COP4600 Operating Systems

Assignment 5

Due: Monday, December 2nd by midnight

1 Problem

In this assignment you are going to add a new system call to the Linux kernel to find out whether a virtual address in a user process address space is part of a copy-on-write page or not. As we discussed in class, when a parent process creates a child process the kernel lets the parent process and the child process share parent's pages as long as they access those pages for reading only. When one of those processes attempts to write to a copy-on-write page, a page fault is generated. A page fault of this type is handled by creating a new copy for that process and clearing the copy-on-write flag on the page table entry.

Linux can recognize page faults due to writing to a copy-on-write type of page because the virtual memory area that the page belongs to has WRITE access permission whereas the page table entry has READ-ONLY permission.

For testing your code, you will be using the same setup that you used for Assignment 4.

2 Guideline

This section explains the specific kernel data structures you will be using (see Section 2.1), the individual tasks you need to do in your implementation (see Section 2.2), where to put your code in the kernel source tree (see Section 2.3), and finally how to test your program in a UML session (see Section 2.4).

2.1 Kernel Data Structures of Interest

To be able to do this assignment, you should learn about the following data structures/variables:

- Kernel variable *current* points to the current user process' task_struct.
- Type *task_struct* represents a process control block (PCB).
- Type *mm_struct* represents address space of a process.

- Type *vm_area_struct* represents a virtual memory area such as the stack segment, the data segment, etc.

2.2 How to Implement

You need to implement a new system call `is_address_cow` (system call no 312 for kernel version 3.2) that takes a single parameter: unsigned long address. This system call should return 1 if the address corresponds to a copy-on-write page.

To implement the system call, you need to find out how to do the following:

- Find the virtual memory area that corresponds to a given virtual address.
- Check whether the virtual memory area has WRITE permission.
- Walk the page table, i.e., map the virtual address to a page table entry.
- Check whether the page table entry is READ-ONLY.

Important Note: If the kernel provides a function to do any of the above tasks, you should use that function rather than implementing a new function to navigate the data structure. The reason is that such navigation of kernel data structures requires using appropriate synchronization mechanisms to avoid race conditions and this can be challenging. However, when you use a function provided by the kernel, hopefully, it will be doing the appropriate synchronization. **Hint:** You may also need to copy an existing kernel function, rename it, and in that new function gather the information that you are looking for and return that information without causing any side-effects. Check out *follow_page* function to see how you can use it in this way.

2.3 Where to Put Your Code in the Kernel

We suggest that you create a new file mysyscall.c under *mm* directory of the kernel source tree and put the system call (sys_is_address_cow) implementation and any helper functions in this file.

You will also need to update *mm/Makefile* to have mysyscall.c compiled and linked to the kernel image.

Anytime you make a change to mysyscall.c, you will need to rebuild the kernel, copy the kernel image to the directory where you execute the UML session.

2.4 How to Test

We provided a test file, testCOW.c, (see Section 5.1) to demonstrate the copy-on-write mechanism. The test program basically creates a child process using fork() and uses semaphores to let the parent process and the child process synchronize so that they can take turns doing their show. Each process changes some global variable and displays the values of the global variables as well as their copy-on-write status by making a system call (is_address_cow). You can find a sample output in Section 5.2.

You need to mount a directory on the host file system (your CISE user account) at mount point /mnt so that you can access your host files inside a User Mode Linux (UML) session. This will especially be useful as we can write some test programs and compile them on the host system. This way the only thing we would need to do inside a UML session is to access those executables and run them.

1. Create a directory on your home directory, if you have not created the same one already:

   ```
   $ cd ~
   $ mkdir umlTest
   $ cd umlTest
   ```

   and copy testCOW.c under this directory.

2. Compile testCOW.c and generate an executable named testCOW (DO NOT FORGET TO INCLUDE -lpthread IN THE COMMAND BELOW):

   ```
   $ gcc testCOW.c -o testCOW -lpthread
   ```

3. Now change directory to /cise/tmp/CISEUserName/kernelHacking/uml directory.

   ```
   $ cd /cise/tmp/CISEUserName/kernelHacking/uml
   ```

4. Start UML with COW mode and login as root (**make sure that the command given below appears on a single line and there is no space before or after the comma that separates the local cow file name and the shared file name**):

   ```
   $ ./linux ubd0=/cise/homes/YourCISEUserName/uml/my-fs-cow,
   /cise/class/cop4600fa13/sharedRootFS/Debian-Squeeze-AMD64-root_fs mem=128M
   ```

5. Mount the host directory that contains your executable file testCOW at /mnt in UML:

   ```
   root@(none):/# mount none /mnt -t hostfs -o /cise/homes/CISEUserName/umlTest
   ```

6. Change directory to /mnt and execute ls command to make sure you can see all the files, including test, under the home directory that you mounted :

   ```
   root@(none):/# cd /mnt
   root@(none):/# ls
   ```

7. Now, execute test inside UML:

   ```
   root@(none):/# ./testCOW
   ```

3 Resources

- Attend the discussion session on Tuesday, November 19th.
- Check out documentation on Linux Memory Management:
  - https://www.kernel.org/doc/gorman/html/understand/understand007.html
  - https://www.kernel.org/doc/gorman/html/understand/understand006.html
- Use the Linux Source Cross Reference Website to help you navigate the kernel source code: http://lxr.linux.no/+trees

4 Submission Instructions

Please submit mysyscall.c and any other file that you created.

