MMAP

CS3500 Operating Systems Jul-Nov 2025

Sanjeev Subrahmaniyan S B

EE23B102

1 Objective

The objective of this lab is to add the mmap and munmap system calls to the xv6 operating system to support other virtual memory area operations. The lab is split into multiple tasks each of which is solved and its procedure and outputs shown.

2 Task 1 - Basic System Call Setup

In this part, the stubs of code required to make the system calls available and callable to the operating system are added to the appropriate files. Specifically, the changes made in different files are:

2.1 Makefile

The mmap test program is added as a UPROG entry in the makefile.

```
diff --git a/Makefile b/Makefile
index ee67e06..9865105 100644
--- a/Makefile
+++ b/Makefile
@@ -142,6 +142,7 @@ UPROGS=\
        $U/_logstress\
        $U/_forphan\
        $U/_dorphan\
+       $U/_mmaptest\
```

2.2 kernel/fcntl.h

The global defines required for the permission and shared primitives are added to this file.

```
diff --git a/kernel/fcntl.h b/kernel/fcntl.h
index 44861b9..27a1d30 100644
--- a/kernel/fcntl.h
+++ b/kernel/fcntl.h
@@ -3,3 +3,11 @@
 #define O_RDWR    0x002
 #define O_CREATE  0x200
 #define O_TRUNC   0x400
+
+#define PROT_NONE 0x0
+#define PROT_READ 0x1
+#define PROT_WRITE 0x2
+#define PROT_EXEC 0x4
+
+#define MAP_SHARED 0x01
+#define MAP_PRIVATE 0x02
```

2.3 kernel/syscall.c

The system calls are exposed in the syscall array to be called correctly.

```
diff --git a/kernel/syscall.c b/kernel/syscall.c
index 076d965..c46b85b 100644
--- a/kernel/syscall.c
+++ b/kernel/syscall.c
@@ -101,6 +101,8 @@ extern uint64 sys_unlink(void);
 extern uint64 sys_link(void);
 extern uint64 sys_mkdir(void);
 extern uint64 sys_close(void);
+extern uint64 sys_mmap(void);
+extern uint64 sys_munmap(void);

 // An array mapping syscall numbers from syscall.h
 // to the function that handles the system call.
@@ -126,6 +128,8 @@ static uint64 (*syscalls[])(void) = {
 [SYS_link]    sys_link,
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
+[SYS_mmap]    sys_mmap,
+[SYS_munmap]  sys_munmap,
 };
```

2.4 kernel/syscall.h

Index numbers are provided for both the system calls to be able to index into the function pointer array.

```
diff --git a/kernel/syscall.h b/kernel/syscall.h
index 3dd926d..7120171 100644
--- a/kernel/syscall.h
+++ b/kernel/syscall.h
@@ -20,3 +20,5 @@
 #define SYS_link   19
 #define SYS_mkdir  20
 #define SYS_close  21
+#define SYS_mmap   22
+#define SYS_munmap 23
```

2.5 kernel/sysfile.c

The code stubs returning a simple error code is written for both of the system calls.

```
diff --git a/kernel/sysfile.c b/kernel/sysfile.c
index d8234ce..964198a 100644
--- a/kernel/sysfile.c
+++ b/kernel/sysfile.c
@@ -503,3 +503,15 @@ sys_pipe(void)
   }
   return 0;
 }
+
+uint64
+sys_mmap(void)
+{
```

```
+  return -1;
+}
+
+uint64
+sys_munmap(void)
+{
+  return -1;
+}
```

2.6 user/user.h

The system call function prototypes are added here.

```
diff --git a/user/user.h b/user/user.h
index ac84de9..6ddca65 100644
--- a/user/user.h
+++ b/user/user.h
@@ -24,6 +24,9 @@ int getpid(void);
 char* sys_sbrk(int,int);
 int pause(int);
 int uptime(void);
+void *mmap(void *addr, size_t len, int prot, int flags,
+           int fd, off_t offset);
+int munmap(void *addr, size_t len);
```

2.7 user/usys.pl

Required to generate the assembly wrapper stub for each of the system calls.

```
diff --git a/user/usys.pl b/user/usys.pl
index c5d4c3a..545a5dd 100755
--- a/user/usys.pl
+++ b/user/usys.pl
@@ -42,3 +42,5 @@ entry("getpid");
 entry("sbrk");
 entry("pause");
 entry("uptime");
+entry("mmap");
+entry("munmap");
```

2.8 Execution output for task 1

```
hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
test basic mmap
mmaptest failure: mmap (1), pid=3
```

The output seen here matches the expected output as shown.

