Interaction Between the User and Kernel Space in Linux

Kai Lüke, *Technische Universität Berlin*

Abstract—System calls based on context switches from user to kernel space are the established concept for interaction in operating systems. On top of them the Linux kernel offers various paradigms for communication and management of resources and tasks. The principles and basic workings of system calls, interrupts, virtual system calls, special purpose virtual filesystems, process signals, shared memory, pipes, Unix or IP sockets and other IPC methods like the POSIX or System V message queue and Netlink are are explained and related to each other in their differences. Because Linux is not a puristic project but home for many different concepts, only a mere overview is presented here with focus on system calls.

1 Introduction

Kernels in the Unix family are normally not initiating any actions with outside effects, but rely on requests from user space to perform these actions. In a similar way, a user space program running without invoking kernel services has no visible effect out of its internal computations. Therefore, both need to interact to produce results, and a common program execution trace consists of interwoven kernel and user space code.

In a wide-spread end user operating system the management of resources needs to happen through a definded, stable interface to enhance portability of applications. An operating system also needs to provide a security model based on priviliges if it is to execute untrusted code or serves as a multi-user environment which should e.g. shield filesystem and network operations from each other.

The interfaces between user and kernel space in Linux will be introduced in the following sections with the intention to give insights in how they work and what they are used for. System calls as the primitives of interaction in Linux are mentioned first and all other concepts will involve system calls. It makes sense to look at interrupts in general and also the option of virtual system calls which avoid the context switch.

The filesystem tree exposes certain kernel interfaces in an accessible way at various points. While the idea *everything is a file* is not completely fulfilled in Linux there are still many possibilities to introspect or manage processes, resources and configurations. Some of these special purpose virtual filesystems are commonly present on all systems, others can be activated on demand.

Since a process is always a possible subject to signals they are a good entry to inter-process communication (IPC). Semaphores, shared memory and message queues are both supported in POSIX as well as System V style. A very common principle for IPC are sockets, and pipes can be seen as their most simple case. Besides the popular IP family with TCP/UDP, the local Unix domain sockets play a big role for many applications, while Netlink sockets are specific to the Linux kernel and are not often found in user space applications due to portability to other operating systems.

There have been attempts to bring the D-BUS IPC into the kernel with a Netlink implementation, kdbus and then Bus1, but this will not be covered here. Also comparative studies with other operating systems are out of scope.

2 Kernel and User Space

The processes have virtualized access to the memory as well as the processor and its registers in the sense that they do not have to care about other programs also making use of it because the kernel saves and restores the state [1]. In its x86_64 variant Linux is utilizing the memory management unit (MMU) to provide a flat memory layout of continuous logical address space by mapping it to the physical memory in units of pages [2].

The page table and its cache, the translation lookaside buffer (TLB), is always present and needs to be exchanged if a different process is scheduled. The kernel memory is mapped to the higher canonical address space of every process and code execution there makes also use of logical addresses. If a kernel thread is to be scheduled, the page table of the previous process can thus remain [2].

The user space program is not allowed to directly access any logical address because not all is mapped or mapped for a special purpose like the kernel memory. Entries in the page table are annotated with attributes like read, write and execute permissions, presence of the mapping and if it is meant for access from kernel or user space.

The program stack is used during the execution of user code and an additional kernel stack is maintained for execution in kernel mode. This transition needs lifted priviliges to jump to into the kernel memory and is usually done by registering interrupt handlers in the CPU.

Besides the hardware interrupts to handle device notifications there are software interrupts which are triggered through execution of instructions. Common exceptions are invalid memory access or invalid instruction usage or computation errors [3]. In the Intel 32-bit architecture Linux uses the software interrupt `int 0x80` to trigger a system call and in the 64-bit variant there are special system call instructions to enter and leave a system call but the result is the same. The registers are saved, the kernel stack of this process used and the requested system call function invoked and afterwards execution returns to user space code.

Code in the kernel section can perform almost all functionality known from user space. This ranges from a simple `printk()` which is always allowed and issues a message for the kernel log (accessible via the `SYSLOG(2)` syscall or `/proc/kmsg`) up to e.g. an implementation of an in-kernel TCP server since all system calls function are available. But because context switches are expensive, kernel code avoids touching the floating point and SIMD registers.

