System Calls

Kartik Gopalan

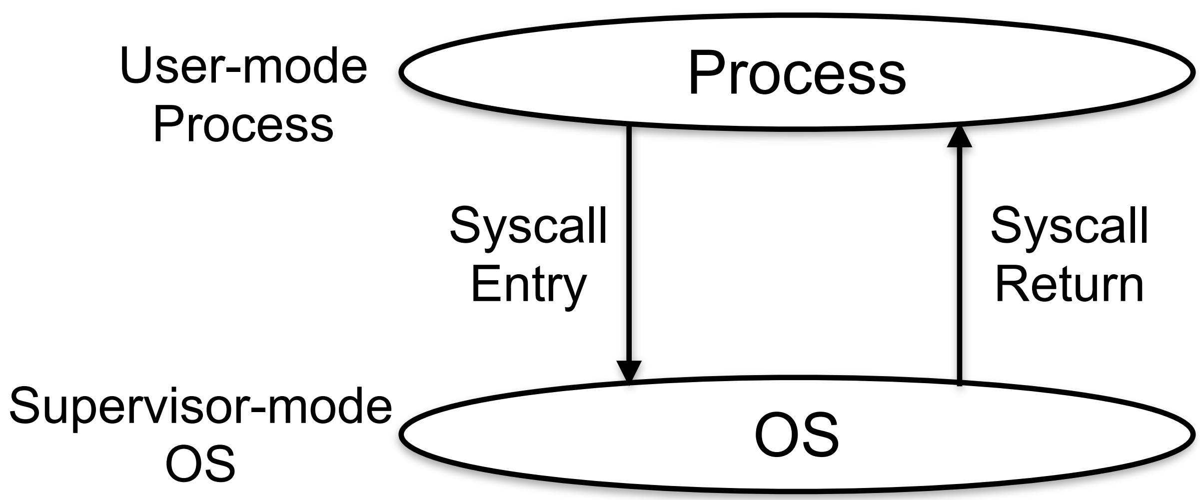
System Calls

- Modern CPUs support at least two levels of privileges:
  - User mode – application execute at this level

  - Supervisor mode – OS (kernel) code executes at this level

- System calls
  - Interface to allow User-level processes to safely invoke OS routines for privileged operations.

  - Safely transfer control from lower privilege level (user mode) to higher privilege level (supervisor mode), and back.

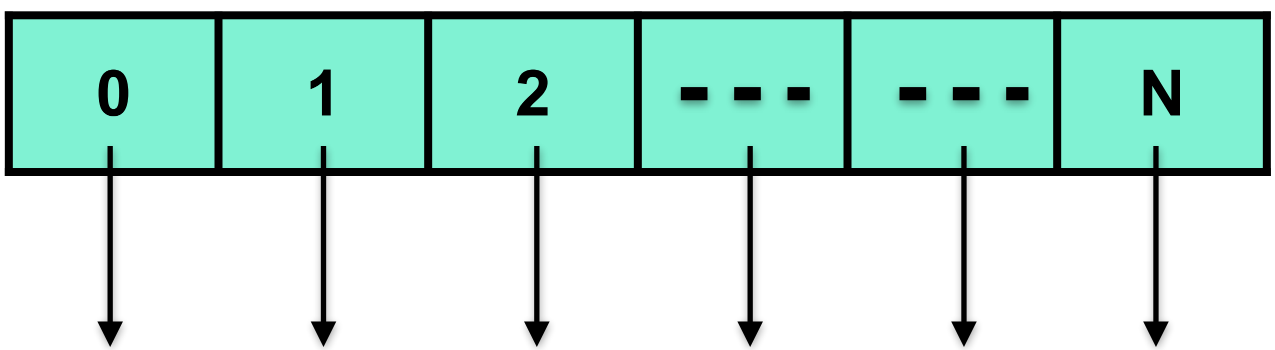


System Call table

- Protected entry points into the kernel for each system call
  - We don't want application to randomly jump into any part of the OS code.

