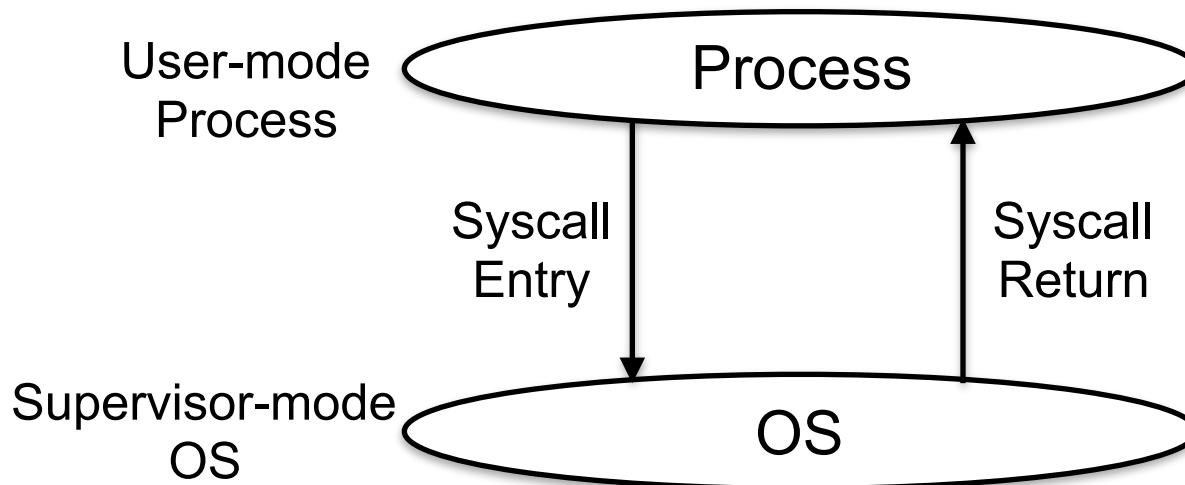


System Calls

Kartik Gopalan

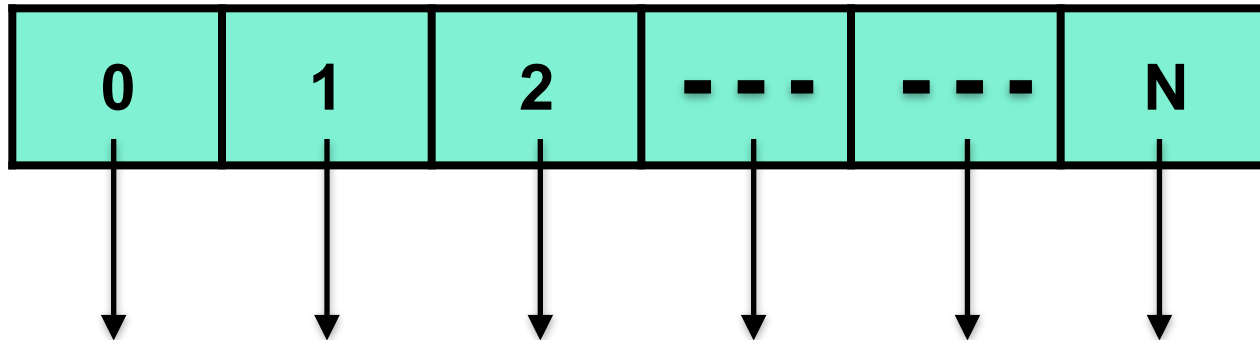
System Calls

- 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



System call invocation

1. System call is invoked via a special CPU instruction
 - Such as SYSENTER/int 0x80/lcall7/lcall27 etc.
 - The system call number and arguments passed via CPU registers and optionally stack.
2. CPU saves process execution state
3. CPU switches to higher privilege level
 - Jumps to an entry point in OS code.
4. OS indexes the system call table using the system call number
5. OS invokes the system call via a function pointer in the system call table.
 - For performance reasons, the system call usually executes in the execution context of the calling process, but in privileged mode.
 - Some operating systems may execute the system call in a separate execution context for better security.
6. If the syscall involves blocking I/O, the calling process may block while the I/O completes.
7. When syscall completes, the calling process is moved to ready state.
8. The saved process state is restored
9. Processor switches back to lower (user) privilege level using SYSEXIT/iret instructions
10. Process returns from the system call and continues.

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

- 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.
- 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.
- Generate/Use a user-level system call stub
 - 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)

- arch/x86/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

```
asmlinkage int sys_foo(void) {  
    printk (KERN_ALERT "I am foo. UID is %d\n", current->uid);  
    return current->uid;  
}
```

- Syscall with one primitive argument

```
asmlinkage int sys_foo(int arg) {  
    printk (KERN_ALERT "This is foo. Argument is %d\n", arg);  
    return arg;  
}
```

- To see log: dmesg, /var/log/kern.log

Step 2: Write the system call handler

(cont...)

- Verifying argument passed by user space

```
asmlinkage long sys_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.

```
asmlinkage ssize_t sys_read ( unsigned int fd,
                             char __user * buf, size_t count)
{
    ...

    if( !access_ok( VERIFY_WRITE, buf,
count))

        return -EFAULT;

    ...
}
```

Example syscall implementation

```
asm linkage int sys_foo(void) {  
    static int count = 0;  
    printk(KERN_ALERT "Hello World! %d\n", count++);  
    return -EFAULT; // what happens to this return value?  
}
```

```
EXPORT_SYMBOL(sys_foo);
```

Step 3: Invoke your new handler with syscall

- 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 with syscall (cont...

```
#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;
}
```