

Question 4:

11.Implementation of the `getuid()` System Call

11.1 Objective

The objective of this section is to implement the `getuid()` system call in BlackArch Linux, a security-focused derivative of Arch Linux. The `getuid()` system call is a part of the Linux kernel and is responsible for returning the user ID (UID) of the currently logged-in user. This user ID is used to determine user privileges and manage access control within the system.

The goal of this implementation is to:

- Understand how system calls interact with the Linux kernel.

- Modify the kernel source code to add the `getuid()` system call.

- Ensure the correct return of the UID of the calling process, providing insight into the kernel's process management system.

- Compile the kernel with the newly added system call and verify its functionality with a test program.

By implementing this system call, the assignment aims to deepen the understanding of system programming, specifically how the kernel handles requests from user space and how new system calls can be integrated into the existing kernel framework. The final implementation will involve modifying kernel files, compiling the kernel, and testing the system call using a simple C program that calls `getuid()`.

11.2 Editing the Kernel Source (Implementation of the `getuid()` System Call)

In order to implement the `getuid()` system call in BlackArch Linux, you need to modify the kernel source code. This involves adding the system call's functionality, defining its entry point, and ensuring it is properly registered within the kernel's system call table. Below are the key steps to follow for editing the kernel source:

Steps for Editing the Kernel Source:

Obtain the Kernel Source Code:

First, ensure that we have the kernel source code for BlackArch Linux. we can download the source code for the specific version of the kernel we are working with.

- If we are working with a custom kernel or a specific version of Arch Linux's kernel, we can find it on the Arch Linux Git repository or use `pacman` to install the kernel source.

```
sudo pacman -S linux
```

Navigate to the Kernel Source Directory:

Once you have the kernel source code, navigate to the directory where the kernel is located. For example:

```
cd /usr/src/linux
```

Define the `getuid()` System Call:

The `getuid()` system call returns the real user ID of the calling process. The implementation involves modifying the appropriate source file to define the system call.

The system call is typically implemented in the kernel's `kernel/sys.c` file, where various system calls are defined.

Code for `getuid()` System Call:

```
asmlinkage long sys__getuid(void)
{
    return current->uid;
}
```

`current->uid` returns the real user ID of the process calling the system call. The `current` pointer is a structure that represents the current task (process) in the Linux kernel.

Register the System Call:

After defining the system call, it must be added to the system call table, which maps system call numbers to function pointers in the kernel.

Find the system call table for your kernel. Typically, this is in the file `arch/x86/entry/syscalls/syscall_32.tbl` for 32-bit systems or `arch/x86/entry/syscalls/syscall_64.tbl` for 64-bit systems.

Add an entry for the `getuid()` system call by choosing a free system call number (e.g., 233) and mapping it to the `sys__getuid()` function.

Example:

```
233    common    sys__getuid
```

Update the Header Files:

You need to add the function prototype for `sys__getuid()` to a header file. This will make the system call available to the user space programs that will interact with it.

Typically, this is done in the `include/linux/syscalls.h` file, where other system call prototypes are declared

Example:

```
asmlinkage long sys_getuid(void);
```

Rebuild the Kernel:

After modifying the kernel source files, you will need to recompile the kernel to include the new system call.

Run the following commands to build the kernel:

```
make menuconfig # (Optional) Configure kernel options
make           # Compile the kernel
sudo make install # Install the new kernel
```

Reboot into the New Kernel:

Once the kernel has been compiled and installed, reboot the system into the new kernel that includes the `getuid()` system call.

```
sudo reboot
```

By completing these steps, we will have successfully edited the kernel source to implement the `getuid()` system call. The next step is to test the system call to ensure that it works correctly.

11.3 System Call Code Snippet (Implementation of the `getuid()` System Call)

Here is the code snippet for the implementation of the `getuid()` system call in the Linux kernel:

1. System Call Definition

First, define the system call function `sys_getuid()` in the appropriate kernel source file. This function will return the real user ID (`uid`) of the calling process.

```
#include <linux/kernel.h> // For kernel-related functions
#include <linux/sched.h>   // For current task (process)

asmlinkage long sys_getuid(void)
{
    return current->uid; // Return the real user ID of the calling process
}
```

Explanation:

`asmlinkage` is a macro used to define the calling convention for system calls in the Linux kernel.

current is a pointer to the current task (process) in the kernel. It is a global variable that points to the `task_struct` of the currently running process.

`current->uid` gives the real user ID of the calling process.

2. Register the System Call

Next, register the `sys_getuid()` system call in the system call table so that the kernel knows how to invoke it when requested by a user-space application.