- Syscall table is usually implemented as an array of function pointers, where each function implements one system call

- Syscall table is indexed via system call number



Steps in system call execution

User process	Invoke syscall using, say, SYSENTER instruction (arguments in registers/stack)
CPU	Switch CPU to supervisor mode. Jump to entry point in kernel.
Kernel	Save process state Lookup Syscall table. Invoke syscall.
Kernel	Optionally Block process if it needs to wait for I/O or other events. Return process to ready state when woken.
Kernel	Restore saved process state SYSEXIT
CPU	Switch CPU to user mode Return to user process
User Process	Return from system call. Continue

Syscall Usage

- To make it easier to invoke system calls, OS writers normally provide a library that sits between programs and system call interface.
    - Libc, glibc, etc.

- This library provides wrapper routines

- Wrappers hide the low-level details of
    - Preparing arguments
    - Passing arguments to kernel
    - Switching to supervisor mode
    - Fetching and returning results to application.

- Helps to reduce OS dependency and increase portability of programs.

Implementing System Calls

Steps in writing a system call

1. Create an entry for the system call in the kernel's syscall_table
   - User processes trapping to the kernel (through SYS_ENTER or int 0x80) find the syscall function by indexing into this table.

2. Write the system call code as a kernel function
   - Be careful when reading/writing to user-space
   - Use *copy_to_user()* or *copy_from_user()* routines.
     - These perform sanity checks.

3. Implement a user-level wrapper to invoke your system call
   - Hides the complexity of making a system call from user applications.
   - See *man syscall*

Step 1: Create a sys_call_table entry (for 64-bit x86 machines)

- Syscall table initialized in arch/x86/entry/syscall_64.c
- arch/x86/entry/syscalls/syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
...
309  common    getcpu                sys_getcpu
310  64        process_vm_readv      sys_process_vm_readv
311  64        process_vm_writev     sys_process_vm_writev
312  common    kcmp                  sys_kcmp
313  common    foo                   sys_foo
```

Step 2: Write the system call handler

- System call with no arguments and integer return value

```
SYSCALL_DEFINE0(foo){
    printk (KERN ALERT "sys_foo: pid is %d\n", current->pid);
    return current->pid;
}
```

- Syscall with one primitive argument

```
SYSCALL_DEFINE1(foo, int, arg){
    printk (KERN ALERT "sys_foo: Argument is %d\n", arg);
    return arg;
}
```

- To see system log:
  - dmesg
  - less /var/log/kern.log

Step 2: Write the system call handler

- Verifying argument passed by user space

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
        struct file * filp;
        struct files_struct *files = current->files;
        struct fdtable *fdt;
        spin_lock(&files->file_lock);
        fdt = files_fdtable(files);

        if (fd >= fdt->max_fds)
                goto out_unlock;
        filp = fdt->fd[fd];
        if (!filp)
                goto out_unlock;
        …
out_unlock:
        spin_unlock(&files->file_lock);
        return -EBADF;
}
```

- Call-by-reference argument
    - User-space pointer sent as argument.
    - Data to be copied back using the pointer.

```
SYSCALL_DEFINE3( read, unsigned int, fd,
                char __user *, buf,  size_t, count)
{
                …
    if( !access_ok( VERIFY_WRITE, buf, count))
                return –EFAULT;
                …
}
```

Step 3: Invoke syscall handler from user space

- Use the **syscall(...) library function**.
  - Do a "man syscall" for details.

- For instance, for a no-argument system call named foo(), you'll call
  - ret = syscall(__NR_sys_foo);
  - Assuming you've defined __NR_sys_foo earlier

- For a 1 argument system call named foo(arg), you call
  - ret = syscall(__NR_sys_foo, arg);

- and so on for 2, 3, 4 arguments etc.

- For this method, check
  - https://developer.ibm.com/tutorials/l-system-calls/

Step 3: Invoke your new handler from user space

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <linux/unistd.h>
// define the new syscall number. Standard syscalls are defined in linux/unistd.h
#define __NR_sys_foo 333
int main(void)
{
        int ret;
          while(1) {
         // making the system call
                        ret = syscall(__NR_sys_foo);
                        printf("ret = %d errno = %d\n", ret, errno);
                        sleep(1);
        }
         return 0;
}
```