Obadah Abozaid

ID: 5351127

Exploring Kernel Virtual Machine (KVM) and System Call Modifications

Introduction

In this report I will explore a Linux Kernel Virtual Machine (KVM) and how it can be used to simulate modifications made directly to the kernel of an operating system - from the initial boot to a fully functional system. I will also add two of my own system calls to demonstrate the functionality of making modifications to a system kernel, and to implement my own function using native headers.

Understanding System Calls

System calls provide the interface between a running program and the operating system. They are a set of functions that are provided by the operating system's kernel, which applications running in user mode can call. When a system call is invoked, the execution context switches from user mode to kernel mode, allowing the program to access protected system resources.

Examples of system calls include read(), write(), open(), close(), wait(), exec(), fork(), exit(), among others. These calls allow a program to interact with the system, such as reading from or writing to files, creating new processes, and communicating over a network.

System calls are crucial for several reasons:

Resource Management: System calls allow programs to request resources from the operating system. From requesting memory to creating a new process, the operating system can manage these resources, ensuring that all programs get executed effectively.

Abstraction: System calls provide an abstraction layer between the hardware and the application software, implying a developer does not need to know the specifics of how certain hardware works to interact with it.

Security and Stability: By requiring programs to use system calls to access system resources, the operating system can ensure that programs do not accidentally or maliciously interfere with each other. This helps maintain the stability of the system and protects it from potential security threats.

System calls are a fundamental part of any operating system. They provide an interface for user programs to request services from the operating system's kernel, which manages all the machine's hardware resources.

Modifying System Calls

In this section, I will describe the specific system calls that were modified and the reasons for choosing them. I will also explain testing and implementation.

First System Call: ObadahAbozaid5351127

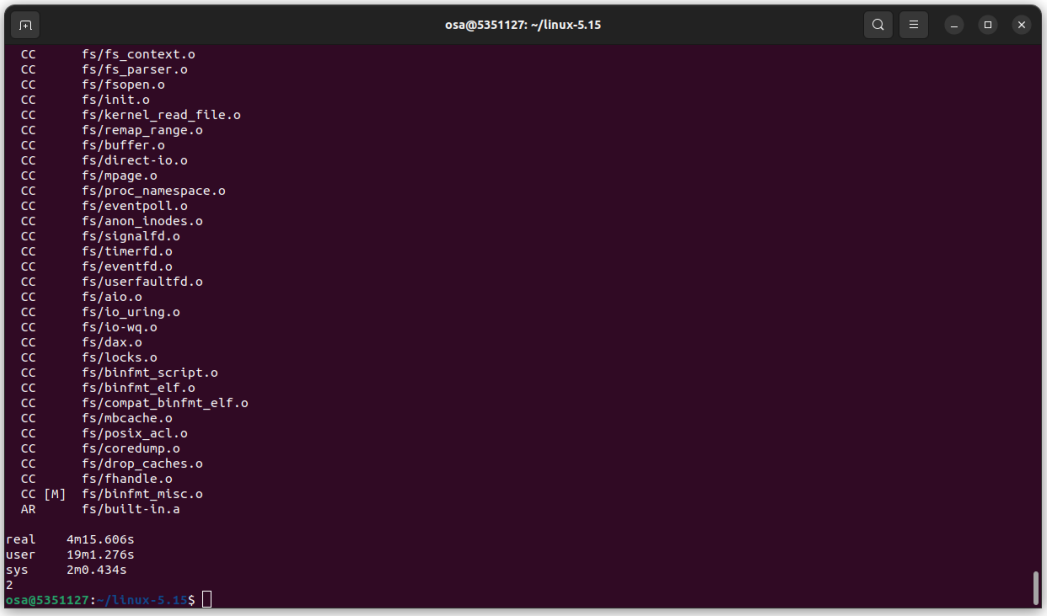
This system call takes no parameters and returns 0 on success. The purpose of this system call is to print a message to the kernel log, containing my name and student ID number. This system call is a simple example of how to create a custom system call and how to use the printk function to write to the kernel log.

To create this system call:

I created a new directory ObadahAbozaid5351127 under the kernel source tree root, and a new file ObadahAbozaid5351127.c in that directory.

I modified the provided code for my system call in ObadahAbozaid5351127.c, using the SYSCALL_DEFINE0 macro and the printk function.

