# Communications between kernel space and user space

## Taxonomy on Linux x86 architecture

Pipereau Yohan

October 29, 2018

## Contents

# 1 Introduction

The structure of the Linux operating system on x86 architecture, is often described with a hierarchical ring model. The separation between core functionalities (kernel land) and the user environment (user land) aims at enforcing the security of the system. Though, as communications between user land and kernel land are necessary for many operations, there is a need to make them as efficient as possible.

Kernel space refers to set of mechanisms of an operating systems, while user space refers to operating systems policies. Communicating between user land and kernel land is necessary to access kernel mechanisms like file-system operations, block devices operations, network operations, to perform memory allocation, operate on processes, etc. Every program in user land rely on kernel mechanisms and need an efficient and generic way to access them.

Improving latency or throughput for user-kernel communications is critical for system performance. However, there are other important aspect of this communication like reliability, standardization of the mechanism, whether it is synchronous or asynchronous, bidirectional or unidirectional, user-space initiated or kernel-space initiated. This is why different mechanisms exists to realize this communication.

There are several parameters to take into account regarding kernel-space and user-space communications. [1]

- **"event-based signaling mechanism"** : avoid user-space loop to poll information from kernel-space;
- **"large data transfers"** or **"throughput"** : amount of data transferred per unit of time;
- **"latency"**;
- **"extensibility"** : ability to add new protocols for communication;
- **"architectural portability"** : communication require a common agreement regarding types size.

This survey focus on x86 processors, thus this introduction will give more details about architecture specific security mode. In its design, x86 processors provide different modes to execute or access hardware at various level of privileges. There is a hardware mode named **Protected Mode** or **Supervisor Mode** [1] on Intel x86 processors which provides itself four levels or rings numbered from zero to three, zero being the most privileged level. On Linux and other monolithic kernels, only two modes are used : user mode (ring 3) and privileged mode (ring 0). The latest allows any processor instruction to be run (ex : *LGDT, HLT, WRMSR* , ...). The switching between these different levels relies on three register fields:

- **CPL, Current Privilege Level :** the privilege level of current process;
- **DPL, Descriptor Privilege Level :** the privilege level of a segment;
- **RPL, Requested Privilege Level :** the requested privilege level;

Monolithic Kernel are built to make the kernel run in supervisor mode and user applications in unprivileged mode. Memory for example relies on this ring model to determine access to pages, "Memory protection is supported by a bit in each page table entry which indicates whether the page can be accessed by level $0-2$ or by level 3. Instructions executed while in supervisor mode have access to all system memory. Instructions executed in user mode only have access to a private address space. " [2]

When the processor switches between two modes, there is a **priviledge context switch**[2]. They are required in various situations like interrupts, exceptions, system call, etc. However, privilege context switching has a cost. "The processor registers need to be saved and restored, the OS kernel code (scheduler) must execute, the TLB entries need to be reloaded, and processor pipeline must be flushed." [3] Moreover, recurrent switching is likely to reduce the effectiveness of caches.

# 2 System calls

In this section, we will dwell on the system call mechanism, then on vsyscall and vDSO mechanisms.

[1] Other hardware modes like Real Mode, Hypervisor Mode or System Management Mode (SMM) exist but are not detailed here.

[2] Do not mistake privilege context switching for process context switching. They work differently.

## 2.1 System calls

System call is probably the first idea one might have about kernel land and user land communications. They are essential for every other communication mechanisms, so they are not to be compared with the other mechanisms.

"In an operating systems, all the privileged operations are carried out in the privileged rings, such as modifing kernel data structure, interacting with devices, etc. OS does not allow user programs to invoke them in an arbitrary way, such as jumping to the middle of a privileged operations. Instead, OS provides a number of interfaces to users programs, which can only invoke those provileged operations via the interfaces. These interfaces are often called system calls." [4]

System calls are functions which are architecture dependent. These functions work like interrupts, hence in x86_64 on Linux, one can use *int 0x80* or *syscall* assembly instructions.

On Linux, all system calls are defined in *'/usr/include/sys/syscall.h'* which include *'/usr/include/bits/syscall.h'*. Most syscalls are wrapped by some code by the libc before sending the effective syscall to the system call handler in kernel space. The system call handler is an array of function pointers indexed by system call numbers.

