# Laboratory on Linux Kernel Haking

## OS - Operating System

Simone Rossi

December 1, 2016

| | |
|---|---|
| Date Performed: | December 1, 2016 |
| Partners: | Simone Rossi |

## 1 Compiling the Linux kernel

To analyze thet version of the kernel currently under execution, the command to be typed is `uname -vr`[1].

From the source code of `linux-2.4`, I tried to remove the support for the Ext3 filesystem. The new kernel has been compiled (computing all the dependencies and compiling all the sources) and mount using `lilo`.

Of course now, without the support for this version of filesystem, the kernel could not correctly load and mount the root filesystem; as conseguence, it entered in the kernel panic state (with error 19) with no possibilities to be recovered.

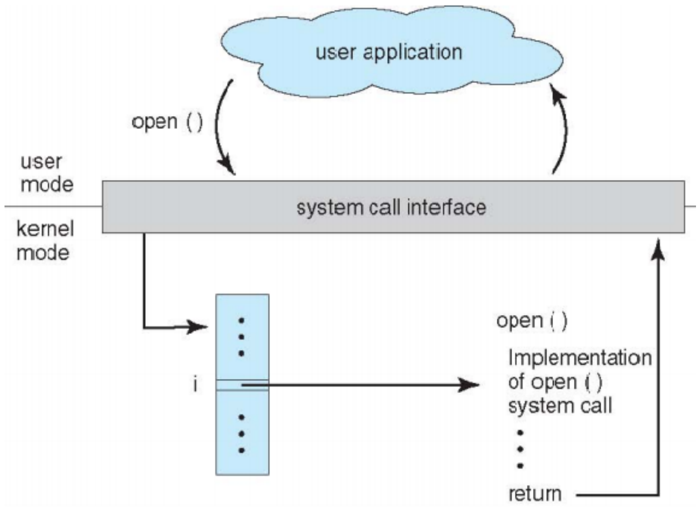
[1] The command returns both the kernel version and release

```
Loading BusLogic.o module
/lib/BusLogic.o: kernel-module version mismatch
        /lib/BusLogic.o was compiled for kernel version 2.4.20-8
        while this kernel is version 2.4.28.
ERROR: /bin/insmod exited abnormally!
Loading jbd.o module
/lib/jbd.o: kernel-module version mismatch
        /lib/jbd.o was compiled for kernel version 2.4.20-8
        while this kernel is version 2.4.28.
ERROR: /bin/insmod exited abnormally!
Loading ext3.o module
/lib/ext3.o: kernel-module version mismatch
        /lib/ext3.o was compiled for kernel version 2.4.20-8
        while this kernel is version 2.4.28.
ERROR: /bin/insmod exited abnormally!
Mounting /proc filesystem
Creating block devices
Creating root device
Mounting root filesystem
mount: error 19 mounting ext3
pivotroot: pivot_root(/sysroot,/sysroot/initrd) failed: 2
umount /initrd/proc failed: 2
Freeing unused kernel memory: 144k freed
Kernel panic: No init found.  Try passing init= option to kernel.
```

# 2 Adding new system calls to Linux

The man page of Linux describes the system call as a fundamental interface between an application and the Linux kernel itself. [...] Usually they are not invoked directly but rather via wrapper functions.

The system calls deal with low aspects of the operating systems, while providing high-level Application Programming Interface (API), allowing the user to work trasparantelly without dealing with hardware - software integretion. Let's assume the user application calls a library function which actually during its execution makes a system call to the kernel. The name of the system call is used as offset in a system call table and an interrupt is raised.




As explained in the document, to add new system calls to the kernel, two files should be modified.

`include/asm-i386/unistd.h` contains the system call numbers for the system call table and seven different template prototype for the function (with different numbers of parameters).

```
#define __NR_sched_setaffinity  241
#define __NR_sched_getaffinity  242
#define __NR_set_thread_area    243
#define __NR_get_thread_area    244
#define __NR_io_setup           245
#define __NR_io_destroy         246
#define __NR_io_getevents       247
#define __NR_io_submit          248
#define __NR_io_cancel          249
#define __NR_alloc_hugepages    250
#define __NR_free_hugepages     251
#define __NR_exit_group         252
#define __NR_kernelprint        253
#define __NR_rot13              254
/* user-visible error numbers are in the range -1 - -124: see <asm-i386/errno.h>
 */

#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
```

`arch/i386/kernel/entry.S` contains the system call low level handling routines and the link between a system call and a entry of the systel call table.

```
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_set_thread_area */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_get_thread_area */
.long SYMBOL_NAME(sys_ni_syscall)       /* 245 sys_io_setup */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_io_destroy */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_io_getevents */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_io_submit */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_io_cancel */
.long SYMBOL_NAME(sys_ni_syscall)       /* 250 sys_alloc_hugepages */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_free_hugepages */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_exit_group */
.long SYMBOL_NAME(sys_kernelprint)      /* <- Simo sys_lookup_dcookie */
.long SYMBOL_NAME(sys_rot13)            /* <- Simo sys_epoll_create */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_epoll_ctl 255 */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_epoll_wait */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_remap_file_pages */
.long SYMBOL_NAME(sys_ni_syscall)       /* sys_set_tid_address */
```

Up to now, I have only the defined the entry point for my future kernel calls but actually no function has been written yet. For this point, writing a system call has not big differences with respect to normal library functions: prototypes will be declared in a dedicated `.h` file and the source in a `.c` file. Anyway, the execution of a system call is radically different.