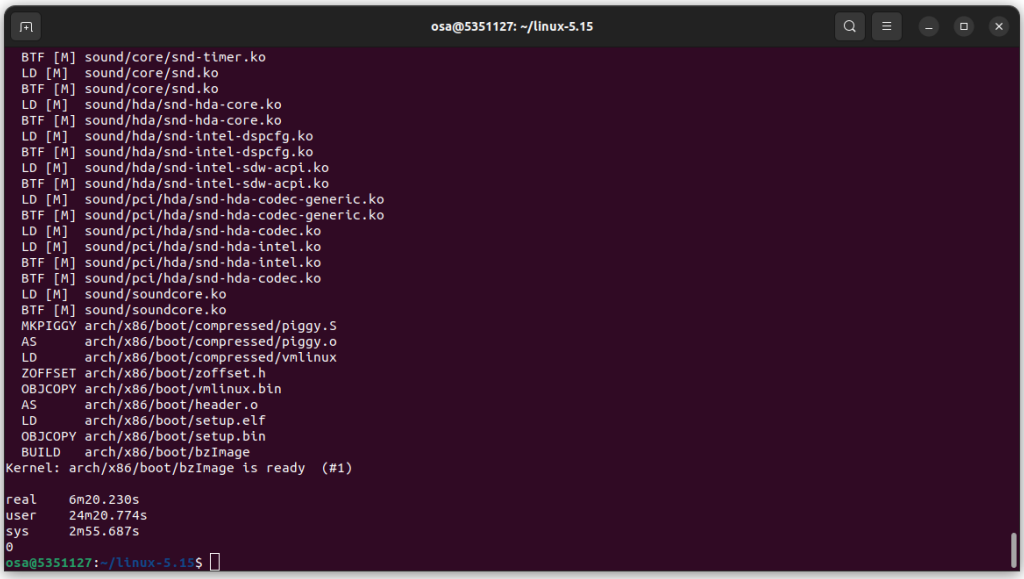
I had initially created a Makefile in the same directory, listing the .c file as an object file (.o).



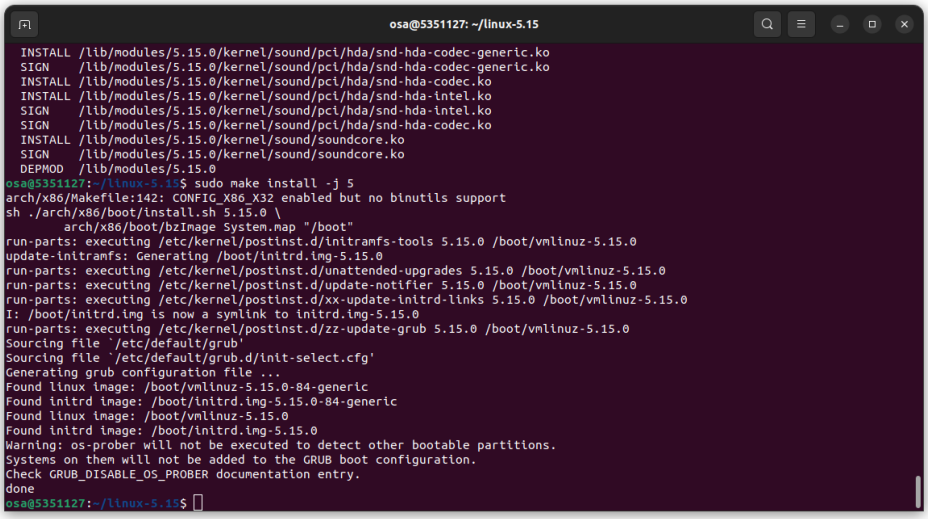
Kernel failed to compile

Shown above by the error code 2, the kernel failed to compile initially, as I later found that I misspelled Makefile as MakeFile. This was one of a few mistakes made along the way.

I added my directory to the kernel's Makefile, using the core-y += line, and declared my system call in the syscalls.h file, using the asmlinkage keyword and the long return type. I assigned a unique number, 449, to my system call in the syscall_64.tbl file, using the common qualifier and the sys_ prefix. I then successfully compiled and installed the new kernel, and then rebooted the system.