3 Task 2 - VMA Management

This part required adding an array of vma structs to the process control blocks and defining the struct itself. The changes done for this task are:

3.1 kernel/proc.h

Defined the vma struct and added the array to the process control block. The requirements for each of the fields is self explanatory.

```
diff --git a/kernel/proc.h b/kernel/proc.h
index d021857..f8cc26c 100644
--- a/kernel/proc.h
+++ b/kernel/proc.h
@@ -28,6 +28,18 @@ struct cpu {

 extern struct cpu cpus[NCPU];

+#define NVMA 16
+
+struct vma_t {
+  uint64 addr;       // start point of the address map
+  size_t size;      // size of the map in bytes
+  int prot;         // permissions for read or write or both
+  int flags;        // flags for SHARED, PRIVATE
+  struct file *file;           // file descriptor number
+  int used;         // marker to signify if the VMA slot is used
+  uint64 offset;
+};
+
 // per-process data for the trap handling code in trampoline.S.
 // sits in a page by itself just under the trampoline page in the
 // user page table. not specially mapped in the kernel page table.
@@ -91,7 +103,8 @@ struct proc {
   int killed;                  // If non-zero, have been killed
   int xstate;                  // Exit status to be returned to parent's wait
   int pid;                     // Process ID
-
+  struct vma_t vma[NVMA];    // VMA array for the process
   // wait_lock must be held when using this:
   struct proc *parent;         // Parent process
```

3.2 Execution and output

There is no change in the output for this task. Notably, I did not face any issues in implementing this section.

4 Task 3 - Implement MMAP

In this part of the assignemnt, the memory areas are chosen in which the VMA entries are allocated. The skeleton implementation of the mmap system call is also performed by intializing the vma structs during

process allocation. The mmap system call is implemented by copying the arguments, verifying validity and simply implementing the mapping. The changes are in the following files:

4.1 kernel/memlayout.h

```
diff --git a/kernel/memlayout.h b/kernel/memlayout.h
index 9bc9424..331259a 100644
--- a/kernel/memlayout.h
+++ b/kernel/memlayout.h
@@ -57,3 +57,5 @@
 //   TRAPFRAME (p->trapframe, used by the trampoline)
 //   TRAMPOLINE (the same page as in the kernel)
 #define TRAPFRAME (TRAMPOLINE - PGSIZE)
+#define MMAPBASE 0x40000000L
+#define MMAPTOP 0x80000000L
```

4.2 kernel/proc.c

```
    diff --git a/kernel/proc.c b/kernel/proc.c
index 9d6cf3f..d00b310 100644
--- a/kernel/proc.c
+++ b/kernel/proc.c
@@ -125,6 +125,17 @@ found:
   p->pid = allocpid();
   p->state = USED;

+  for(int i = 0; i < NVMA; i++){
+    p->vma[i].valid = 0;
+    p->vma[i].start = 0;
+    p->vma[i].end = 0;
+    p->vma[i].len = 0;
+    p->vma[i].flags = 0;
+    p->vma[i].offset = 0;
+    p->vma[i].fd = 0;
+  }
+  p->cur_vma_start = MMAPBASE; // the address from where mmap may begin
+
   // Allocate a trapframe page.
   if((p->trapframe = (struct trapframe *)kalloc()) == 0){
     freeproc(p);
```

