Mini Project 3: Memory

Rubric

Report 20%
Code and documentation 80%
This Rubric just applied after presentation; no presentation means zero in all the previous components

Note: The rubric above takes place just after the presentation.

Participant name	Code Section	Report Section	Documentation Sec	Presentation Sec

Submission deadline: Nov 18th at 11:59 pm

Working in class: Nov 18th at class time.

Instructions: Download XV6 directly from Canvas (fresh version)

Deliveries:

- Upload a zip version of your modified XV6 to Canvas.dd
- Report including the list of modifications. This list of modifications should be formatted in a table indicating the name of the modified file and the line of the modification. Show screenshots of your modification. In addition, include the code and output of the testing part.

Note: The only way to receive your grade is by presenting your project in class. Remember, all the team members should participate in the presentation. If a team member is absent she/he does not receive a grade.

Objectives

There are two objectives to this assignment:

- To familiarize you with the xv6 virtual memory system.
- To add a few new Virtual Memory (VM) features to xv6 that are common in modern OSs.

Resources

- VIDEO Project descriptions by Dr. Remzi H. Arpaci-Dusseau (Book author) : https://www.youtube.com/watch?v=M2CPjjVTmpg

- **Definition of Null Pointer Dereference:** What EXACTLY is meant by "de-referencing a NULL pointer"? Check the answer in this link:http://stackoverflow.com/questions/4007268/what-exactly-is- meant-by-de-referencing-a-null-pointer

FLORIDAPOLY

Overview

In this project, you'll be changing xv6 to support a feature virtually every modern OS does: causing an exception to occur when your program dereferences a null pointer. Sound simple? Well, it mostly is. But there are a few details.

Project description

In xv6, the VM system uses a simple two-level page table as discussed in class. As it currently is structured, user code is loaded into the very first part of the address space. Thus, if you dereference a null pointer, you will not see an exception (as you might expect); rather, you will see whatever code is the first bit of code in the program that is running. Try it and see!

Thus, the first thing you might do is create a program that dereferences a null pointer. It is simple! See if you can do it. Then run it on Linux as well as xv6, to see the difference.

Your job here will be to figure out how xv6 sets up a page table. Thus, once again, this project is mostly about understanding the code, and not writing very much. Look at how exec() works to better understand how address spaces get filled with code and in general initialized. That will get you most of the way.

You should also look at fork() , in particular the part where the address space of the child is created by copying the address space of the parent. What needs to change in there?

The rest of your task will be completed by looking through the code to figure out where there are checks or assumptions made about the address space. Think about what happens when you pass a parameter into the kernel, for example; if passing a pointer, the kernel needs to be very careful with it, to ensure you haven't passed it a bad pointer. How does it do this now? Does this code need to change in order to work in your new version of xv6?

One last hint: you'll have to look at the xv6 makefile as well. In there user programs are compiled so as to set their entry point (where the first instruction is) to 0. If you change xv6 to make the first page invalid, clearly the entry point will have to be somewhere else (e.g., the next page, or 0x1000). Thus, something in the makefile will need to change to reflect this as well.

You should be able to demonstrate what happens when user code tries to access a null pointer. When this happens, xv6 should trap and kill the process. The good news: this will happen without too much trouble on your part, if you do the project in a sensible way, because xv6 already catches illegal memory accesses.

In summary, you will need to make changes to the following files:

- exec.c, in function int exec()
- vm.c in function copyuvm
- user/ makefile.mk # location in memory where the program will be loaded

Passing the first two test cases

- makefile.mk: Change the place where the program will start to load
- kernel\exec.c : Change the value of PGSIZE to to leave the null page inaccessible.
- Kernel\vm.c: There check the functions// in copyuvm

Passing the third case (You need more time and dedication to pass this case)

Note about third case: Test case three contains bad pointers being passed into the xv6 write() method. The first, (a), is a null pointer, the second, (b), is a pointer that starts at a non-zero location but still in the null page, and the third (c) is a pointer that starts in the correct place but the length of the object extends beyond the stack and into the code section of the program space.

Changes to pass this case include changes in **kernel\syscall.c,** in particular, the following functions:

Clue 1 In **Fetchint** method, we check that the int, which are of size 4 in xv6, do not extend outside of the stack into the code of the program (USERTOP). (modify the part of the code in RED) !!!

```
// Fetch the int at addr from process p.
int
fetchint(struct proc *p, uint addr, int *ip)
{
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
```

Clue 2 In **fetchstr** method check that the integer does not touch the code of the program (USERTOP). If it is so, we trap this case and return -1.

```
int
fetchstr(struct proc *p, uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= p->sz)
    return -1;
  *pp = (char*)addr;
  ep = (char*)p->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

.

Clue 3 if(addr >= p->sz) //extend this check so that addr is greater than USERTOP (code section of the process)

```
    return -1;
  *pp = (char*)addr;
  ep = (char*)p->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

Clue 4: Finally, In *argptr* function we check that the pointers remain in the stack and they do not point to the code of the program.!!

Here you have to add many checks (Red if statement) . Pointers must remain in the stack, above memory location
PGSIZE or 0x1000 or 4096 but not into the user space of the program (USERTOP). Additionally, any part of the
pointer may not end up in user space.


```
int
argptr(int n, char **pp, int size)
{
  int i;
  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

TESTING CASES

Testing cases are provided through the file test1-test2-test3.c all place it in user folder and include it in the makefile.mk in user