Memory Layout

Kernel `0xffffffffffffffff-ffff800000000000` (c.f. `x86/x86_64/mm.txt` in [6])

- vsyscalls
- kernel module mapping
- kernel code
- stacks
- kasan shadow memory
- virtual memory map
- vmalloc/ioremap space
- direct mapping of all phys. memory
- guard hole, reserved for hypervisor

[Cannonical address sign extension hole]

User `0x00007fffffffffff-0000000000000000` (c.f. `/proc/[PID]/maps`, randomized)

- virtual dynamic shared object (vDSO)
- stack
- dynamic linker
- mmaps
- other shared objects (libraries)
- heap
- binary executable

Fig. 1. Virtual Address Space

3 Linux System Calls

System calls are the main primitives for communication with the kernel. Together they define an abstraction interface for the management of files, devices, processes and communication with the advantage that e.g. writing to a file does not need knowledge about the filesystem and disk drivers performing the write.

Also restrictions are set in place by the user permissions the process is running under. In addition to these traditional security model Linux offers capabilities, which are single aspects of the hightest (root) privilige level and can be supplied to processes either during runtime or attached to the binary executable. The list in `CAPABILITIES(7)` covers network operations, chaning file ownership, killing processes, mounting, loading kernel modules and more [4]. Another circumstance which has an effect on system calls is the control group of the process, and various controllers can specify resource limits and access policies, cf. `CGROUPS(7)`.

With Linux `NAMESPACES(7)` the different contexts can also have an impact on available mount points, user and process IDs or visible network and IPC resources. Finally, SELinux policies or seccomp mode (allowing only reads and writes) are to be mentioned and it is the kernel's duty to check all restrictions if a system call is requested.

Now to the details of the call procedure starting from the C standard library used by an application. In a POSIX compatible system many functions are just wrappers for the system calls. But linking against the kernel C functions is not possible and an architecture specific system calling convention has to be followed.

A call to e.g. `fopen(3)` or `open(2)` for file access in the libc implementation uses a macro from `linux/x86_64/sysdep.h` to resolve the syscall name to the syscall number from `asm/unistd_64.h`. The calling convention demands the arguments to be loaded into registers which involves inline assembly and the numer of the system call becomes the first argument. Then the system call instruction is issued and execution continues in kernel space. The return value will afterwards be in the usual register as with normal calls, so there is no additional assembly involved.

If the system call should not be invoked by a wrapper function there is also the generic `syscall(2)` function which only needs the number of the system call. In its documentation one can find the calling convention by architecture, which makes it easy to write it in plain assembly and monitor the system calls of the process with the `strace` utility:

```
global _start
section .data
  fpath: db '/dev/null'
section .text
  _start:
  mov rax, 2     ;; open(
  mov rdi, fpath ;;      fpath,
  mov rsi, 0     ;;      O_RDONLY)
  syscall
  ;; returns file descriptor in rax
  mov rax, 60 ;; exit(
  mov rdi, 0  ;;      0)
  syscall
```

```
$ nasm -f elf64 -o open.o open.S
$ ld -o open open.o
$ strace ./open
execve("./open", ["./open"], [/* 62 vars */]) = 0
open("/dev/null", O_RDONLY)                  = 3
exit(0)                                      = ?
+++ exited with 0 +++
```

Fig. 2. Interfacing the Linux Kernel

A detail of process creation in Linux is that the current process needs to be duplicated by `fork` and can then be replaced with a new executable image by `execve` as the first action in this new process. Some other well-known syscalls are `open, read, write, close` for file descriptors, `clone` for threads or tracking file events with `inotify_init`. The full list of around 300 available system calls is documented in `SYSCALLS(2)`.