4.3 kernel/sysfile.c

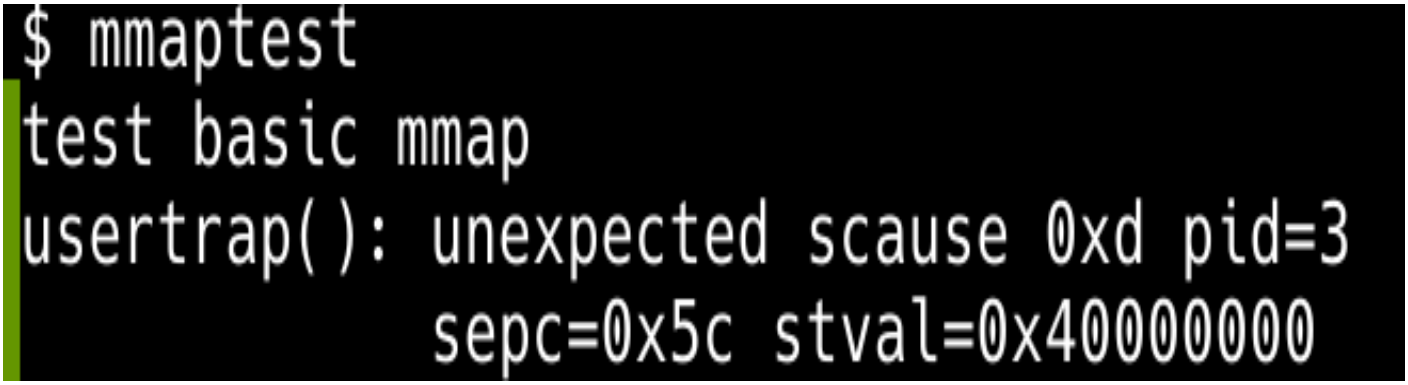
```
diff --git a/kernel/sysfile.c b/kernel/sysfile.c
index cb25f70..83480ef 100644
--- a/kernel/sysfile.c
+++ b/kernel/sysfile.c
@@ -15,7 +15,7 @@
 #include "sleeplock.h"
 #include "file.h"
 #include "fcntl.h"
-
+#include "memlayout.h"
 // Fetch the nth word-sized system call argument as a file descriptor
```

```
 // and return both the descriptor and the corresponding struct file.
 static int
@@ -504,10 +504,69 @@ sys_pipe(void)
   return 0;
 }


 uint64
 sys_mmap(void)
 {
-  return -1;
+  uint64 addr;
+  uint64 offset_temp;
+  int len, prot, flags, fd;
+
+  struct proc *p = myproc();
+
+  argaddr(0, &addr);
+  argint(1, &len);
+  argint(2, &prot);
+  argint(3, &flags);
+  argint(4, &fd);
+  argaddr(5, &offset_temp);
+
+  uint64 offset = (uint64) offset_temp;
+
+  if(len == 0 || (offset % PGSIZE) != 0)
+    return 0xffffffffffffffff;
+
+  struct file *f = p->ofile[fd]; // reference the open file
+
+  if(flags & MAP_SHARED){
+    if(!(f->writable) && (prot & PROT_WRITE)){
+      return 0xffffffffffffffff;
+    }
+  }
+
+  uint64 cur_max = p->cur_vma_start;
+  uint64 start_addr = cur_max;
+  uint64 end_addr = PGROUNDUP(cur_max + len);
+  if(end_addr >= MMAPTOP)
+    return 0xffffffffffffffff;
+  struct vma_t *v = 0;
+  for(int i = 0; i < NVMA; i++){
+    if(p->vma[i].valid == 0){
+      v = &p->vma[i];
+      break;
+    }
+  }
+
+  if(v){
+    v->valid = 1;
+    v->start = start_addr;
+    v->end = end_addr;
+    v->len = end_addr - start_addr;
+    v->prot = prot;
+    v->flags = flags;
+    v->fd = fd;
+    v->file = p->ofile[fd];
```

```
+    if(v->file)
+      filedup(v->file);
+    p->cur_vma_start = end_addr;
+    v->offset = 0;
+  } else {
+    return 0xffffffffffffffff;
+  }
+
+  return start_addr;
 }
```

4.4 Execution output



5 Task 4 - Page Fault Handling

In this part of the assignment, the fault handler is implemented to handle faults that happen in the virtual address ranges corresponding to the memory mapped address. The important changes are made to implement the vmfault function.