The paths in which these files will be saved should not be constrained; anyway, following the nominal convention, the header has been saved in `include/linux` (since is hardware independent), while the source simply in `kernel/`.

In the following pictures, both header and source will be present.

```
#ifndef __LINUX_MYSERVICES_H
#define __LINUX_MYSERVICES_H

#include <linux/linkage.h>
#include <linux/unistd.h>

_syscall0(void, kernelprint);

_syscall2(void, rot13, char*, str, char*, result);

#endif
"include/linux/myservices.h" 11L, 203C
```

Figure 1: `myservices.h` content

```
#include <linux/myservices.h>
#include <linux/kernel.h>
#include <linux/malloc.h>

asmlinkage void sys_kernelprint () {
  printk("Printed by the kernel! Yuppi\n");
}

asmlinkage void sys_rot13 (char* str, char* result) {
  int i = 0;
  // loop until str itself is not NULL and str[i] is not zero
  for(i=0; str[i]; i++)
  {
    char a = ~str[i];
    result[i] = ~a-1/(~(a|32)/13*2-11)*13;
  }
}
"kernel/myservices.c" 17L, 398C
```

Figure 2: `myservices.c` content

`kernelprint()` is the same function provided in the document and simply print (using `printk`) a message on the terminal. It has been defined as a system call that takes no parameters and returns to void.

On the other hand, `rot13(...)` is more complex. To avoid problems related to memory allocation and data copy between kernel space and user space (which eventually may lead to a Segmentation Fault), the function has been defined with two parameters (the pointer to the original string and the pointer to the modified one) and returns to void. For the function, the return string has supposed to be already allocated and pointing to a valid memory address.

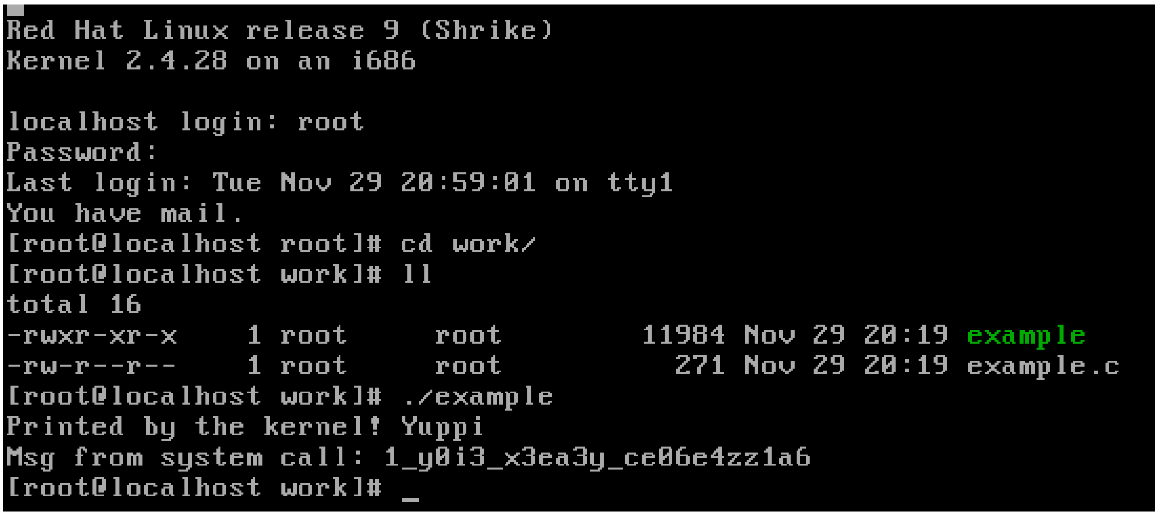
The functions are now defined and written; to make them available as system calls, the kernel should be recompiled (note that the Makefile should be updated with the new required object).

Let's look at the definition of a function; one small detail identify the function to behave as a system call: the flag `asmlinkage`. This access identifier tells to the GCC compiler to look on the CPU stack for the function parameters, instead of registers. As aforementioned, system calls are services that user space can call to request the kernel to perform something for them (and therefore execute in kernel space). These functions will not behave like normal functions, where parameters are typically passed by writing to the program stack, but instead they are written to registers. While still in userspace, calling a syscall requires writing certain values to certain registers is translated. The system call number will always be written in `eax`, while the the rest of the parameters will go into `ebx`, `ecx`, etc. The CPU is interrupt with a software `int` and the switches to kernel mode to then execute `system_call()`. So, since all the information about the parameters passed all the way from userland to this point is nicely stored in the stack, the compiler must be instructed about this.

After the recompiling operation has been completed, the system has been reboot on the new kernel and this is the result of the use of these two new system calls.

---

https://www.quora.com/Linux-Kernel-What-does-asmlinkage-mean-in-the-definition-of-system-calls



```
Red Hat Linux release 9 (Shrike)
Kernel 2.4.28 on an i686

localhost login: root
Password:
Last login: Tue Nov 29 20:59:01 on tty1
You have mail.
[root@localhost root]# cd work/
[root@localhost work]# ll
total 16
-rwxr-xr-x    1 root     root        11984 Nov 29 20:19 example
-rw-r--r--    1 root     root          271 Nov 29 20:19 example.c
[root@localhost work]# ./example
Printed by the kernel! Yuppi
Msg from system call: 1_y0i3_x3ea3y_ce06e4zz1a6
[root@localhost work]# _
```

Figure 3: Examples of system call use