

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 4

Modifying the kernel

This week we will start our journey into the kernel. As a start, 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 about the kernel.

1. The kernel source

The kernel is by no means small: the Redhat 2.4.20 kernel that we will use is 35MiB bytes in size even when compressed, and when decompressed it occupies 183MiB. Although it is really large, most of it consists of device driver, filesystems, network, and code to cope with different architectures. The “core” kernel, i.e., system calls and resource management code is merely around 3.5MiB, which is small enough that Linus Torvalds (the creator and primary maintainer of the development versions of Linux) or another maintainer (like Marcelo Tosatti, the current maintainer of the 2.4 kernel) can look after its every line.

Once you expand the kernel source archive, you get a directory that contains the whole kernel. We call this the **top level kernel source directory**. For convenience, we will denote this as **<kroot>**. It consists of the following sub-directories:

- **<kroot>/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.
- **<kroot>/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/...>**.
- **<kroot>/arch**: functions that differs among architectures, like how registers are used, how memory mapping works, etc.
- **<kroot>/kernel**: the core kernel code. Most system calls are defined here, as well as some kernel mechanisms like timer management, scheduler and I/O handling.
- **<kroot>/mm**: the Linux memory manager, supporting the memory allocation strategy and virtual memory system of Linux.
- **<kroot>/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.
- **<kroot>/net**: various networking protocols, including Ethernet, Novell IPX, TCP/IP, etc. The network devices (“network cards”) are implemented in the driver/ directory.
- **<kroot>/fs**: various filesystems. They all share the same interface, called VFS (virtual filesystem).
- **<kroot>/lib**: the implementation of the generic functions mentioned in “<kroot>/include” described above.

2. Kernel entry points: `entry.S`

When the kernel is invoked via system calls, interrupts 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 `<kroot>/arch/i386/kernel/entry.S`. Note that the filename 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_set_tid_address)

    .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.

3. System calls: kernel code

Most system calls are implemented in the `kernel`, `mm` and `net` directories. One simpler example is the `getuid` system call in `<kroot>/kernel/timer.c`:

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

The *asmlinkage* macro is used for platform-dependent behaviour of system calls. (In i386, it makes sure that function arguments are passed on stack rather than registers.)

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**. E.g., to signal an error ESRCH (meaning “process not found”, with code 3), one should return `-ESRCH` to indicate the error. The file `<kroot>/include/asm/errno.h` defines a list of error codes. This return value will be interpreted by the C library so that its absolute value is stored in the *errno* variable.

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 `<kroot>/kernel/`. To allow the kernel build system to find your new file, edit the `Makefile` of the directory, locate the line beginning with `obj-y`, and append the name of your own object file to it. Note that the **object** filename (ending with `.o`) is appended, not the source filename (ending with `.c`). The build system will automatically find the right source file to compile.

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 `<kroot>/libs/` or `<kroot>/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 `<kroot>/include/linux/kernel.h`, so it is a good idea to have a quick look at it.

5. Making system calls in user programs

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 in `<syscall.h>` to build a C function that makes the system call. For example, this is how the C library implements `getpid()`:

```
#include <syscall.h>
#include <errno.h>
...
#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 */
```

(Note that we have two underscores before NR, and one after.) It is important to understand that the above does not call a system call. Instead, it **creates a function** that would make the call if being called. For example, the above `_syscall0` macro will be expanded to something like the following:

```
int getpid(void) {
    long __res;
    __asm__ volatile ("int 0x80" : "=a" (__res) : "0" (__NR_getpid));
    if ((unsigned long)(__res) >= (unsigned long)(-125)) {
        errno = -(__res);
        __res = -1;
    }
    return (int)(__res);
}
```

So from now on, we have a function `getpid()` that will do the clumsy “int 0x80” for us to call the system call. Also note how the error code returned from the kernel is stored into the *errno* variable, and how the `__NR_getpid` constant is being used in the system call.