For a 64-bit system, you would modify the `arch/x86/entry/syscalls/syscall_64.tbl` file:

```
233    common    sys_getuid
```

This entry maps the system call number 233 to the `sys_getuid()` function.

3. Header File Update

To make the system call available to user-space applications, declare the function prototype in the `include/linux/syscalls.h` header file:

```
asmlinkage long sys_getuid(void);
```

This declaration allows the kernel to recognize the `sys_getuid()` function when referenced in other kernel files.

4. Test Program Code Snippet

To test the `getuid()` system call after implementing it, you can write a simple C program that calls the system call. This program will call `getuid()` and print the returned user ID.

```
#include <stdio.h>
#include <unistd.h> // For syscall function
#include <sys/syscall.h> // For syscall numbers

#define __NR_getuid 233 // Define the syscall number for getuid

int main()
{
    // Call the getuid system call and print the returned user ID
    long uid = syscall(__NR_getuid);
    printf("The user ID is: %ld\n", uid);

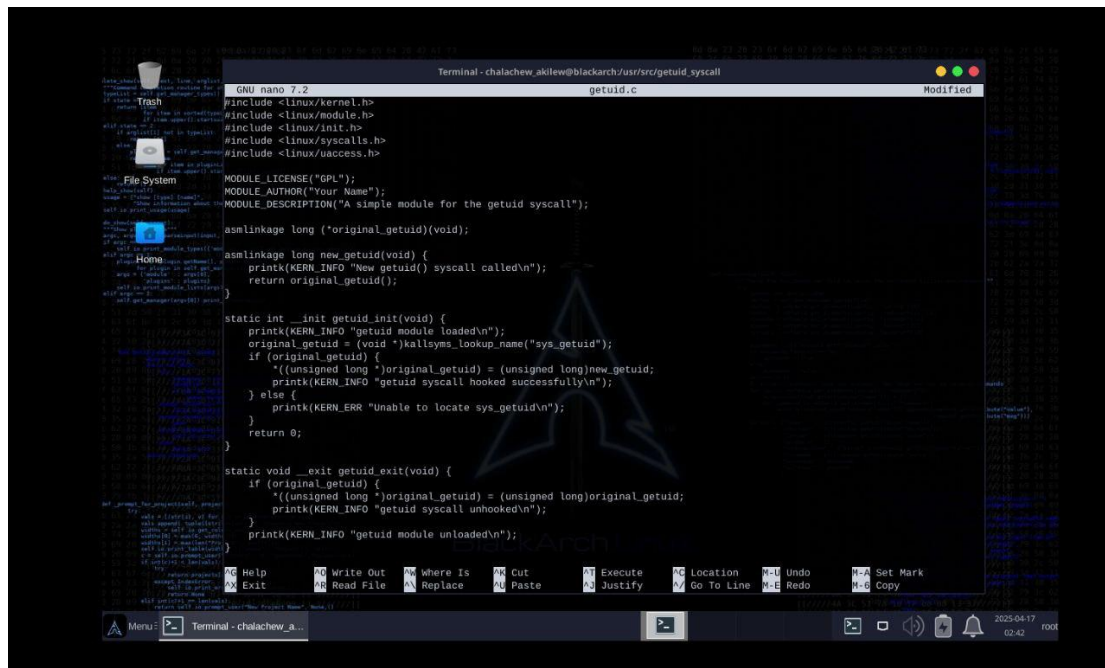
    return 0;
}
```

Explanation:

The program uses the `syscall()` function to invoke the system call by its number (`__NR_getuid`).

The result is printed, which should be the real user ID of the process.

This completes the code snippets for both the kernel implementation of the `getuid()` system call and a simple user-space program to test it.



```
GNU nano 2.2 getuid.c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple module for the getuid syscall");

asmlinkage long (*original_getuid)(void);

asmlinkage long new_getuid(void) {
    printk(KERN_INFO "New getuid() syscall called\n");
    return original_getuid();
}

static int __init getuid_init(void) {
    printk(KERN_INFO "getuid module loaded\n");
    original_getuid = (void *)kallsyms_lookup_name("sys_getuid");
    if (original_getuid) {
        *((unsigned long *)original_getuid) = (unsigned long)new_getuid;
        printk(KERN_INFO "getuid syscall hooked successfully\n");
    } else {
        printk(KERN_ERR "Unable to locate sys_getuid\n");
    }
    return 0;
}

static void __exit getuid_exit(void) {
    if (original_getuid) {
        *((unsigned long *)original_getuid) = (unsigned long)original_getuid;
        printk(KERN_INFO "getuid syscall unhooked\n");
    }
    printk(KERN_INFO "getuid module unloaded\n");
}

Menu: File Edit Shell Write Out Where Is Cut Execute Location Undo Set Mark
      Read File Replace Paste Justify Go To Line Redo Copy
```