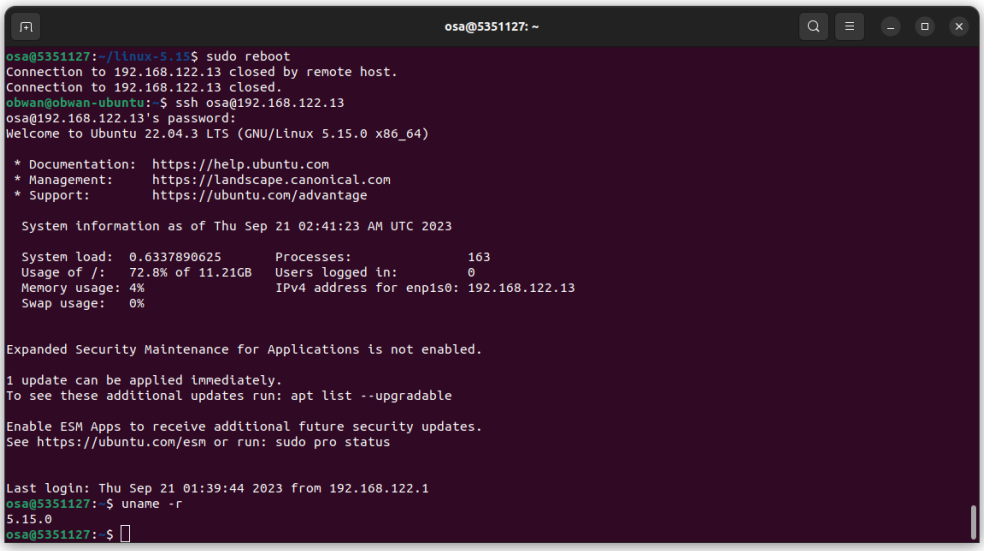


Kernel compiled successfully

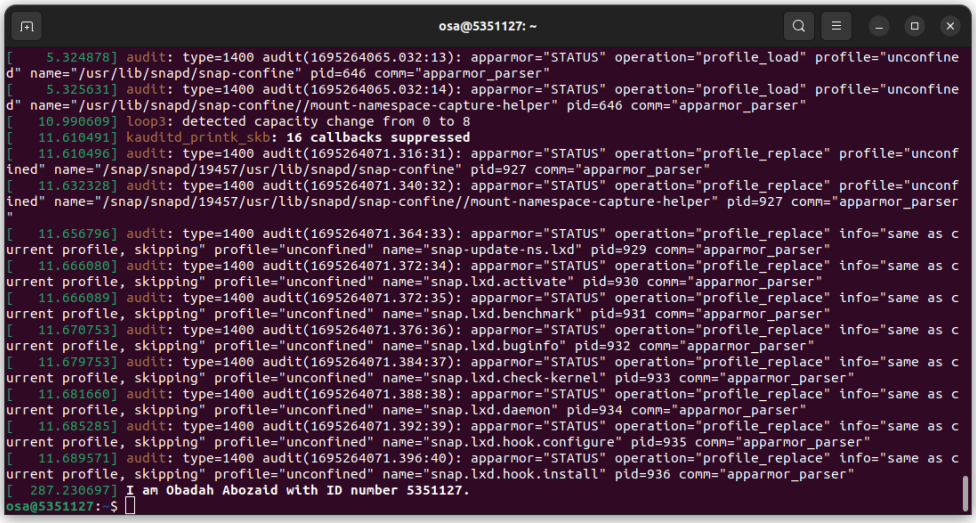


Kernel installed successfully

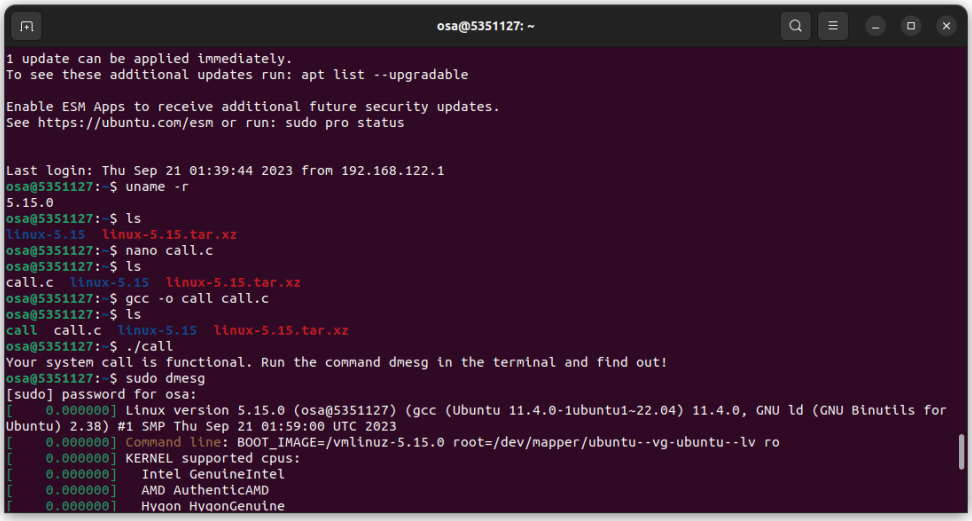
To test the system call, I modified the provided user program that calls the system call using the syscall function. I used the __NR_ ObadahAbozaid5351127 constant to refer to my system call number. I compiled and ran the program and checked the kernel log using the dmesg command. I verified that the message was printed correctly.



'uname -r' output shows newly modified kernel



System call shown in log with 'dmesg'



call program functionality

call.c:

```
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define __NR_ObadahAbozaid5351127 449

long ObadahAbozaid5351127_syscall(void)
{
  return syscall(__NR_ObadahAbozaid5351127);
}

int main(int argc, char *argv[])
{
  long activity;
```

```
  activity = ObadahAbozaid5351127_syscall();


  if(activity < 0)
  {
     perror("Your system call appears to have failed");
  }


  else
  {
     printf("Your system call is functional. Run the command dmesg in the terminal and find
out!\n");
  }
  return 0;
}
```

Second System Call: getprocessdetails

The second system call created is called getprocessdetails. This system call takes a pointer to a struct as a parameter and returns 0 on success or -1 on error. The purpose of this system call is to get and return some information about the calling process, such as its process id, static priority, and name. This information is stored in a struct that is defined in both user space and kernel space. This system call is a more advanced example of how to create a custom system call and how to use the current macro and the copy_to_user function to access and transfer process information.

I first defined a struct called process_info that can hold the process id, static priority, and name of the calling process. I used types such as pid_t and char[16] for these fields. Then I declared this struct in a new header file called process_info.h, which I placed in the include/linux/ directory. I also included the necessary header files for the types used in the struct, such as <linux/types.h>.