The system call interrupt lets the CPU fetch the position of the handler function from a model specific register as new value for the instruction pointer and the privilige level is set to kernel mode [3]. On the kernel side this handler activates the kernel stack and saves the registers to it. Just for a short period interrupts have been disabled when the handler was started, but are activated again in order to have preemptible system calls [2].

```
/*  linux/fs/open.c
 *  Copyright (C) 1991, 1992 Linus Torvalds
 */
[...]
/* sys_open(const char __user *filename,
 *          int flags, umode_t mode)
 */
SYSCALL_DEFINE3(open, const char __user *,
                filename, int, flags,
                umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;

        return do_sys_open(AT_FDCWD,
                           filename,
                           flags, mode);
}
[...]
```

Fig. 3. Implementation of `open(2)` which is `sys_open()` internally [3]

System calls are written as C functions using the `SYSCALL_DEFINE` macro which takes care of the metadata. This ensures that the `sys_call_table` contains the function pointers according to the syscall numbers, and defines `asmlinkage` as calling convention, i.e. they receive their arguments internally from the stack [2]. The system call handler can then issue a normal `call` instruction to the address of the related syscall entry in the table. Inside the system call it is important to distinguish between pointers into user space and in-kernel pointers but helper functions like `copy_from_user()` and `copy_to_user()` are provided.

When the function is finished the execution returns to the user process with restored state and privilige level, and the return value of the system call is available.

Normally system calls have a single purpose but `ioctl(2)` sends requests to special device files and is used for a variety of operations instead of new system calls since they need common sense and recompilation of the kernel. Nowadays it is preferred to expose attributes in the `sysfs`.

In contrast to calls to the kernel form user space Linux also supports shifting tasks to the user space with `call_usermodehelper()` which provides a wrapper function to spawn a user process and wait for the result.

4 VIRTUAL SYSTEM CALLS

Not every request really needs a context switch and for commonly used functions where shared data is accessed there are two machanisms. One is the legacy `vsyscall` memory mapping which contains simple pseudo-syscall functions like `gettimeofday()` and a shared memory region which holds the data to return.

Due to security reasons with the `vsyscall` statical mapping a new machanism of a *virtual ELF dynamic shared object* (vDSO) was developed because it supports address space layout randomization (ASLR). Where it resides is passed to the process as *auxiliary vector* variable, cf. `VDSO(7)`.

5 VIRTUAL FILESYSTEMS

Unix shells are well-suited for file processing and exposing the operating system through file objects is a powerful idea. The tree structure helps to find orientation and the set of actions on files is comprehendible.

```
/
|-- dev/
|   |-- audio
|   |-- null
|   |-- sda
|   |-- block/
|   |   `-- 8:0 -> ../sda
|   |-- bus/
|   |   `-- usb/
|   `-- char/
|       `-- 1:3 -> ../null
|-- proc/
|   |-- 12345/
|   |   |-- cgroup
|   |   |-- cmdline
|   |   |-- cwd
|   |   |-- environ
|   |   |-- fd/
|   |   |   `-- 0
|   |   |-- io
|   |   |-- mem
|   |   |-- mounts
|   |   |-- syscall
|   |   `-- tasks/
|   |-- cgroups
|   |-- partitions
|   `-- sys/
`-- sys/
    |-- block/
    |-- bus/
    |-- class/
    |-- devices/
    |-- fs/
    |   |-- cgroup/
    |-- kernel/
    |   |-- debug/
    |   |-- config/
    |   `-- cpuset/
    |-- module/
    `-- power/
```

Fig. 4. Parts of the virtual filesystem tree

The devtmpfs virtual filesystem is filled by the kernel with all device nodes requested by drivers. It is normally mounted in `/dev` and also managed by the udev service in addition. These device files represent block or character stream devices (non necessarily physical) which are handled in the kernel. They can also be created with `mknod(1)` and are determined by a major and a minor ID.

Examples for character devices are the random generator or the null device. The common storage volumes and their partitions are accessible as block devices and can be manipulated like files.

```
$ ls -l /dev/loop0 /dev/null
brw-rw---- 1 root disk 7, 0 /dev/loop0
crw-rw-rw- 1 root root 1, 3 /dev/null
$ stat -c "%F, Mm: %t,%T" /dev/{loop0,null}
block special file, Mm: 7,0
character special file, Mm: 1,3
```

Fig. 5. The type, major and minor number of a file

While various virtual filesystems can be mounted to interface with the kernel the `proc(5)` filesystem is almost always there. It allows configuration and introspection of processes or kernel subsystems.

Each process has a subfolder given by its process ID. It is populated with information on the environment variables, command line arguments, links to the files behind the acquired file descriptors, current syscall, process threads, cgroup information, the mount environment and I/O statistics.