5.1 kernel/vm.c

```
void            plicinit(void);
diff --git a/kernel/vm.c b/kernel/vm.c
index 28f4248..4cc4052 100644
--- a/kernel/vm.c
+++ b/kernel/vm.c
@@ -7,7 +7,9 @@
 #include "spinlock.h"
 #include "proc.h"
 #include "fs.h"
-
+#include "fcntl.h"
+#include "sleeplock.h"
+#include "file.h"
 /*
  * the kernel's page table.
  */
@@ -455,6 +457,10 @@ vmfault(pagetable_t pagetable, uint64 va, int read)
   uint64 mem;
   struct proc *p = myproc();
```

```
+  if(va >= MMAPBASE && va <= p->cur_vma_start){
+    // address inside the VMA area of the process
+    return lazy_mmap(pagetable, va, read);
+  }
   if (va >= p->sz)
     return 0;
   va = PGROUNDDOWN(va);
@@ -472,6 +478,61 @@ vmfault(pagetable_t pagetable, uint64 va, int read)
   return mem;
 }

+uint64 lazy_mmap(pagetable_t pagetable, uint64 va, int read){
+  // find the VMA in which the mapping exists
+  struct vma_t *v = 0;
+  struct proc *p = myproc();
+  for(int i = 0; i < NVMA; i++){
+    if(p->vma[i].valid && p->vma[i].start <= va && va < p->vma[i].end){
+      v = &p->vma[i];
+      break;
+    }
+  }
+
+  if(v == 0){
+    printf("no vma slot found\n");
+    return 0;
+  }
+
+  if(read){
+    if(!(v->prot & PROT_READ))
+      return 0;
+  } else {
+    if(!(v->prot & PROT_WRITE))
+      return 0;
+  }
+
+  char *mem = kalloc();
+  if(mem == 0){
+    printf("lazymmap kalloc fail\n");
+    return 0;
+  }
+  memset(mem, 0, PGSIZE);
+
+  if (v->file) {
+    uint64 off = v->offset + (va - v->start);
+    ilock(v->file->ip);
+    int n = readi(v->file->ip, 0, (uint64)mem, off, PGSIZE);
+    iunlock(v->file->ip);
+    // n may be < PGSIZE, rest already zeroed
+    if (n < 0) {
+      printf("negative size write in lazymmap\n");
+      return 0;
+    }
+  }
+
+  int perm = PTE_V | PTE_U;
+  if(v->prot & PROT_READ) perm |= PTE_R;
+  if(v->prot & PROT_WRITE) perm |= PTE_W;
+
+  if(mappages(pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, perm) != 0){
```

```
+    printf("lazymmap mapping failed\n");
+    kfree((void*)mem);
+    return 0;
+  }
+
+  return (uint64)mem;
+}
 int
 ismapped(pagetable_t pagetable, uint64 va)
 {
```

5.2 Execution and output

```
hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
test basic mmap
mmaptest failure: munmap (1), pid=3
panic: freewalk: leaf
QEMU: Terminated
```

6 Task 5 - Implement munmap

In this part of the assignment, the munmap syscall is implemented. The changes in this file also have parts relevant to implement the task 8 of shared mapping. The implementation specifically attempts to find a valid address range in the MMAP area of the process and uses one such instance to map the memory address. The output for this task is as below. Please ignore the debug information dprinted. This was used to verify the operations correctly.

6.1 Execution and output

```
hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
test basic mmap
found vma for start:40000000, end:40002000, vma: start:40000000 end:40002000
test basic mmap: OK
test mmap private
found vma for start:40002000, end:40004000, vma: start:40002000 end:40004000
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
found vma for start:40004000, end:40006000, vma: start:40004000 end:40007000
writing back in munmap due to shared
write back in mmunmap successful
write back in mmunmap successful
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
found vma for start:40006000, end:40007000, vma: start:40006000 end:40007000
writing back in munmap due to shared
write back in mmunmap successful
test not-mapped unmap: OK
test lazy access
found vma for start:40007000, end:40009000, vma: start:40007000 end:40009000
writing back in munmap due to shared
write back in mmunmap successful
write back in mmunmap successful
test lazy access: OK
test mmap two files
found vma for start:40009000, end:4000a000, vma: start:40009000 end:4000a000
found vma for start:4000a000, end:4000b000, vma: start:4000a000 end:4000b000
test mmap two files: OK
test fork
usertrap(): unexpected scause 0xd pid=4
            sepc=0x5c stval=0x4000b000
fork_test failed
panic: freewalk: leaf
```

The expected output is seen here.