In order to pass arguments of system calls, Linux follows the System V AMD64 ABI and pass arguments to the following registers: *RDI, RSI, RDX, RCX, R8, R9.* Stack is used for more arguments. Note that using processor register is preferred over passing from user process stack to kernel stack.

At startup time, the kernel writes the address of the **system call handler** into the *LSTARMSR* register. Every time a system call is made, syscall jumps to this address. For the Linux Kernel on x86_64, the syscall handler is *entry_SYSCALL_64*. The syscall function is triggered based on the value passed in the RAX value. Then, during execution time of a program, "SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the *IA32_LSTARMSR* (after saving the address of the instruction following *SYSCALL* into *RCX*. (The *WRMSR* instruction ensures that the *IA32_LSTARMSR* always contain a canonical address.)" [5]

Using system call is considered expensive as the processor needs to do the following steps:

1. interrupt the current task
2. switch context to kernel mode
3. execute the system call handler
4. jump into userspace

## 2.2 vsyscall

System calls are expensive for communication between user land and kernel land. Thus, the linux kernel has introduced vsyscall to accelerate system calls.

The kernel maps into user space a page containing some variables and syscalls. Then system calls are executed in user space without context switching. Note that usage of vsyscall requires the *CONFIG_X86_VSYSCALL_EMULATION* option. However, vsyscalls are causing some security concerns as it maps the page at a fixed address every time.

The output of *'cat /proc/self/maps'* shows virtual memory mapping for the current process:

```
7fffd5bd4000-7fffd5bd6000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Here this output illustrates the fact that kernel memory is mapped into every process to avoid context switching as much as possible. Virtual memory saves a portion for the kernel memor from `0xffff 8000 0000 0000` to `0xffff ffff ffff ffff` and the one for user space is `0x0000 0000 0000 0000` to `0x0000 7fff ffff ffff`.

### 2.3 vDSO

vDSO is similar to vsyscalls, but provides some improvements. The vDSO mechanism (Virtual Dynamically linked Shared Objects) is a memory area allocated in user space which exposes some kernel functionalities at user space in a safer manner than vsyscall. vDSO works similarly to the mapping of dynamic library in process memory. It is possible to create a new vDSO with a bit of hacking [6]. Though, this approach will not provide a generic communication interface.

"The main difference between the vsyscall and vDSO mechanisms is that vDSO maps memory pages into each process in a shared object form, but vsyscall is static in memory and has the same address every time." [7]

## 3 Netlink sockets

"Netlink is a socket family that supplies a messaging facility based on the BSD socket interface to send and retrieve kernel-space information from user-space." [1]

It is architecture portable, allow large data transfers, and provide event based signaling. [1] When used with attributes in a networking format, it is useful to developers willing to pull specific attributes and prevents attributes type misunderstanding between sender and receiver. It also supports unicast and multicast and it was designed to be used as a potential substitution mechanism to *ioctl*. However, netlink is datagram-oriented thus it is **not reliable** as provided.

Netlink requires two implementations: a user space implementation based on BSD socket system calls and a kernel space implementation. It also requires *CAP_NET_ADMIN* capability to communicate.

### 3.1 Netlink architecture

Messages going from user to kernel space are not queued thus transmitted directly by the kernel subsystem. Yet, messages going from kernel to user space are queued.

### 3.2 Description of Netlink API

Netlink RFC (3549) provides most informations on the format of netlink messages. There is also an API named libnl [3] which provided many primitives.

#### 3.2.1 Socket creation

Netlink relies on datagram communication thus both *SOCK_RAW* and *SOCK_DGRAM* are valid values for socket_type. Linux Kernel already offers a few implementation of netlink families *NETLINK_ROUTE*, *NETLINK_USERSOCK*, *NETLINK_FIREWALL*, ...

Creation of a netlink socket in userland relies on *socket* system call.

