

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 4

Modifying the kernel

In the last tutorial, we started our quest to get close to the Linux kernel. We have tried upgrading a kernel. We will build on that knowledge, and actually modify the kernel this week. To get started, we will do something extremely simple: **to add a trivial system call**. We will get some real work done in the kernel.

1. The kernel source

The kernel is by no means small: the 2.2.19 kernel is more than 73M bytes in size. As much as 41M of the code is lying in device drivers, while filesystems and network code constitutes another 10M. The remaining 20M bytes of code includes 15M of architecture dependent code (around 2M for each of the 9 supported architectures: Intel x86, Sun SPARC, 64-bit SPARC, IBM S390 main-frame, Motorola PowerPC, Motorola 680x0, DEC Alpha, the MIPS used in early DEC machines, and the low-power RISC called ARM). Most of the remaining 5M is considered the “core” kernel code. This core is small enough that Linus, the creator and primary maintainer of Linux, can look after its every line.

The code is separated into the following top-level directories.

- **init/**: kernel initialization code. This includes all functions to boot up the kernel, in particular **start_kernel()** which is the first function executed by the kernel.
- **include/**: function prototypes of kernel-related functions like memory management, and generic functions like string handling. The directory includes some architecture dependent files in **include/asm-xxx/** subdirectories, which hide the code differing among architectures. The currently used architecture is available in the **include/asm/** subdirectory.
- **arch/**: functions that differs among architectures, like how registers are used, how memory mapping works, etc.
- **kernel/**: the central kernel code. Most system calls are defined here, as well as some kernel mechanisms like timer management, scheduler and I/O handling.
- **mm/**: the Linux memory manager, supporting the memory allocation strategy and virtual memory system of Linux.
- **driver/**: various device drivers. This is where specific ways to communicate with individual hardware devices like the disk, network card, display card, sound card, serial devices, mouses, USB bus, video devices, etc. are implemented.
- **net/**: various networking protocols, including Ethernet, Novell IPX, TCP/IP, etc. The network devices are implemented in the **driver/** directory.
- **fs/**: various filesystems. They all share the same interface, called VFS (virtual filesystem).
- **lib/**: the implementation of the generic functions mentioned in “include/” above.
- **modules/**: this is where compiled modules are temporarily stored, to be installed into `/lib/module` for use by a running kernel.

2. Kernel entry point: `entry.S`

The list of system calls is defined in a file called `entry.S` in the architecture dependent directory `arch/xxx/kernel`. Note the file name ends not in `.c` but in `.S`, meaning that the file is written

in assembly language, not C. The file contains an entry called the system call table:

```
.data
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall) /* 0 */
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    ...
    .long SYMBOL_NAME(sys_vfork)          /* 190 */
    .rept NR_syscalls-190
        .long SYMBOL_NAME(sys_ni_syscall)
```

To add a system call, one adds an entry at the end of the list, just before the `.rept` line.

The `.rept` lines means that `NR_syscalls-190` entries are added, all containing the address of `sys_ni_syscall`. `NR_syscalls` is a constant, with value 256. Thus there are 256 system calls, some of them containing `sys_ni_syscall`, which just signify the function is Not Implemented. Later we will use these entries to add new system call dynamically.

3. System calls: kernel code

The list of system call are documented in the appendix A of the book Linux Kernel Internals. One simpler example is in `kernel/sched.c`:

```
asmlinkage int sys_getuid(void)
{
    return current->uid;
}
```

The *asmlinkage* macro illustrates how the kernel handle architecture differences. In most architectures, normal function calls are used for the system call handler to call individual functions. However, the 80x86 architecture has a more efficient parameter passing mechanism that can be used for small function calls like this. The make use of this, the macro *asmlinkage* is defined so that it expands to nothing for most architecture, and to a GNU C directive that specify the more efficient parameter passing mechanism for 80x86.

The above code never returns an error. In case a system call function needs to communicate an error, it returns a negative number, which absolute value is the error code. The file `asm/errno.h` defines a list of error codes.

In order to add a new system call, one can modify an existing file to contain another `sys_XXX` function. But it is probably better not to touch existing files, but instead to add a new one in an existing directory, say `kernel/`. To allow `make zImage` to find your new file, edit the `Makefile` of the directory, locating the line containing the list of object (`.o`) files to generate, and add your own object file name into it.

4. Functions in the kernel

As we have mentioned, C library functions are not available in the kernel. For instance, *printf* cannot be used for printing a message. However, many of them have substitute as utility

functions in `libs/` or `kernel/`. E.g., you can use *printk* to print a message from the kernel to the console. Again, the Linux Kernel Internals book contains an appendix containing them.

5. System calls: libc glue code

Once you have implemented a system call, user code can call them through the system call interface. But instead of writing assembly code like `int $0x80` by yourselves, you can use the macros `_syscallN` (*N* is the number of arguments) defined by `<syscall.h>` to build a C function that makes the system call. This is how the C library implements `getpid()`:

```
#define __NR_getpid 20 /* getpid is the 20th system call in syscall table */
_syscall0(int, getpid) /* Now getpid is a function that makes the syscall */
```

If the system call returns an error code (i.e., a small negative number), the C function returns -1, assigning the absolute value of the error code to the global variable `errno`. Otherwise, the value returned by the system call is propagated to the caller of C function.

So much for now.

Happy kernel hacking!