The `sysctl(8)` utility essentially accesses the files in `/proc/sys/` to e.g. configure IP package forwarding or the memory swapping policy. The kernel maintains tables which specify the allowed content and a handler function for value change [5].

Read-only kernel debugging is possible in gdb through the `/proc/kcore` file and many other files like e.g. `/proc/partitions` exist. Internally there are two APIs, the original and the newer seq_file interface to overcome the limit of single page reads [5].

A more systematic approach, but similar to `/proc/sys/` is found in the newer `sysfs` which can expose kernel objects and their attributes more easily by making use of the internal hierarchy [2]. It is commonly mounted in `/sys` and uses directories to represent objects with their parent/child relation. In this way symbolic links are used to interconnect e.g. device classes with the devices of this kind.

The object attributes are contained as files in the directory and for read/write operations the defined functions in the kernel are invoked. This superseeded the practice of `ioctl` syscalls on special devices. And since `kobject_uevents()` can be used to emit an event for a kernel objects via Netlink messages to user space, the advantage of static exposure and dynamic messaging can be combined.

Other special purpose filesystems for interfacing kernel subsystems which may be found in `/sys/kernel/` are debugfs to modify values via the seq_file API, and configfs to create kernel objects from user space, cf. documentation in `filesystems/` of [6].

Control `cgroups(7)` were mentioned in the section on system calls and they can also be managed through the virtual filesystem. Modern Linux distributions mount the cgroups controllers which specified as mount options for the `cgroup` (v1) filesystem to subfolders in `/sys/fs/cgroup/`. The `cgroup2` filesystem abandons this flexibility for a unified mount point. New groups for each controller can be created through new directories and then processes be added by writing the PID to the `cgroup.procs` file. An example use case is to implement access restrictions in the device controller by writing e.g. `a *:* rwm` to `devices.deny` to deny reads, writes and mknods for all types and all major/minor device IDs. CPU sets allow to pin a process group to a certain CPUs. Memory, I/O or network restrictions are also possible.

All these user space requests need to go through the virtual filesystem tree into the specific filesystem which has a significant latency compared to direct system calls [7]. Sometimes it is possible to use either the `sysfs` entry or an (`ioctl`) system call depending on the needs.

6 Signals

Process signals as defined in POSIX are tied to actions (where also ignoring is an action). A process can register a handler function to replace the default action to e.g. handle a Ctrl-C `SIGABRT` on the terminal. The handler is asynchronously invoked independently from the normal programm execution [1]. A signal mark can block signals during handler execution. Magic SysRq keyboard commands can issue termination and kill signals to all processes directly from the kernel side [2].

Linux supports POSIX standard and realtime signals as listed in `SIGNAL(7). SIGTERM` asks the process to terminate while `SIGKILL` even cannot be handled and directly halts the process. They have the IDs 15 for `TERM` and 9 for `KILL`, where 15 is the default value for the `kill(1)` utility. `SIGSTOP` can also not be handled and prevents the process from being scheduled.

Processor exceptions which trigger a kernel interrupt will come as e.g. `SIGFPE` or `SIGSEGV` signals to the process for invalid floating point operations or memory access. Signals can carry a data word which might be useful for simple IPC with the non-specific `SIGUSR1`.

7 Shared Memory, Semaphores and Queues

Both the System V `SVIPC(7)` and POSIX API variants for shared memory, semaphores and message queues are offered. Shared memory regions can be created or the same file mapped to memory. This concept is essential if copying large amounts of data is to be avoided, but can also serve as communication method combined with mutal exclusion through blocking semaphores as in `SEM_OVERVIEW(7)`. A simple list of active resources is available through the `lsipc` or `ipcs` utility. They are persistent in the kernel if the processes do not remove them. Access is gained through unique identifiers and the creating process may set restrictions.

The POSIX shared memory as described in `SHM_OVERVIEW(7)` can be named and referenced in `/dev/shm` or anonymous. Message queues as in POSIX `MQ_OVERVIEW(7)` allow multiple reading and writing processes.

8 Inter-Process Communication Sockets

Messages through sockets offer flexibility and portability compared to direct syscalls or shared memory. Pipes, IP and Unix domain sockets are common in user space IPC and can also be used in kernel space but Netlink sockets are unique to the Linux kernel, and therefore not often used seen in user space.

