Kinjal Mugatwala (SID: 862014507)
kmuga001
March 6, 2020

Lab 3 Report

Summary:
In this lab, we are trying to move our stack upward in a particular way that will ultimately allow us to grow the stack more. To do that, I needed to change the top of our stack and the end of the stack. The top of the stack is right under KERNBASE. I fixed allocuvm() to allocate pages to help give room for the stack to grow and allocate the pages right under KERNBASE. I then adjusted the stack pointer to NEWKERNBASE, which is the address right under KERNBASE and that will represent the top word in the stack page. To take account of our new stack positioning, I needed to adjust the iteration loop in copyuvm() to take account of the stack and copy over the right pages. Finally, we needed to add a case to handle the page fault trap and try to make sure that error gets handled properly while using rcr2() as that function gives us the address that caused the page fault.

Lab 3 Output:

```
$ lab3
Usage: lab3 levels
$ lab3 100
Lab 3: Recursing 100 levels
Lab 3: Yielded a value of 5050
$ lab3 1000
Lab 3: Recursing 1000 levels
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 2
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 3
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 4
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 5
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 6
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 7
case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: 8
Lab 3: Yielded a value of 500500
$
```

exec.c:
In this file, I adjusted the allocuvm() arguments to make sure that we are allocating the right pages in the correct location. The variable user_top is initialized as such below because PGSIZE is the size of the page and so to get the top value (right under KERNBASE) we need to start right at the top of the page under KERNBASE. To make sure our stack starts right under KERNBASE, the stack pointer or sp is set to NEWKERNBASE (which is the address right below KERNBASE). The first page I am mapping is to the user_top as that is the top of the stack and the end page I am mapping is to is user_top + 8 to give more space for the stack. I also set the stacksize variable to 1 as instructed in the todo 4 survival guide.

```
  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  uint user_top = KERNBASE - PGSIZE;  //add for lab 3
  if((sp = allocuvm(pgdir, user_top, user_top + 8)) == 0) //added for lab 3
    goto bad;
  //clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = NEWKERNBASE;    //changed for lab 3 (original: sp = sz), addr of top word of stack page, under KERNBASE
```

...

```
  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->stacksize = 1; //mentioned in lab 3, todo 4
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;
```

syscall.c:

To make sure our code is consistent with our new stack positioning and trying to change sz, I switched every time sz was used to NEWKERNBASE as that is the value I am using now.

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
  //struct proc *curproc = myproc();

  if(addr >= NEWKERNBASE || addr+4 > NEWKERNBASE) //put in newkernbase for top word in lab 3
    return -1;
  *ip = *(int*)(addr);
  return 0;
}

// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;
  //struct proc *curproc = myproc();

  if(addr >= NEWKERNBASE) //added newkernbase for top word in lab 3
    return -1;
  *pp = (char*)addr;
  ep = (char*)NEWKERNBASE; //added for top word in lab 3
  for(s = *pp; s < ep; s++){
    if(*s == 0)
      return s - *pp;
  }
  return -1;
}
```

...

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;
 // struct proc *curproc = myproc();

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= NEWKERNBASE || (uint)i+size > NEWKERNBASE) //added for top word in lab 3
    return -1;
  *pp = (char*)i;
  return 0;
}
```

proc.h:

I added the stacksize value to keep track of the size of the entire stack.

```
// 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 stacksize;              // Tracks size of stack (bytes)
};
```

vm.c:

I added another for loop to copy the pages over from the user stack properly and to keep track of the stack size. The only difference between the first for loop and the for loop that I added is the for loop's iteration style. We want to start iterating from the top of the user stack and stop when our stacksize is not greater than 0 as that indicates that the size has diminished and that there is no more room in the stack and so we need to stop copying. Every time we go through an iteration, we decrease the page size and the stacksize as it conveys that we have copied a page from the stack.

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz, uint stacksize)
{
  //struct proc *curproc = myproc();
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  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");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
      goto bad;
  }
  //lab 3 - iteration over stack pages & tracking stack size
  //use PGROUNDUP NEWKERNBASE - PGSIZE + 1
  for(i = PGROUNDUP(NEWKERNBASE - PGSIZE); stacksize > 0; i -= PGSIZE, stacksize--){ //may need to change, unsure
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) //this part is the same from the above for loop
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
      goto bad;
  }

  return d;

bad:
  freevm(d);
  return 0;
}
```

trap.c:

I added a case for the trap page fault to make sure that page fault gets handled correctly and properly. rcr2() conveys the address that caused the page fault to occur. So, we need to error handled when the address from rcr2() is out of bounds, then we want to check if the allocation of the page is within the range of the stack. If it isn't, we need to send an error print message. We then need to increment the stacksize as this means that we need to grow our stack.

```
  case T_PGFLT:;
    if(((rcr2()) > NEWKERNBASE - (PGSIZE * (myproc()->stacksize + 1) + 1))){
        uint user_top = KERNBASE - (PGSIZE * (myproc()->stacksize + 1));
        if(allocuvm(myproc()->pgdir, user_top, user_top + 8) == 0){
                cprintf("case T_PGFLT from trap.c: allocuvm failed. Number of current allocated pages: %d\n", myproc()->stacksize);
                exit();
        }

        myproc()->stacksize += 1;
        cprintf("case T_PGFLT from trap.c: allocuvm succeeded. Number of pages allocated: %d\n", myproc()->stacksize);
        break;
    }
    //break;
```

proc.c:

Because I added a parameter to the copyuvm() function, I needed to add that stacksize value in the third argument. I also updated np's stack size by updating it with curproc's stack size.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz, curproc->stacksize)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  np->stacksize = curproc->stacksize;
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;
```

defs.h:

Because I changed the function definition of copyuvm() in vm.c, I needed to make sure the function is properly defined in defs.h.

```
pde_t*          copyuvm(pde_t*, uint, uint);
```

memlayout.h:

I defined a NEWKERNBASE value that will be right under KERNBASE and this will represent the top of the stack.

```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address

//LAB 3 change --> right under Kernbase
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
#define NEWKERNBASE (KERNBASE-1)

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
~
```