Lab 5: mmap - Alejandro Vargas (avarg116)

Virtual memory maps a program's address space onto physical memory. Virtual memory areas (VMA's) keep track of memory regions in an address space from its start to end, determining permissions and behaviors. To find a VMA we use a fixed size array in the mmap field. The mmap and munmap UNIX system calls create and delete mappings in the virtual address space. SHARED handles a memory mapped file by making updates to the mapping visible to other processes while PRIVATE does not. In this lab, we implement mmap and munmap functionality to xv6. We use lazy page allocation for better performance and in case mmap is a large file.

Implementation step 0: Add flags/definitions for the syscalls

I added mmap and munmap system calls. The definitions in fcntl.h specify read and write permissions for the memory mapped area. PRIVATE and SHARED are flags that indicate if updates should be visible. The values are for checking using bitwise AND and combining with OR.

Implementation: defining and allocating structure vma

```
struct vma_struct{
  uint64 addr; //start
  uint64 end;
  int prot; //permissons (read or write)
  int flags; //behaviors
  struct file *file; //file object
  int offset; //always 0
  int valid_vma; //check vma used
};
```

```
struct vma_struct vma_table[16]; //vma regions
```

The VMA structure in proc.h includes the start, end, permissions, behaviors, file object, and offset. I added a table in struct proc of size 16 of all the VMAs for a process.

Implementation: mmap()

```
uint64 sys_mmap() {
  struct proc *p= myproc();
  struct file *file;
  int length, prot, flags, fd, offset, pos;

  //check permissions, flags.
  if(argint(1, &length) < 0 || argint(2, &prot) < 0 || argint(3, &flags) < 0 ||
    argfd(4, &fd, &file) < 0 || argint(5, &offset) < 0)
    return -1;
  if((prot & PROT_WRITE) && (!file->writable) && (flags == MAP_SHARED))
    return -1;

  pos= 0;
  while(pos < 16 && p->vma_table[pos].valid_vma != 0)
      pos++;

  if(pos != 16) { //allocate new vma at unused region
    int start = p->sz;
    p->sz = start + length;
    //update vma
    p->vma_table[pos].addr = start;
    p->vma_table[pos].end = start + length;
    p->vma_table[pos].prot = prot;
    p->vma_table[pos].flags = flags;
    p->vma_table[pos].file = file;
    p->vma_table[pos].offset = 0;
    p->vma_table[pos].valid_vma = 1;
    filedup(file); //incr ref count
    return start; //pointer to mapped area
  } else {
    return -1;
  }
}
```

In the mmap function in sysfile.c, I first get the arguments length, protection and mapping flags, fd, and offset. The VMA contains a pointer *file to a struct file for the file being mapped. If the writing permission is checked and the mapping is SHARED, an error is returned. The while loop checks if a VMA table entry is in use. If an unused region is found in the VMA table, a VMA is allocated for the new mapping and updated. The reference count is increased with filedup so the structure doesn't disappear when the file is closed.

Implementation: usertrap()

```
int map_vma(pagetable_t pagetable, uint64 va)
{
    struct proc *p = myproc();
    struct vma_struct *vm = 0;
    for(int i= 0; i < 16; i++) { //find vma
      if(p->vma_table[i].valid_vma == 1 && va >= p->vma_table[i].addr && va < p->vma_table[i].end) {
        vm = &p->vma_table[i];
        break;
      }
    }
    if(!vm) { //no valid vma
      p->killed = 1;
      return -1;
    }

    char *mem= (char *)kalloc(); //allocate physical page
    memset(mem, 0, PGSIZE);
    begin_op(ROOTDEV);
    ilock(vm->file->ip);
    //transfer data from file to physical page
    if(readi(vm->file->ip, 0, (uint64)mem, va - vm->addr, PGSIZE) < 0) {
        iunlock(vm->file->ip);
        end_op(ROOTDEV);
        kfree(mem);
        p->killed = 1;
        return -1;
    }
    iunlock(vm->file->ip);
    end_op(ROOTDEV);

    uint64 flags= PTE_U; //set permissions
    if(vm->prot & PROT_READ)
        flags |= PTE_R;
    if(vm->prot & PROT_WRITE)
        flags |= PTE_W;
    //map physical map to user's virtual memory
    if(mappages(pagetable, va, PGSIZE, (uint64)mem, flags) != 0) {
        kfree(mem);
        p->killed = 1;
        return -1;
    }
    return 0;
}
```