```
netlink_socket = socket(AF_NETLINK, socket_type, netlink_family);
```

#### 3.2.2 Binding & multicast

Binding information are registered in *struct sockaddr_nl*. Regarding multicast, each netlink family has a set of 32 multicast groups. It is recommended to use *setsockopt()* to set any option including subscribing to a multicast group. It is not necessary to subscribe to a group to send messages to that group.

[3] `https://www.infradead.org/~tgr/libnl/`

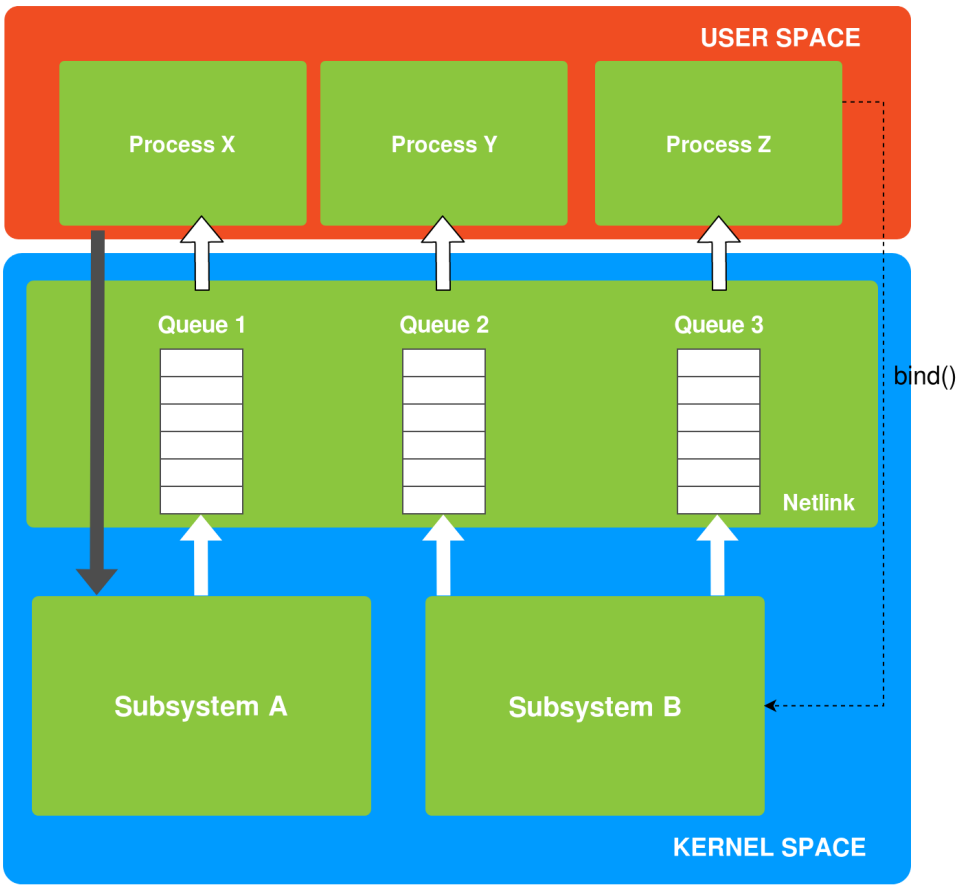


Figure 1: Netlink transmission between kernel land and user land

### 3.2.3 Accessing data

*NLMSG** macros must be used in user space to access netlink message parts. Netlink uses the host endianness instead of using the Big Endian network byte order commonly used in BSD socket programming in C.

The socket is represented with *struct sock*. The message header is represented with *struct nlmsghdr*. The message is represented with *struct sk_buff*.

### 3.2.4 Scattering & Gathering based sending and receiving

Sockets have two queues : one for receiving and one one for sending. These queues are sk_buff double linked lists (*struct sk_buff_head*) sk_buff contains fields for data and length. In netlink the data field contains netlink header *struct nlmsghdr* and payload.

The structure represented here, correspond to vectorized I/O using *struct iovec*. Vectorized sending will benefit the application in terms of global throughput. The message is carried by *struct msghdr*, the msg_iov field points to the iovec structure of the message. The iov_base field of this structure finally contains the *struct nlmsghdr* corresponding to the header of the netlink message.

At reception, you can either allocate a fixed buffer and put message content in it to optimize latency or you can read the header, allocate and copy the payload. The first method is to be preferred if message size is known beforehand, the second method is expensive in latency but optimizes memory consumption.

### 3.3 Netlink reliability

"Netlink is not a reliable protocol because it relies on datagrams. Message lost is always caused by congestion error occuring when the socket is not able to store a new message in the socket buffer. This can be caused by:

- a message bigger than the authorized socket buffer capacity;
- a buffer full of multiple messages because the kernel writes quicker than the user process is able to read.

However, there are different solutions to ensure message delivery or reduce the occurence of this problem:

- acknowledgement messages
- *nlmsg_unicast* return code
- increase socket buffer size

For reliable transfer the sender can request an acknowledgment from the receiver by setting the *NLM_F_ACK* flag. An acknowledgment is an *NLMSG_ERROR* packet with the error field set to 0. The kernel tries to send an *NLMSG_ERROR* message for every failed packet. A user process should follow this convention too.

Kernel *nlmsg_unicast* returns an error in case of failure to copy the message into the buffer. It raises a *ENOBUFS* which you can handle to divide the message in smaller chunks or by queuing it and reemitting it later.