The most simple case is piping with anonymous pipes, a form of FIFO buffers. A single pipe implies unidirectional communication. They connect two processes through linked file descriptors, e.g. one for the standard input and the other for the output stream. Then there are named pipes which are located in the filesystem and can be created with `mknod(1)` or `mkfifo(1)`.

Concerning sockets there are various families available and it is to be distinguished between datagram and stream mode which gives a guaranteed ordering without the notion of packages.

TCP/UDP could cover many use cases but also comes with additional overhead and complexity. If there is no need to route the packages through a network then the local Unix domain sockets tend to be used. They can be anonymous or named, which either means an abstract identifier or a special file in the filesystem. Through `socket(2)` they can be created with the `AF_UNIX` domain family and are configured with `setsockopt(2)`.

More similar to `AF_INET` IP sockets with UDP, but not intended for network usage, are Netlink sockets. Netlink provides a uni- and multicast message bus with general or special purpose protocols [8]. The `NETLINK_ROUTE` protocol is used in the IP network routing stack of the kernel, others are `NETLINK_FIREWALL` and `NETLINK_FILTER`.

In the `AF_NETLINK` domain the number of protocols is limited to 32 and thus the generic GeNetlink multiplexer has a special role [9]. Through it 65520 families with multicast groups are available to be used as special protocols for kernel and user space communication, yet all using one single bus. An implementation can define message attributes as well as commands which have a callback function [5]. Libraries like libnl for user space applications exist.

9 Conclusion

The different concepts for interaction between the Linux kernel and user space have been briefly explained. System calls are the core mechanism and all others involve system calls. Through the evolution of Unix operating systems and the pragmatic approach in the Linux project there are many overlaps between them and historic luggage.

For new implementations it is wanted that they are not based on legacy concepts, but the borders are not always clear and the decision on what to use depends heavily on the purpose.

Adding system calls does not need to many changes but has disadvantages in the compile workflow and resulting portability. In their basic principle system calls only assume the user space to initiate communication and are not very extensible.

All other approaches can mostly be implemented in additional kernel modules and may thus provide a quicker development workflow. The use of filesystems for information exposure is a proven common practice. Particulary during development there are many ways to ease debugging with special filesystems.

For dynamic exchange the Netlink message bus is to be recommended since consumers can attach to it and the kernel is able to start communication. It is extensible and also suited for transport of larger data amounts.

In comparison with e.g. microkernel operating systems that necessarily feature an appropriate IPC mechanism, Netlink does not fill this gap for Linux. With the raise of containers there might be more attempts to implement the functionality of D-BUS in the kernel space.

But it is unlikely that system calls based on context switches will be replaced soon by parallel execution of kernel and user space. The presence of multiple CPU cores has promoted the use of asynchronous calls within user space already.

References

[1] Robert Love, *Linux System Programming*, 1st ed. O'Reilly Media, Inc., 2007, ISBN 978-0-596-00958-8

[2] Robert Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010, ISBN 978-0-672-32946-3

[3] Alexander Kuleshov, *Linux Insides*, 2017, Commit 3410012, https://0xax.gitbooks.io/linux-insides/content/index.html

[4] Various, *The Linux man-pages project*, 1994-2017, https://www.kernel.org/doc/man-pages/

[5] Ariane Keller, *Kernel Space, User Space Interfaces*, 2008, Rev. #11, http://wiki.tldp.org/Kernel_userspace_howto

[6] Various, *Linux Kernel 4.9 Documentation Files*, 2017, https://www.kernel.org/doc/Documentation/

[7] S. Maliye, S. Krishnaswamy and H. Gajula, *Quick access of sysfs entries through custom system call*, MicroCom, 2016, DOI: 10.1109/MicroCom.2016.7522511

[8] Neil Horman, *Understanding and Programming with Netlink Sockets*, 2004, http://people.redhat.com/nhorman/papers/netlink.pdf

[9] Pablo Neira-Ayuso, Rafael M. Gasca and Laurent Lefevre, *Communicating between the kernel and user-space in Linux using Netlink sockets*, Softw. Pract. Exper., 2010, DOI: 10.1002/spe.981