5 Appendix

5.1 Test file

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <fcntl.h>

long isAddressCow(unsigned long address)
{
      long ret;
      __asm__("mov $312, %rax"); // storing the system call number
      __asm__("syscall");
      __asm__("mov %rax, -8(%rbp)");

      return ret;
}

int cowByParent[2000];
long stuff[1000];
int cowByChild[2000];

void checkCowStatus()
{
   printf("Global variables: cowByParent[0](@%lx)=%d cowByChild[1999](@%lx)=%d\n",
        (unsigned long)cowByParent,cowByParent[0], (unsigned long)&cowByChild[1999],
        cowByChild[1999]);
   printf("%lx on a cow page?=%ld\n",(unsigned long)cowByParent,
          isAddressCow((unsigned long)cowByParent));
   printf("%lx on a cow page?=%ld\n",(unsigned long)&cowByChild[1999],
          isAddressCow((unsigned long)&cowByChild[1999]));
}
```

```
int main()
{
   /* You should give another name to your semaphore! */
   /* 0600 has RW permission and 0 for initial value of 0 */
   sem_t *parentGoPtr = sem_open("tyavuz_parentGo", O_CREAT, 0600, 0);
   sem_t *childGoPtr = sem_open("tyavuz_childGo", O_CREAT, 0600, 0);

   printf("*************PARENT*************\n");
   printf("Creating a child\n");
   int pid = fork();
   if (pid == 0)    /* You should give another name to your semaphore! */
   /* 0600 has RW permission and 0 for initial value of 0 */
   { // Child
     printf("=============CHILD=============\n");
     checkCowStatus();
     printf("Changing GLOBAL variable at address %lx\n",
             (unsigned long)&cowByChild[1999]);
     cowByChild[1999] = 10;
     checkCowStatus();
     printf("=============Let me wake up my parent...=============\n");
     sem_post(parentGoPtr);
     sem_wait(childGoPtr);
     printf("=============CHILD again...=============\n");
     checkCowStatus();
     printf("Changing GLOBAL variable at address %lx\n",
             (unsigned long)cowByParent);
     cowByParent[0] = 2;
     checkCowStatus();
     printf("That's all folks\n");
     return 0;
   }
   else
   { // Parent
     int status;
     sem_wait(parentGoPtr);
     printf("*************PARENT again...*************\n");
     checkCowStatus();
     printf("Changing GLOBAL variable at address %lx\n",
             (unsigned long)cowByParent);
     cowByParent[0] = 3;
     checkCowStatus();
     printf("Changing GLOBAL variable at address %lx\n",
             (unsigned long)&cowByChild[1999]);
     cowByChild[1999] = 11;
     checkCowStatus();
     printf("*************Let me wake up my child...*************\n");
     sem_post(childGoPtr);
     wait(&status);
   }
   return 0;
}
```

5.2 Sample Output

```
*************PARENT*************
Creating a child
```

```
=============CHILD=============
Global variables: cowByParent[0](@602fc0)=0 cowByChild[1999](@606e3c)=0
Found vma: start=602000 end=628000
602fc0 on a cow page?=1
Found vma: start=602000 end=628000
606e3c on a cow page?=1
Changing GLOBAL variable at address 606e3c
Global variables: cowByParent[0](@602fc0)=0 cowByChild[1999](@606e3c)=10
Found vma: start=602000 end=628000
602fc0 on a cow page?=1
Found vma: start=602000 end=628000
606e3c on a cow page?=0
=============Let me wake up my parent...=============
*************PARENT again...*************
Global variables: cowByParent[0](@602fc0)=0 cowByChild[1999](@606e3c)=0
Found vma: start=602000 end=628000
602fc0 on a cow page?=1
Found vma: start=602000 end=628000
606e3c on a cow page?=1
Changing GLOBAL variable at address 602fc0
Global variables: cowByParent[0](@602fc0)=3 cowByChild[1999](@606e3c)=0
Found vma: start=602000 end=628000
602fc0 on a cow page?=0
Found vma: start=602000 end=628000
606e3c on a cow page?=1
Changing GLOBAL variable at address 606e3c
Global variables: cowByParent[0](@602fc0)=3 cowByChild[1999](@606e3c)=11
Found vma: start=602000 end=628000
602fc0 on a cow page?=0
Found vma: start=602000 end=628000
606e3c on a cow page?=0
*************Let me wake up my child...*************
=============CHILD again...=============
Global variables: cowByParent[0](@602fc0)=0 cowByChild[1999](@606e3c)=10
Found vma: start=602000 end=628000
602fc0 on a cow page?=1
Found vma: start=602000 end=628000
606e3c on a cow page?=0
Changing GLOBAL variable at address 602fc0
Global variables: cowByParent[0](@602fc0)=2 cowByChild[1999](@606e3c)=10
Found vma: start=602000 end=628000
602fc0 on a cow page?=0
Found vma: start=602000 end=628000
606e3c on a cow page?=0
That's all folks
```