You can grow the socket buffer size via procfs (*/proc/sys/net/core/rmem_max*), or use *fcntl* with *SO_RCVBUF* option.

What netlink lacks is a control flow algorithm to make sure that we do not send data when the link is full. Let us have a quick overview of TCP flow control mechanism. TCP uses a sliding window [4] to solve the problem of congestion. Compared to TCP, netlink benefits of a reliable communication link. Indeed, no data is supposed to be lost between user land and kernel land except in case of crash. However, TCP benefits of a header which includes a field for the window, this field does not exist on netlink and it will limit the algorithm we can implement with netlink.

This implementation is based on a timer used to let the user enough time to deliver messages.

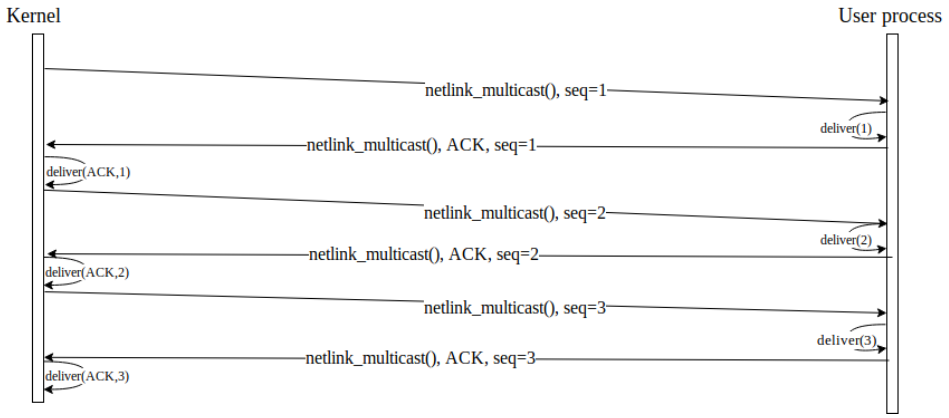


Figure 2: netlink with 1 message-1 ACK

This implementation makes sure every single message has been received before sending the next one.

In the current implementation of netlink, we can not have a dynamic sliding window. Indeed the size of the window can not change without patching the netlink implementation in the kernel. And because ioctl SIOCINQ is not supported for netlink socket, we can not get the available size of the socket buffer. Moreover there is no field provided for window in the netlink message header structure. Thus, the best we can do is to have a fixed size of sliding window based on the socket buffer size divided by the fixed size of each message.

[4] A window is the maximum number of data that can be sent before an ACK is received.

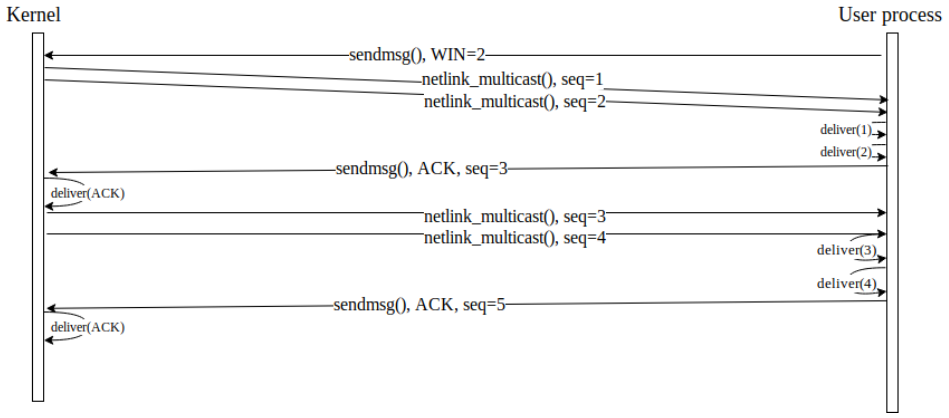


Figure 3: netlink with fixed window size

### 3.4 Netlink dump

"If Netlink fails to deliver a message that goes from kernel to user-space, the recvmsg() function returns the No buffer space available (ENOBUFS) error. Thus, the user-space process knows that it is losing messages so, if the kernel subsystem supports dump operations (that are requested by means of a Netlink message with the NLM_F_DUMP flag set), it can resynchronize itself to obtain up-to-date information." [1]

It is the developer responsability to implement message splitting and flow control in dump operations.

### 3.5 Generic Netlink

Generic netlink is a netlink family (NETLINK_GENERIC, family number 16) which offers another API for kernelspace-userspace communications. The API was designed to be very similar to netlink API.

Generic netlink main advantage compared to netlink is that developer do not need to register a family identifier in the kernel to make sure no one else uses the same communication channel. Moreover, generic netlink offers a wider range of port than netlink (32 ports).

Generic netlink uses netlink header and adds its own header (*struct genlmsghdr*) and let some place to define a user specific header as well as a message payload which generally contain attributes.