I included this header file in both the kernel code and the user program, so that they can share the same struct definition. I created a new directory getprocessdetails under the kernel source tree root, and a new file getprocessdetails.c in that directory.

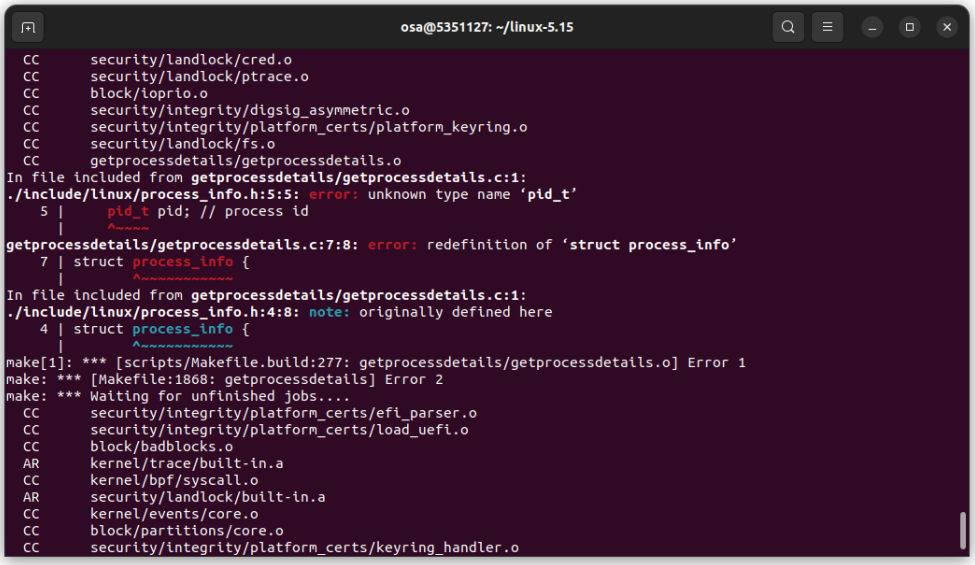
getprocessdetails.c:

```
#include <linux/process_info.h>

#include <linux/kernel.h>
```

```
#include <linux/syscalls.h>
#include <linux/sched.h>
#include <linux/uaccess.h>



SYSCALL_DEFINE1(getprocessdetails, struct process_info *, info)
{
    struct process_info kernel_info; // temporary struct in kernel space

    // get the current task's pid, static_prio, and comm
    kernel_info.pid = task_pid_nr(current);
    kernel_info.static_prio = current->static_prio;
    strncpy(kernel_info.comm, current->comm, 16);

    // copy the kernel_info to the user space pointer info
    if (copy_to_user(info, &kernel_info, sizeof(struct process_info))) {
        return -EFAULT; // return an error if copy fails
    }

    // print some information to the kernel log
    printk("The process %s with pid %d has static priority %d\n", kernel_info.comm,
kernel_info.pid, kernel_info.static_pr>

    return 0; // return success
}
```

I wrote the code for the system call in getprocessdetails.c, using the SYSCALL_DEFINE1 macro and the copy_to_user function. I also used the current macro to access the task_struct of the calling process, and retrieved its pid, static_prio, and comm fields. I stored these values in a temporary struct in kernel space, and then copied it to the user space pointer passed as a parameter. I also printed some information to the kernel log using the printk function.

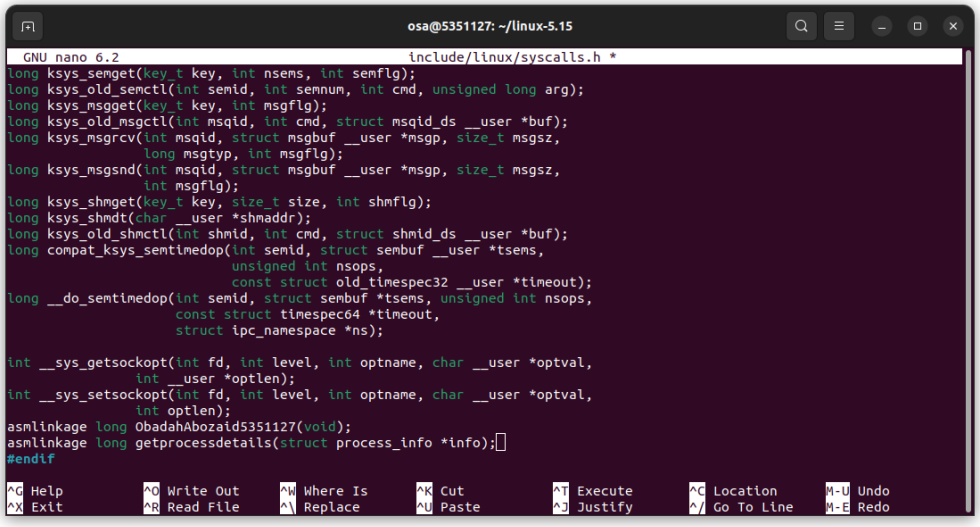
Initially however, the kernel did not compile correctly when implementing the syscall, because the type pid_t wasn't recognized in the process_info.h file. This type is defined in the linux/types.h header file, so I needed to include this header.



