Mehdi Maleki – OS Lab Project

Adding a System Call to xv6 RISC-V

Overview

This guide explains how to run the xv6 RISC-V operating system on Linux and add a new system call to it. xv6 is a simple Unix-like educational operating system developed at MIT.

A system call is a mechanism that allows user programs to request services from the operating system's kernel. It acts as an interface between user applications and the operating system, enabling programs to perform tasks such as reading and writing files, creating processes, and communicating with hardware.

In xv6, the system call mechanism works as follows:

1. **User Program Requests**: A user program invokes a system call by executing a special assembly instruction (**ecall** on RISC-V) to transition from user mode to kernel mode.
2. **Trap Handling**: The CPU switches to kernel mode and jumps to a predefined entry point in the kernel, known as the trap handler. In xv6, this is located in **kernel/trap.c**.
3. **System Call Dispatch**: The trap handler identifies the system call request by reading a specific register or memory location that holds the system call number. This number corresponds to a function in the kernel.
4. **Function Execution**: The kernel executes the corresponding function for the system call, performing the requested operation (e.g., reading a file).
5. **Return to User Mode**: Once the system call function completes, the kernel switches back to user mode and returns control to the user program, providing any results through registers or memory.

Impact of System Calls on the Operating System

System calls play a crucial role in the overall behavior and performance of an operating system. They serve as the primary interface between user applications and the kernel, enabling programs to request essential services such as file operations, process control, and communication.

1. **Resource Management**: System calls manage resources like CPU, memory, and I/O devices, ensuring efficient and fair allocation among processes. Mismanagement can lead to resource contention, bottlenecks, and degraded system performance.
2. **Security and Protection**: They enforce security and protection mechanisms, preventing unauthorized access to system resources. Each system call invocation

transitions the CPU from user mode to kernel mode, where the kernel performs necessary checks and validations.

3. **System Stability**: System calls contribute to system stability by handling errors and exceptions gracefully. Properly designed system calls ensure that erroneous or malicious user programs do not crash the entire system.

4. **Performance Overhead**: Frequent system call invocations can introduce performance overhead due to context switching between user mode and kernel mode. Optimizing system call paths and minimizing unnecessary calls are essential for maintaining high system performance.

Logging System Call Invocations

To analyze and improve the impact of system calls on the operating system, detailed logging of system call invocations, responses, and their effects is necessary. This involves:

1. **Tracking Invocation Details**: Log the time, process ID, system call number, and parameters for each invocation. This helps in identifying patterns and understanding how applications interact with the kernel.

2. **Recording Responses**: Capture the return values and error codes from system calls. This provides insights into the success or failure rates of various calls and can highlight potential issues or inefficiencies.

3. **Analyzing Impact**: Monitor the effects of system calls on system resources and performance metrics. For example, track changes in CPU usage, memory consumption, and I/O activity before and after system call execution.

4. **Auditing Security**: Log security-related events, such as permission checks and access violations, to ensure that the system's security policies are being enforced correctly.

Prerequisites

- Basic understanding of C programming
- Familiarity with operating system concepts
- A working xv6 development environment on RISC-V

Ensure you have the following installed on your Linux distribution:

- `git`

- `qemu`

- `gcc` (with RISC-V support)

- `make`

Setting Up xv6 RISC-V on Linux

1. **Clone the Repository**

```
git clone https://github.com/mit-pdos/xv6-riscv.git
cd xv6-riscv
```

2. **Install RISC-V Toolchain**

Follow the instructions here to install the RISC-V GNU toolchain. If you don't want to download a lot of files, installing the following is enough:

Debian or Ubuntu:

```
sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64 linux-gnu
```

Arch:

```
sudo pacman -S riscv64-linux-gnu-binutils riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb qemu-emulators-full
```

3. **Build the xv6 Kernel**

```
make
```

4. **Run xv6 using QEMU**

```
make
make qemu
```

Steps to Add a System Call

1. Define the System Call

First, decide on the name and functionality of the new system call. For this example, we add a simple system call named `sys_helloworld` that prints "Hello, world!" to the console.

2. Modify the System Call Number

Edit the `kernel/syscall.h` file to add a new system call number. Append the new system call number to the list of existing system calls.

```
//
...

#define SYS_helloworld  22 // Assign an unused number

...
//
```

3. Declare the System Call Function

Declare the new system call function in the `kernel/syscall.c` file.

```
extern uint64 sys_helloworld(void);
```

4. Implement the System Call Function

Implement the system call in a new function in `kernel/sysproc.c`.

```
uint64
sys_helloworld(void)
{
    printf("Hello, world!\n");
    return 0; // Return value to user program
}
```

5. Add System Call to syscalls Array

Add the system call to the syscalls array in `kernel/syscall.c`.

```
static uint64 (*syscalls[])(void) = {
[SYS_fork]        sys_fork,
[SYS_exit]        sys_exit,
[SYS_wait]        sys_wait,
[SYS_pipe]        sys_pipe,
[SYS_read]        sys_read,
[SYS_kill]        sys_kill,
[SYS_exec]        sys_exec,
[SYS_fstat]       sys_fstat,
[SYS_chdir]       sys_chdir,
[SYS_dup]         sys_dup,
[SYS_getpid]      sys_getpid,
[SYS_sbrk]        sys_sbrk,
[SYS_sleep]       sys_sleep,
[SYS_uptime]      sys_uptime,
[SYS_open]        sys_open,
[SYS_write]       sys_write,
[SYS_mknod]       sys_mknod,
[SYS_unlink]      sys_unlink,
[SYS_link]        sys_link,
[SYS_mkdir]       sys_mkdir,
[SYS_close]       sys_close,
[SYS_helloworld]  sys_helloworld, // Add this line
};
```

6. Update the User Space Interface

Add a user space interface for the system call in `user/user.h`.

```
int helloworld(void);
```

Implement the user space interface in `user/usys.pl`.

```
entry("helloworld")
```

7. Test the System Call

Create a user program to test the new system call. For example, add a new file `user/helloworld.c`.

```
#include "kernel/types.h"
#include "user/user.h"

int
main(void)
{
    helloworld();
    exit(0);
}
```

Add the new test program to the `Makefile` to compile it.

```
# In Makefile

...
U_PROGS=\
    $U_cat\
    $U_echo\
    $U_forktest\
    $U_grep\
    $U_helloworld\ # Add this line
...
```

8. Compile and Run xv6

Compile the xv6 kernel and run it.

```
make clean
make
make qemu
```

In the xv6 shell, run the `helloworld` program to see the output of the new system call.

```
$ helloworld
Hello, world!
```

Conclusion

System calls are vital for the interaction between user applications and the operating system kernel. They manage essential operations such as resource allocation, security enforcement, and error handling, directly impacting the system's performance and stability. By extending xv6 with new system calls, such as **sys_helloworld**, and implementing detailed logging, developers can gain deeper insights into system behavior and optimize performance. Properly tracking system call invocations and responses

enhances the ability to monitor, debug, and improve the operating system, ensuring efficient and secure operation.

In this project, the successful addition of a new system call and its thorough testing underscore the importance of understanding kernel-level programming and its implications on system behavior. The detailed logging of system calls provides a foundation for further analysis and optimization, contributing to more robust and efficient system design.

https://github.com/mosioc/xv6-System-Call-and-Driver/