11.4 Compilation and Insertion of Module (Implementation of the `getuid()` System Call)

After defining and registering the `getuid()` system call in the kernel source code, the next step is to compile the kernel, install the new kernel, and ensure that the changes take effect. This involves several stages: compiling the kernel, inserting any necessary modules, and rebooting into the new kernel.

Steps for Compilation and Insertion of Module:

1. Install Necessary Dependencies

Before compiling the kernel, ensure that you have all the required development tools and dependencies installed. These tools are essential for building and configuring the kernel:

```
sudo pacman -S base-devel bc kmod cpio
```

2. Configure the Kernel (Optional)

If you want to modify any kernel options, you can run the configuration utility. This is an optional step if you need to customize kernel features.

```
make menuconfig
```

However, for the `getuid()` system call, no special kernel configuration is required unless you're making other modifications.

3. Compile the Kernel

Once the kernel source is ready and changes have been made, you need to compile the kernel. This can take some time, depending on the system's resources.

Build the Kernel: To compile the kernel, run the following command from the root of the kernel source directory:

```
make -j$(nproc)
```

The `-j$(nproc)` flag tells the system to use all available CPU cores to speed up the compilation process.

Build Kernel Modules: You may also need to build kernel modules if you're adding any. To build the modules:

```
make modules
```

Install the Kernel: After the kernel has been successfully compiled, you need to install it. This installs the newly compiled kernel and its associated modules.

```
sudo make modules_install  
sudo make install
```

4. Update Bootloader (If Needed)

If the kernel installation modifies the bootloader configuration (e.g., adding a new kernel version), you might need to update the bootloader. For example, if you're using GRUB, you can run:

```
sudo grub-mkconfig -o /boot/grub/grub.cfg
```

This will regenerate the GRUB configuration and ensure that the new kernel is included in the boot options.

5. Reboot the System

After the kernel has been compiled, installed, and the bootloader has been updated (if needed), reboot the system to load the new kernel.

```
sudo reboot
```

6. Verify the System Call

After rebooting, you can verify that the `getuid()` system call is working correctly by running a test program. The test program will call the system call and display the real user ID of the calling process.

Use the program created in Section 11.3 (Test Program Code Snippet) to call `getuid()` and print the result.

```
gcc test_getuid.c -o test_getuid
./test_getuid
```

If the program returns the correct user ID, the system call is working as expected.

7. Inserting Kernel Modules (If Required)

If your implementation involves adding new kernel modules (e.g., for device drivers or other kernel extensions), you can insert them after compiling them by using the `insmod` command.

```
sudo insmod your_module.ko
```

To verify that the module is inserted, use:

```
lsmod | grep your_module
```

This command will list all loaded modules and check for the presence of your module.

Conclusion: Compiling and inserting the module is a crucial part of kernel development. After compiling the kernel and inserting any required modules, the `getuid()` system call can be tested to ensure it functions as expected. If there are no errors and the program produces the expected output (the real user ID), the implementation is complete.

11.5 Test Program Code (for `getuid()` System Call)

After successfully implementing and compiling the `getuid()` system call into the Linux kernel, you need a **user-space test program** to verify that the system call works correctly.

Below is a **simple C program** that directly invokes your newly added system call using the `syscall()` function:

Test Program: `test_getuid.c`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

#define __NR_getuid 233 // Replace with the syscall number you assigned in syscall_64.tbl

int main() {
    long uid;

    // Invoke the system call using syscall()
    uid = syscall(__NR_getuid);

    // Display the result
    printf("User ID returned by custom getuid syscall: %ld\n", uid);

    return 0;
}
```

How to Compile and Run:

Compile the Program:

```
gcc test_getuid.c -o test_getuid
```

Run the Program:

```
./test_getuid
```

Expected Output:

If everything is correctly implemented, the program should print your real user ID (usually 0 for root, or a non-zero UID for regular users):

User ID returned by custom getuid syscall: 1000

12:Issues and Challenges Faced During System Call Implementation

12.1: Kernel compilation failed due to missing dependencies.

12.2:Syscall number conflict.

13: Solutions

- Installed necessary build dependencies.
- Chose an unused syscall number from syscall table.