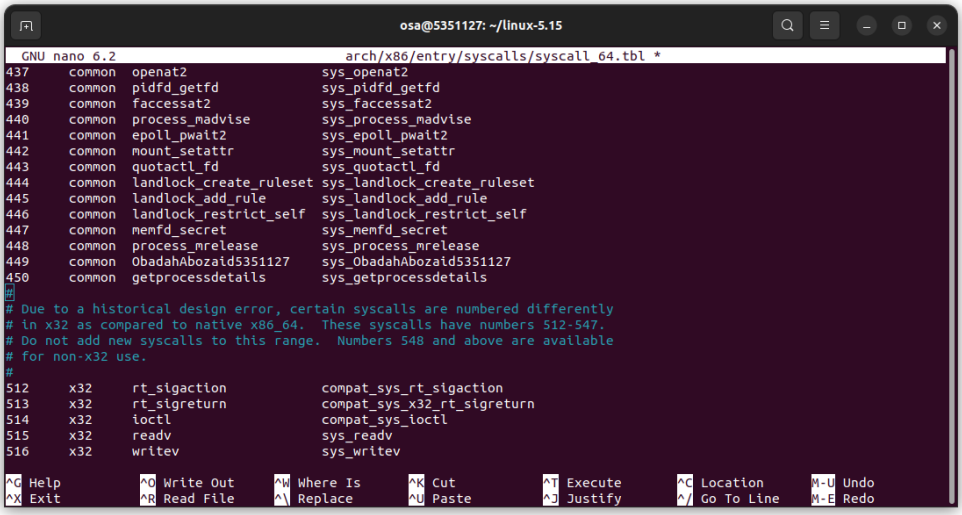
process_info.h error

I created a Makefile in the same directory, listing the .c file as an object file (.o), and added the directory to the kernel's Makefile, using the core-y += line.

I declared the system call in the syscalls.h file, using the asmlinkage keyword and the long return type, including the struct as a parameter type. I then assigned a unique number, 450, to the system call in the syscall_64.tbl file, using the common qualifier and the sys_ prefix.