7 Task 6 - Process Exit Handling

In this task, the kexit function is modified to unmap the pages when the process exits. The changes in the kexit function also require additional changes for task 8. The output of this test is given below. Again, ignore the debug information printed for verification.

7.1 Execution and output

```
$ mmaptest
test basic mmap
found vma for start:40000000, end:40002000, vma: start:40000000 end:40002000
test basic mmap: OK
test mmap private
found vma for start:40002000, end:40004000, vma: start:40002000 end:40004000
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
found vma for start:40004000, end:40006000, vma: start:40004000 end:40007000
writing back in munmap due to shared
write back in mmunmap successful
write back in mmunmap successful
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
found vma for start:40006000, end:40007000, vma: start:40006000 end:40007000
writing back in munmap due to shared
write back in mmunmap successful
test not-mapped unmap: OK
test lazy access
found vma for start:40007000, end:40009000, vma: start:40007000 end:40009000
writing back in munmap due to shared
write back in mmunmap successful
write back in mmunmap successful
test lazy access: OK
test mmap two files
found vma for start:40009000, end:4000a000, vma: start:40009000 end:4000a000
found vma for start:4000a000, end:4000b000, vma: start:4000a000 end:4000b000
test mmap two files: OK
test fork
usertrap(): unexpected scause 0xd pid=4
            sepc=0x5c stval=0x4000b000
fork_test failed
```

8 Task 7 - Fork handling

In this part of the assignment, the kfork() function is modified to copy the vma metadata from parent to child in the event of a fork. The most significant change is to find a new VMA unit in the child that is available, and to copy every entry from the parent VMA into the child. The outputs are shown below.

8.1 Execution and output

```
hart 2 starting
hart 1 starting
init: starting sh
$ mmaptest
test basic mmap
test basic mmap: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test lazy access
test lazy access: OK
test mmap two files
test mmap two files: OK
test fork
test fork: OK
test munmap prevents access
no vma slot found
usertrap(): unexpected scause 0xd pid=5
            sepc=0xa30 stval=0x40010000
no vma slot found
usertrap(): unexpected scause 0xd pid=6
            sepc=0xab8 stval=0x4000f000
test munmap prevents access: OK
test writes to read-only mapped memory
usertrap(): unexpected scause 0xf pid=7
            sepc=0xbfe stval=0x40011000
test writes to read-only mapped memory: OK
mmaptest: all tests succeeded
```

9 Task 8 - Implementing shared memory functionality

In this part of the assignment, the previous implementation of the functions and changes are modified to synchronise changes in the event of shared mapping. Note that the changes are synchronised using locking constructs and are separately handled. The previously written code for the fork and exit programs are also modified to support this shared memory mapping. The code is not included here for want of space. The implementation output is seen below.

9.1 Execution and output

```
hart 1 starting
hart 2 starting
init: starting sh
$ mmaptest
test basic mmap
test basic mmap: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test lazy access
test lazy access: OK
test mmap two files
test mmap two files: OK
test fork
test fork: OK
test munmap prevents access
usertrap(): unexpected scause 0xd pid=5
            sepc=0xa30 stval=0x40010000
usertrap(): unexpected scause 0xd pid=6
            sepc=0xab8 stval=0x4000f000
test munmap prevents access: OK
test writes to read-only mapped memory
usertrap(): unexpected scause 0xf pid=7
            sepc=0xbfe stval=0x40011000
test writes to read-only mapped memory: OK
mmaptest: all tests succeeded
$ mmapsharedtest
mmapsharedtest start
test mmap syscall exists
mmap syscall error handled
mmap syscall: ok
test basic map_shared
basic map_shared: ok
test shared physical pages
parent mapped 0x0000000040001000 ok
child mapped 0x0000000040002000
parent: shared page success
child: shared page success
shared_task: passed
mmapsharedtest done
```

All the tests pass successfully. Additionally, the usertests also all pass to ensure no exisiting functionality of the code is broken. This part was slightly complicated to implement and required careful probing of all the code written previously.

9.2 Final usertests

```
usertrap(): unexpected scause 0xf pid=6647
            sepc=0x43ea stval=0x3e012000
usertrap(): unexpected scause 0xf pid=6648
            sepc=0x43ea stval=0x3f012000
OK
test lazy_copy: OK
ALL TESTS PASSED
```