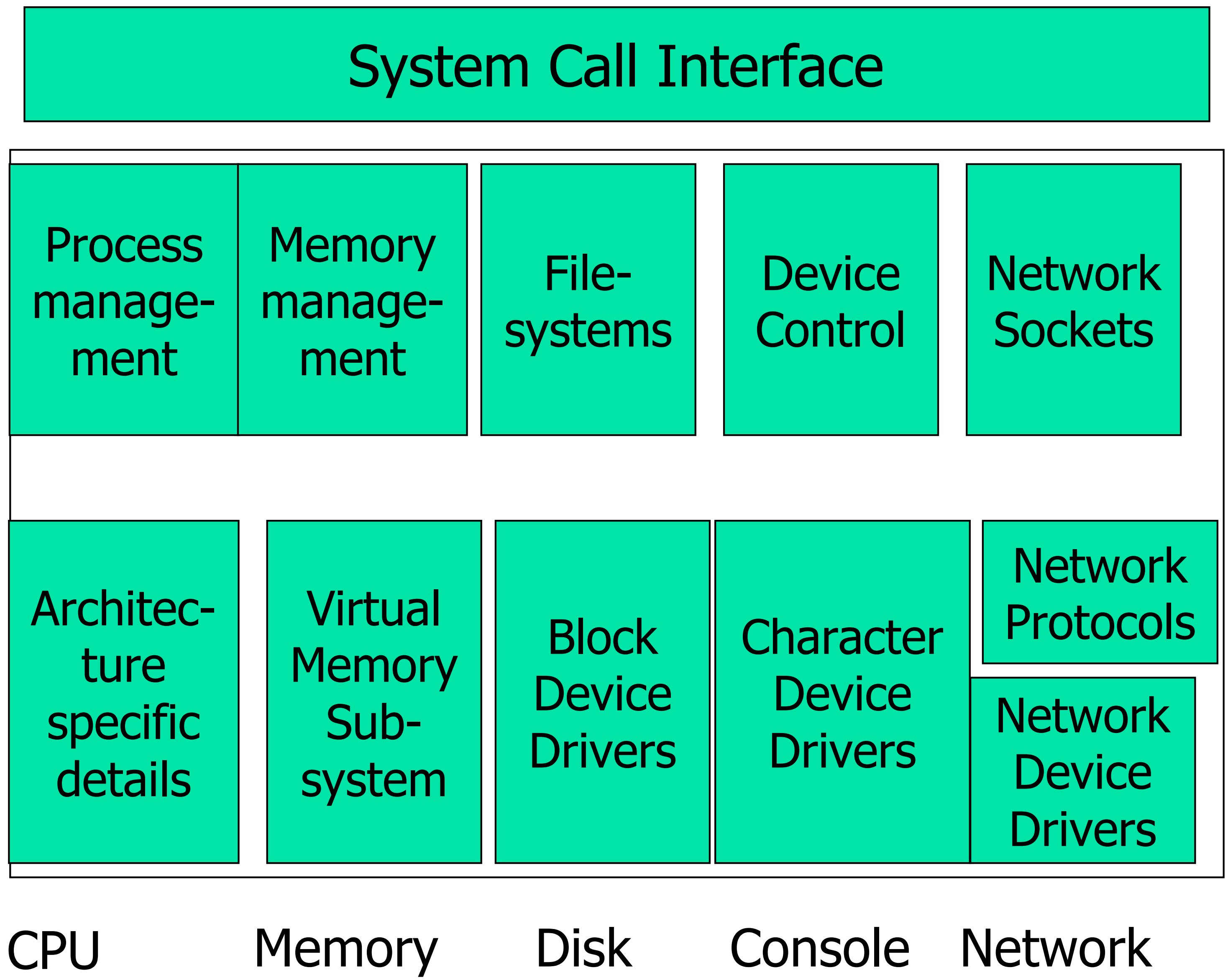
System Calls

Kartik Gopalan

Simplified Organization of Linux Kernel



System Calls

- Operating systems typically support two levels of privileges:
  - User mode – application execute at this level
  - Supervisor mode – OS (kernel) code executes at this level

- Applications need to call OS routines to request privileged operations.

- System calls
  - Safely transfer control from lower privilege level (user mode) to higher privilege level (supervisor mode).
  - Examples: open, read, write, close, wait, exec, fork, kill

- Kernel can tightly control entry points for the application into the OS.
  - Application can't randomly jump into any part of the OS code.

How system call works

1. User process invokes a system call, which invokes a library wrapper routine for the system call.
2. Library routine triggers a special CPU instruction or a software trap
   - SYSENTER, int 0x80, lcall7, lcall27
   - System call number and arguments stored in registers and, if needed, stack.
3. Trap to kernel entry code.
4. Save process state (program counter, registers etc)
5. Switch CPU to higher (kernel) privilege level
6. Determine the syscall being invoked
   - Check syscall number which is passed via registers
7. Use syscall number to index into a syscall table
   - Syscall table = An array of pointers to system call routines in kernel
8. Execute the system call routine.
   - System call executes in the context of the invoking process.
9. System call can possibly block
   - Which blocks the process that invoked the system call.
10. System call completes
11. Restore user-level process state (saved in Step 4).
12. Switch CPU back to lower (user) privilege level
13. Return from system call and continue process execution.

Library wrappers around system calls

- 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.

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)

File: 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

```
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 OR the end of /var/log/messages

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

```
asmlinkage 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
  - http://www.ibm.com/developerworks/linux/library/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;
}
```