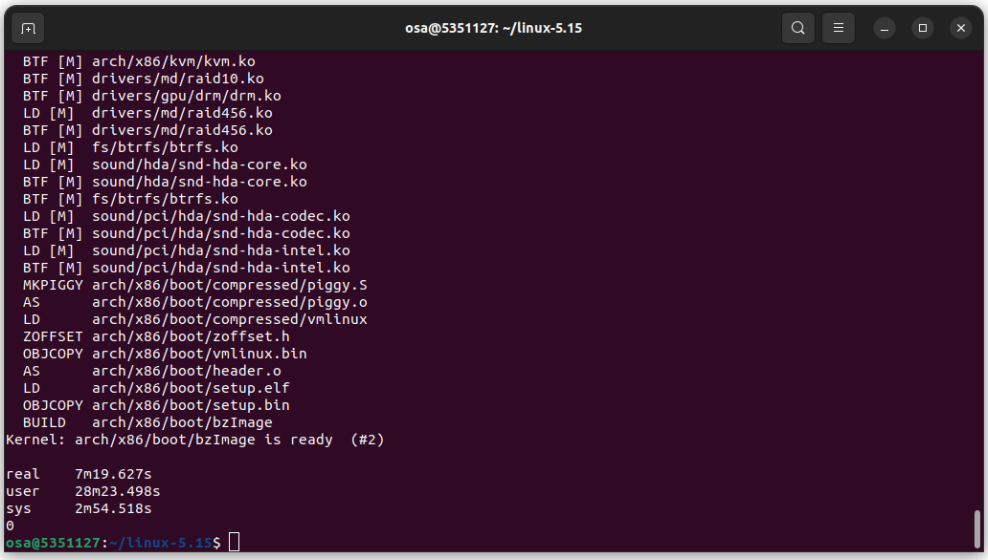


Adding to syscalls.h

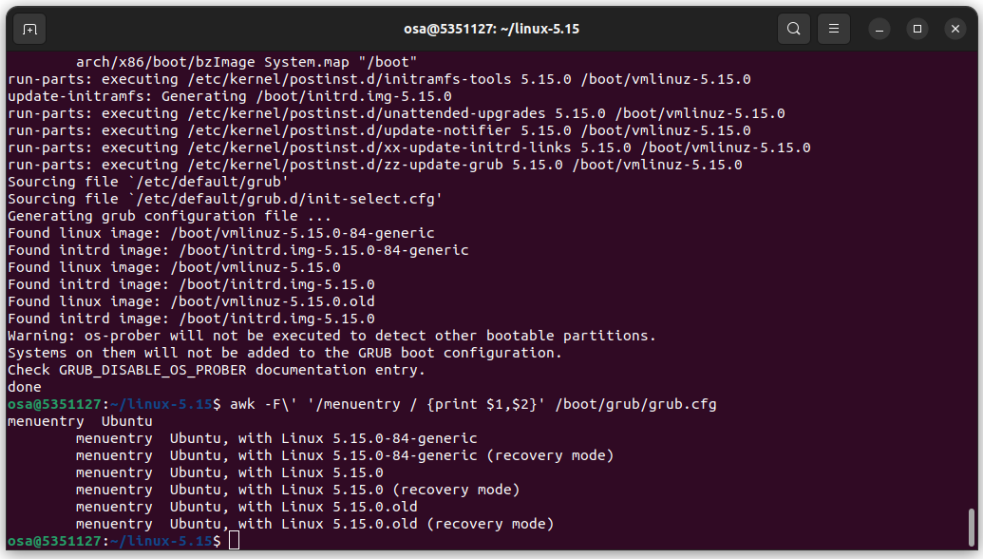


Adding to syscalls_64.tbl

I successfully compiled, installed the new kernel, and rebooted the system.



Successful kernel compilation



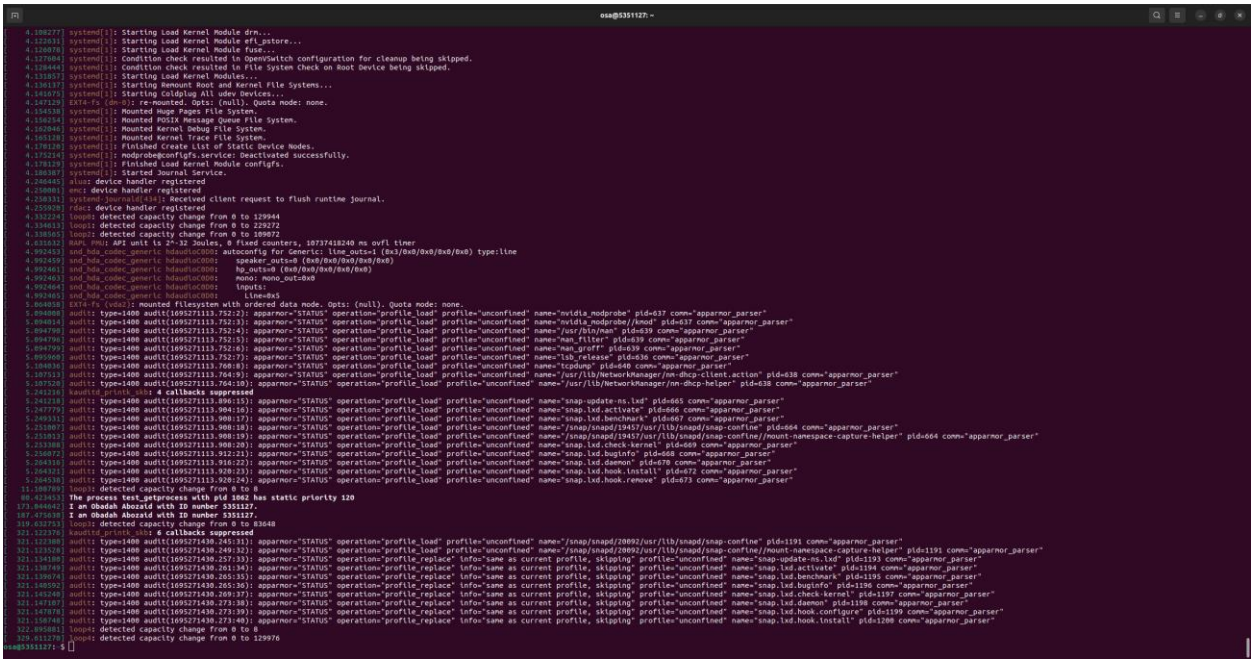
Installation of new kernel

To test the system call, I wrote a user program that calls the getprocessdetails system call using the syscall function. I used the same struct definition as in the kernel code and passed its address as a parameter. I also used an error-checking mechanism to handle any failures. I compiled and ran the program and printed the process details returned by the system call. I also checked the kernel log using the dmesg command, verifying that my custom system call worked correctly and returned identical values.

