

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 3

Modifying the kernel

This week we will start our journey to the kernel. To get started, we will do something extremely simple: **to add a trivial system call**. We will soon get some real work done in the kernel. Before we really start the modification, let's see some basic information useful for getting the work done.

1. The kernel source

The kernel is by no means small: the 2.4.18 kernel that we will use is 24M bytes in size even when compressed, and when decompressed it is of size 143M. As much as 72M of it is are device drivers, while filesystems and network code constitutes another 23M. The remaining 48M bytes of code includes 40M of architecture dependent code (around 2M for each of the 15 supported architectures: Intel x86, Intel IA-64, Sun SPARC, 64-bit SPARC, IBM S390 and S390X main-frame, Motorola PowerPC, Motorola 680x0, DEC Alpha, the MIPS used in early DEC machines, 64-bit MIPS, the RISC processor of HP (HPPA), the Hitachi SuperH usually used as game console, the low-power RISC called ARM, and an embedded CPU of Axis Communications called ETRAX). Documentations and build system constitutes another 6M. The remaining 2M 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 build system link the **asm-xxx** directory of the currently used architecture as **include/asm**, so you will usually see include files of the form **<asm/...>**.
- **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.

2. Kernel entry points: `entry.S`

When the kernel is invoked via system call, interrupt and traps, some code is needed to save the previous state of the processor. The state needs to be restored before leaving the kernel. They are done by functions in the architecture dependent file `arch/xxx/kernel/entry.S`. Note the file name ends not in `.c` but in `.S`, meaning that the file is written in assembly language, not C. We are primarily interested in the list of system calls there:

```
.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_ni_syscall) /* 237 reserved for fremovexattr */
    .rept NR_syscalls-(.-sys_call_table)/4
        .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` line adds some entries to the table all containing the address of `sys_ni_syscall`. `NR_syscalls` is a constant, with value 256. Thus this fills the table to 256 entries, some of them containing `sys_ni_syscall`, signifying the function is Not Implemented. Later we will use these entries to add new system calls dynamically.

3. System calls: kernel code

Most system calls are implemented in the `kernel`, `mm` and `net` directories. One simpler example is the `get-UID` call in `kernel/timer.c`:

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

The *asmlinkage* macro is defined in `linkage.h` of the `include/linux` directory. It adds “**extern "C"**” if being compiled by a C++ compiler, and adds “`__attribute__((regparm(0)))`” if it is compiled for a computer like 80386, Pentium, etc (so that the first argument is passed in a register rather than on the stack). For our purpose, we always add it when defining a system call.

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. Instead, add a new one in an existing directory, say `kernel/`. To allow the kernel build system to find your new file, edit the `Makefile` of the directory, locating the line beginning with `obj-y`, and add your own object file name into it.

4. Functions in the kernel

As we have mentioned in the first tutorial, 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 (but not to standard output, so you won't see it in the X-window terminal). Many of these functions are declared in the include file at `include/linux/kernel.h`, so it is a good idea to have a quick look at it.

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!