Virtual Memory Report

Camille Williford - awillif4

November 8, 2021

The point of this lab was to familiarize students with how the xv6 simulator implements memory management. I implemented two system calls and prevented dereferencing null pointers for this lab. I am going to take a chronological order to this report detailing changes that I made to the xv6 OS in the order that I made them. Here is a link to the change log in GitHub: Virtual Memory Commit

1 Dereferencing Null Pointer

1.1 Makefile

Modified: 3 Lines
I changed the number of CPU's to 1, and made the entry point for page tables $0x1000$ per the lab write-up.

1.2 exec.c

Modified: 1 Lines
I changed the $exec()$ to not create a page at address 0, but the given $PGSIZE$. This way user processes are not clobbering reserved memory.

1.3 vm.c

Modified: 1 Lines
I changed the $copyuvm()$ to not start copying memory at address 0, but the given $PGSIZE$ i.e. the next page. I changed this function because it is responsible for copying a parent process's address state to a child. Now when $fork()$ is called, the child process will have the correct address space.

2 System Calls

2.1 syscall.c

Modified: 4 Lines
Defined two new system calls $sys\_mprotect()$ and $sys\_munprotect()$ and their respective system functions.

2.2 syscall.h

Modified: 2 Lines
Assigned numbers to the new system calls $sys\_mprotect()$ and $sys\_munprotect()$. These numbers are used when a user initiates a system call to safely determine which system call the user requested be performed.

2.3 user.h

Modified: 2 Lines
I prototyped the two new system calls as callable functions. These are the user functions that can be used to to call into the operating system.

$$int\ mprotect(void\ *addr,\ int\ len);$$

$$int\ munprotect(void\ *addr,\ int\ len);$$

2.4 usys.S

Modified: 2 Lines
Assembly code that loads register with syscall number, makes syscall, and makes transition to the kernel mode using int instruction. I added the two new system calls to this list.

2.5 sysproc.c

Modified: 28 Lines
I created the two new system calls $sys\_mprotect()$ and $sys\_munprotect()$! These are a pass-through to the implementation in vm.c. Each parses the user input address with $argptr()$ and length with $argint()$, and returns -1 if unable to parse arguments.

2.6 vm.c

Modified: 2 Lines
Here the $mprotect()$ and $munprotect()$ system calls are implemented by securely changing the write bit on page table entries. These methods differ by only one line whether we are clearing the write bit in $mprotect()$ or setting the write bit in $munprotect()$. Otherwise, I flush the TLB, check the user given length, and ensure the given address is page aligned. After the user input has been validated, I loop through all the pages within the given length and edit the write bit according to which function is used. Before editing the write bit, I ensure that page table entry is valid and that the user has permission to access that page table entry. If I cannot change the write bit, I return -1. Otherwise, I return 0 after looping through all the pages within a given length.

3 Tests

3.1 nullPtrTest.c

New File
This test simply creates a pointer to address 0x0, and then tries to get the value at that pointer.

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char **argv) {
  uint* nullptr = (uint*) 0;
  printf(1, "%x %x\n", nullptr, *nullptr);
}
```

```
$
$ nullPtrTest
pid 11 nullPtrTest: trap 14 err 4 on cpu 0 eip 0x1004 addr 0x0--kill proc
$
```

3.2 protectTest.c

New File

This test is more complex, testing that *mprotect*() and *munprotect*() work appropriately, so this test can be broken into two sections one for each system call. First I get the current process id, and then I initialize an address to protect and unprotect. I check *munprotect*() first by unproctecting a protected space and changing the value, which should not cause a trap. Second, I check that *mprotect*() properly protects an address space, by attempting to edit a protected value. This will cause a trap, since the protected space if uneditable.

```c
int
main(int argc, char *argv[])
{
  int parent = getpid();
  // Create an address to protect
  int *val = (int *) sbrk(0);
  // Make sure the process memory size is at least a page
  sbrk(PGSIZE);
  // Protect the address
  mprotect(val, 1);

  if (fork() == 0) {
    munprotect(val, 1);
    *val = 5;
    printf(1, "TEST PASSED: unprotected value did not cause trap\n");
    exit();

  } else {
    wait();
  }

  if(fork() == 0) {
    sleep(5);
    *val = 5;
    printf(1, "TEST FALIED: protected value did not cause trap\n");
    kill(parent);
    exit();

  // Parent: wait for child
  } else {
    wait();
  }
  printf(1, "TEST PASSED: protected value caused trap\n");
  exit();
}
```

```
$
$ protectTest
TEST PASSED: unprotected value did not cause trap
pid 18 protectTest: trap 14 err 7 on cpu 0 eip 0x1087 addr 0x4000--kill proc
TEST PASSED: protected value caused trap
$
```