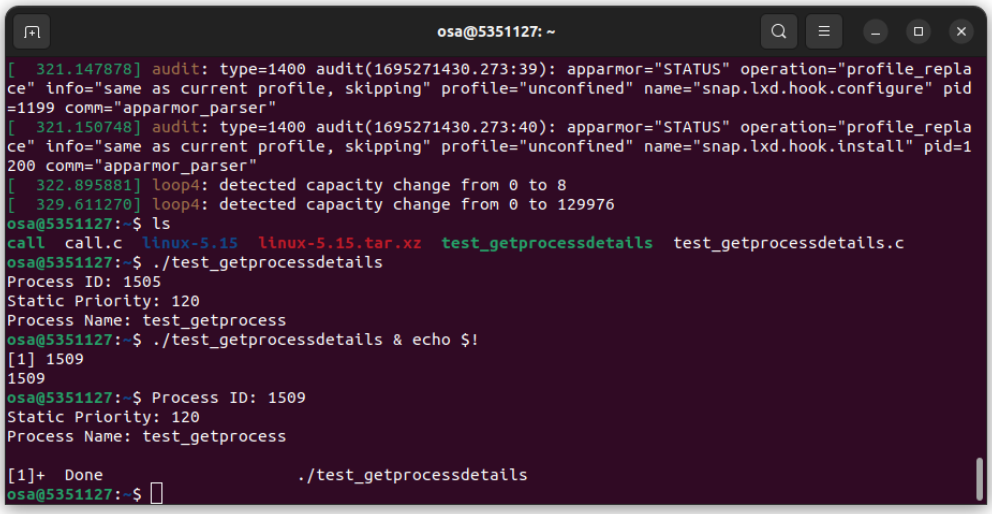
test_getprocessdetails.c:

```
#include <stdio.h>

#include <sys/syscall.h>

#include <unistd.h>


#define SYS_getprocessdetails 450


struct process_info {

    pid_t pid; // process id

    int static_prio; // static priority

    char comm[16]; // process name

};
```

```
int main() {
    struct process_info info;

    // Call the system call and check for errors
    if (syscall(SYS_getprocessdetails, &info) == -1) {
        perror("getprocessdetails");
        return 1;
    }

    // Print the process details
    printf("Process ID: %d\n", info.pid);
    printf("Static Priority: %d\n", info.static_prio);
    printf("Process Name: %s\n", info.comm);

    return 0;
}
```



'sudo dmesg' output



Final matching process IDs. Success.

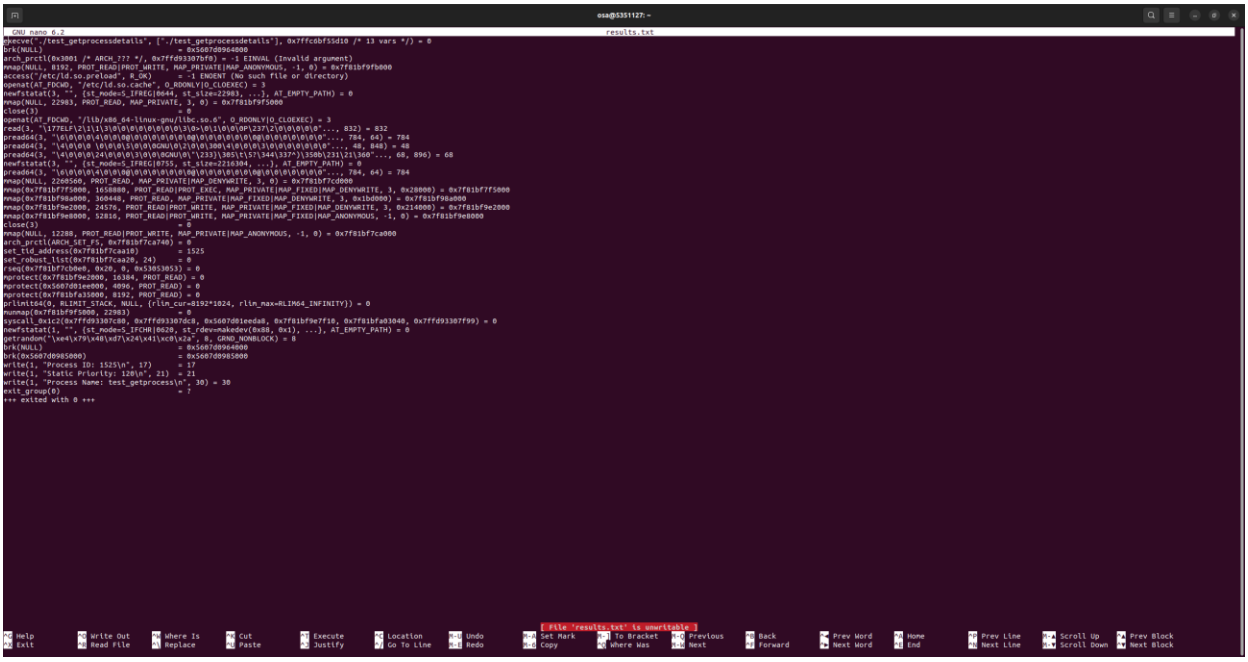
Tracing the System Calls

The strace command is a powerful command-line tool used in Linux for diagnostic, debugging, and instructional purposes. It intercepts and records the system calls which are called by a process and the signals which are received by a process. The strace command can be invoked by prefixing a command with strace, and it will print a list of system calls made by that program.

