to accomplish this task.

```
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  if(size == 0)
    panic("mappages: size");

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      continue;
      //panic("mappages: remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

```
usertrap(void)
{
  int which_dev = 0;
  uint64 addr;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
    //ADDING CHANGES
  } else if(r_scause() == 0xf || r_scause() == 13){

    addr = r_stval();
    if(addr < p -> sz){
      // Allocate a physical memory frame and install the page table mapping
      char *mem = kalloc();
      if (mem == 0) {
        // Handle allocation failure (panic, exit, etc.)
        printf("Out of memory\n");
        p->killed = 1;
      } else {
        // Clear the allocated physical memory
        memset(mem, 0, PGSIZE);
        // Install the page table mapping for the faulting address
        if(mappages(p->pagetable, PGROUNDDOWN(addr), PGSIZE, (uint64)mem, PTE_W | PTE_X | PTE_R | PTE_U) != 0){
          // Handle mapping failure (panic, exit, etc.)
          printf("Failed to map memory\n");
          kfree(mem); // Free the allocated physical memory
          p->killed = 1;
        }
      }
    }else{
      // Invalid access, handle it as appropriate (panic, exit, etc.)
      printf("Invalid memory access at address %p\n", addr);
      p->killed = 1;
    }
  }else{
    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;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

```c
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;

  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
      continue;
      //panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

```c
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  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)
      continue;
      //panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

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

In this task we had to modify some of the code and add some ‘permissions’ in this case |PTE_U as part of our mappages function. In the trap.c file in the usertrap() method we modify an if statement as the following ‘if (mappages(p->pagetable, PGROUNDDOWN(addr), PGSIZE, (uint64)mem, PTE_W | PTE_X | PTE_R | PTE_U) != 0) {‘ and also check for the following case ‘else if(r_scause() == 0xf || r_scause() ==13){‘ allowing us to fix the issues and be able to test the lazy memory allocation.

Task 5. Test your lazy memory allocation.

```
hart 1 starting
hart 2 starting
init: starting sh
$ free -k
130264
$ free
133390336
$ free -m
127
$ memoryuser 1 4 1 &
$ allocating 0x0000000000000001 mebibytes
malloc returned 0x0000000000003010
free
133349376
$ freeing 0x0000000000000001 mebibytes
free
133349376
$ allocating 0x0000000000000002 mebibytes
malloc returned 0x0000000000103020
free
133345280
$ freeing 0x0000000000000002 mebibytes
free
133345280
$ allocating 0x0000000000000003 mebibytes
malloc returned 0x0000000000003020
free
133345280
$ freeing 0x0000000000000003 mebibytes
free
133345280
$ allocating 0x0000000000000004 mebibytes
malloc returned 0x0000000000303030
free
133337088
$ freeing 0x0000000000000004 mebibytes
free
133337088
```

Above we can see the testing for task 5, the ‘free’ command is used to check the system’s free memory in different units (kilobytes and megabytes). ‘memoryuser()’ 1 4 1 &’ indicates a program to try to allocate 4 megabytes of memory, ‘malloc’ allocates a block of memory and returns the pointer to the allocated memory.

Summary:

For this homework, I successfully implemented various modifications to the xv6 operating system. I began by adding declarations for functions such as ‘freepmem()’ and ‘memoryuser()’, enabling successful compilation. The implementations of these functions involved calculating the total free physical memory in the system using ‘kfreepagecount()’. I modified ‘sys_sbrk()’, troubleshooting and fixing issues in ‘usertrap()’, testing lazy memory allocation, and enabling the use of the entire virtual address space. Challenges were overcome through careful code review, resulting in the comprehensive ser of modifications to enhance memory management in xv6.