3. System Calls:

Brief Explanation

What are System Calls?

System calls are the fundamental interface between a process (arunning program in user space) and the operating system kernel(which runs in a protected, privileged kernel space). They are howuser programs request services that they cannot performthemselves because they require higher privileges, such asinteracting with hardware, managing files, creating or managingother processes, and network communication.

Below is simple C programming that demonstrating basic systemcalls.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

// System call to create a new process

pid_t pid = fork();

if (pid == -1) {

perror("Fork failed");

exit(EXIT_FAILURE);
```

```
} else if (pid == 0) {
// Child process
printf("Hello from child process!\n");
} else {
// Parent process
printf("Hello from parent process! Child PID: %d\n", pid)
return 0;

}
```

Explanation:

• fork() is a system call that creates a new process by duplicating the calling process.

• The return value of fork() helps identify whether we are in the parent or child process.

This example illustrates how system calls enable interaction with the operating system's process management functionalities.

Implementing system calls in an operating system like Elementary OS, which is based on Ubuntu, requires a good understanding of the Linux kernel and its architecture.

implementing system calls in a Linux-based operating system:

1. Set Up Your Environment:

• Install the necessary packages for kernel development:

2. Download the Kernel Source:

• Download the Linux kernel source code that matches your Elementary OS version. You can usually find it in /usr/src or download it from the kernel.org website.

3. Navigate to Kernel Source Directory:

cd /usr/src/linux-<version>

4. Choose a System Call Number:

• Each system call has a unique number. Choose an unused number from include/asm/unistd.h.

5. Implement Your System Call:

• Create a new C file for your system call in the kernel/ directory, e.g., my_syscall.c.

• Implement your system call function. For example:

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
SYSCALL_DEFINE0(my_syscall) {
printk(KERN_INFO "My syscall was called\n");
return 0;
```

6. Register Your System Call:

• Add your system call to the syscall table. This is usually located in arch/x86/entry/syscalls/syscall_64.tbl for x86_64 architectures.

• Add a line like this:

<syscall_number> common my_syscall sys_my_syscall

7. Compile the Kernel:

• Configure the kernel if needed:

make menuconfig

• Compile the kernel and modules:

make

sudo make modules_install

sudo make install

8. Update Bootloader:

• Update your bootloader (GRUB) if necessary and reboot the system.

9. Test Your System Call:

• Write a simple user-space program to test your new system call:

#include <stdio.h>

#include <unistd.h>

#include <sys/syscall.h>

```
#define __NR_my_syscall <syscall_number>
int main() {
long result = syscall(__NR_my_syscall);
printf("Result: %ld\n", result);
return 0;
}
```

• Compile and run your test program.

10. Debugging:

• Check the kernel logs using dmesg to see if your syscall is being invoked correctly.

Important Notes

• Modifying the kernel can lead to system instability. Always back up important data.

• Consider using a virtual machine or a separate partition for kernel development.

• Follow best practices for coding and testing to avoid introducing bugs into the kernel.

Conclusion

Implementing system calls in Elementary OS or any Linux-based OS requires careful manipulation of the kernel source code and understanding of how system calls work at a low level

Thank you for all!