To trace the system calls made by test_getprocessdetails, I first compiled the test program with gcc:

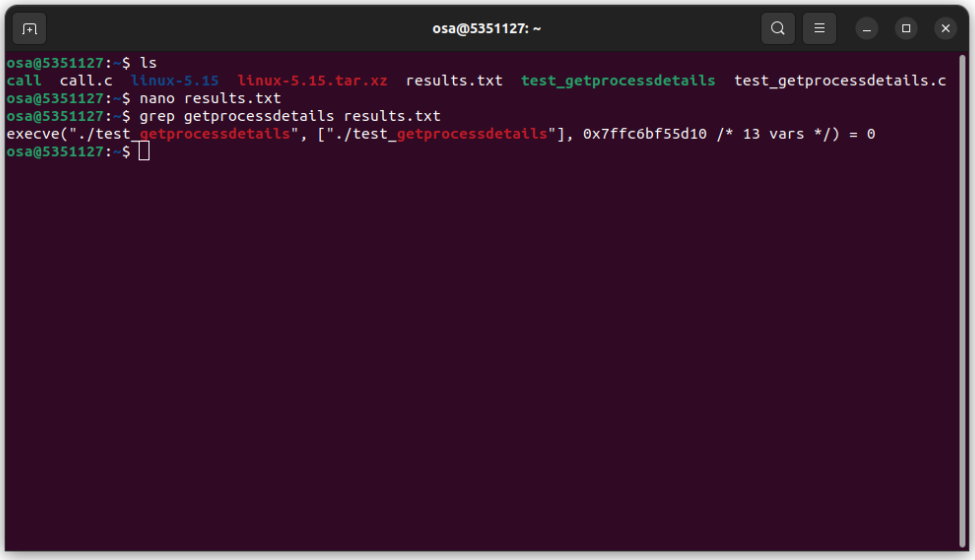
gcc -o test_getprocessdetails test_getprocessdetails.c

I ran my program with strace and redirected the output to a file:



‘strace -o results.txt ./test_getprocessdetails’

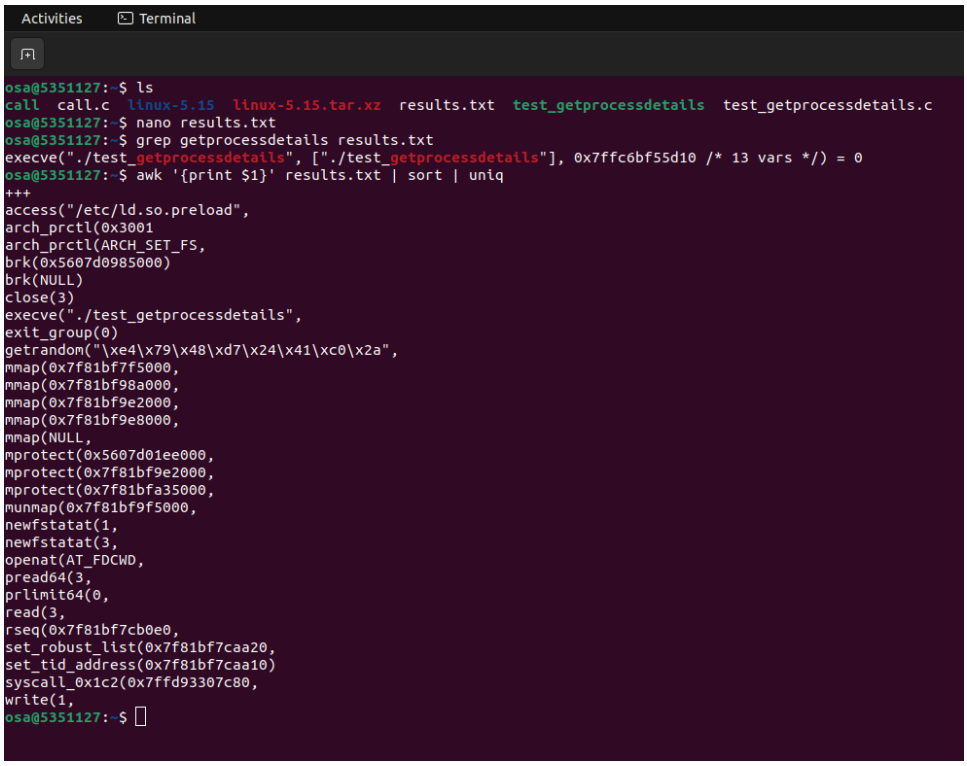
To find the entry corresponding to the getprocessdetails system call, I used the grep command:



‘’grep getprocessdetails output.txt’

This command searched for the string “getprocessdetails” in the results.txt file and printed any matching lines.

To see all the different system calls made by the program, I used:



'awk '{print $1}' output.txt | sort | uniq'

This command extracted the first word (the system call name) from each line in the results.txt file, sorted them, and removed any duplicates.

By using strace, I was able to see exactly what system calls my program was making, including the custom system call created. This helped to verify that the system call was working correctly and understand how a program interacts with the operating system.

Conclusions

To summarize, I explored the KVM and system call modifications in an Ubuntu Linux Server 22.04 LTS environment. I installed the KVM and created a virtual machine for the guest OS with no difficulty. I learned about the concept and importance of system calls in an operating system by creating two custom system calls: ObadahAbozaid5351127 and getprocessdetails. This was implemented and tested using C code and the strace tool.

Ilearned that system calls provide a way for user programs to request services from the operating system's kernel, which manages all of the hardware resources. By modifying system calls, developers can customize the behavior and performance of the system, as well as create new functionalities. However, modifying system calls also requires careful planning, testing, and debugging, as well as understanding the kernel source code and structure.