Similar to lab 2, we implement lazy allocation to fill the page table for a mmap of a file larger than physical memory. First, r_scause checks whether a fault is a page fault. In usertrap, I ran into a ‘lock’ and ‘NDIRECT’ error so I call a helper function map_vma in proc.c. The loop finds the VMA for the faulty address. A physical page is allocated using kalloc. va - vm->addr calculates the offset which is the starting point in the file to map. The file is read with readi which takes an offset argument. Data is read from the file to the allocated physical memory. Finally, flags for the page are set and we map the physical map to user's virtual memory space using mappages.

Implementation: munmap() exit() and fork()

```
uint64 sys_munmap() {
  struct proc *p = myproc();
  uint64 start;
  int length, pos;

  if(argaddr(0, &start) < 0 || argint(1, &length) < 0)
    return -1;

  for(pos= 0; pos < 16; pos++) { //if VMA is used and in region
    struct vma_struct *vma = &p->vma_table[pos];
    if(vma->valid_vma == 1 && vma->addr <= start && start <= vma->end) {
      if(vma->flags == MAP_SHARED) { //write back the modifications
        begin_op(ROOTDEV);
        ilock(vma->file->ip);
        writei(vma->file->ip, 1, start, 0, length); //write data to inode (memory address)
        iunlock(vma->file->ip);
        end_op(ROOTDEV);
      }

      uvmunmap(p->pagetable, start, length, 0); //unmap specified pages
      if(vma->addr == start && vma->end == start + length) { //unmap whole region, start, or end.
        vma->valid_vma = 0;
        filedup(vma->file);
        fileclose(vma->file);
      } else if(vma->addr == start) {
        vma->addr = start + length;
      } else if(start + length == vma->end) {
        vma->end = start;
      }
      return 0;
    }
  }
  return -1;
}
```

munmap: the function finds the VMA for the region and unmaps the specified pages. First, we get the start and length arguments (memory region). The for loop iterates through the process's VMA table to find the specified area (start to start + length). If an unmapped page is modified and the file is SHARED, the page is written back to the file. Next, the pages are unmapped using uvmunmap. fileclose decrements the ref count when all pages of a previous mmap are removed. Finally, the VMA is updated for 3 unmapped cases: the whole vma, start, and end. It should not leave unmapped parts in the region.

```
for(int i = 0; i < 16; i++) { //fork modification
  if(p->vma_table[i].valid_vma == 1) {
    np->vma_table[i] = p->vma_table[i]; //child has same mapped regions as parent
    if(np->vma_table[i].file != 0) {
      filedup(np->vma_table[i].file); //ref to same file struct
    }
  }
}
```

fork: I modified fork by looping through the VMA table of the parent process and copying valid

entries to the child process. The reference count is incremented, ensuring they both reference the same vma file struct.

```
//exit similar to munmap
for(int i= 0; i < 16; i++) {
  if(p->vma_table[i].valid_vma == 1) {
    uint64 start= p->vma_table[i].addr;
    uint64 length= p->vma_table[i].end - start; //specified region
    uvmunmap(p->pagetable, start, length, 0); //unmap (deallocate) process's mapped regions
    if(p->vma_table[i].file != 0) {
      filedup(p->vma_table[i].file);
    }
  }
}
```

exit: the loop retrieves the start and length of a valid memory region and unmaps it using uvmunmap. This is similar to munmap.

uvmcopy and uvmunmap: similar to lab 2, I commented out panics and wrote if(a == last) break; a+= PGSIZE; continue; to avoid kernel crashes (remap panic).

Testing

```
running mmaptest:
$ make qemu-gdb
(5.7s)
  mmaptest: mmap_test: OK
  mmaptest: fork_test: OK
usertests:
$ make qemu-gdb
OK (42.8s)
Score: 100/100
[avarg116@sledge cs179f-fall23]$
```