## 4 file-systems

### 4.1 Procfs

Procfs is a pseudo file-system created to get information and set parameters for processes. Procfs contains an entry for every processes. It was implemented for process information sharing with the user space thus we will not dig into the detail of this interface.

### 4.2 sysfs

Sysfs is a virtual filesystem introduced to avoid the bad habit of using procfs for information sharing between kernel space and userspace. Though, sysfs is only an interface for information sharing of kernel subsystems like devices.

### 4.3 debugfs

Debugfs is “a virtual filesystem devoted to debugging information.” [8]. It is useful to make debugging information available in user land, yet it relies on a unidirectional communication initiated in kernel space for user space.

### 4.4 Creating a new virtual filesystem with libfs

Libfs is a kernel library which can be used to create small filesystem. Libfs is mostly used for in-memory filesystem without a backing store. Though, this solution is expensive in terms of development because it requires implementing a new filesystem just to provide an interface between kernel space and user space. Moreover, most of the primitives used to interact from user space with the filesystem are system calls, thus you won't be a lot more efficient in reducing context switching.

## 5 Drivers & devices

### 5.1 character devices

Character devices are often implemented as a kernel module. They provide callbacks mechanisms for system calls which allow treatment of information upon system call. Though, they are not a mean of communication between user land and kernel land, the programmer can customize the character device to its need.

### 5.2 ioctl system call

Ioctl is a system call often used with devices. It is used to control device parameters. To give more details on it, "It is an unregulated means by which new system calls can be added to the kernel - it is easy to add large numbers of them, and some developers do." [9]

This system call works by using a sequence number to distinguish different ioctls from each other. In Linux, this sequence number is created using _IO, _IOW, _IOR or _IOWR maccros.

Regarding ioctl problems, ioctl is often designated as a 'mysterious system calls' as it allows to control devices in a way that could be dangerous like extending system calls with new operations that could not take into account every requirements to perform the operations.
"A clumsy developer could define a read or write operation based on *ioctl()* without checking buffer overflow as we did in the previous write()." [10]
Moreover, some operations can be sent to the wrong device and the identifier of the operation could be harmful for the device like destroying the hardware instead of resetting it.

It often appears in critics of UNIX design: "Although Unix derivatives try to make many things appear like a filesystem, there are still many issues that break this model within the Unix approach (e.g., ioctl)." [11]
Certain people would thus prefer the use of sysfs over *ioctl()* system call.

## 6 Relay Interface (formerly Relayfs)

"The relay interface provides a means for kernel applications to efficiently log and transfer large quantities of data from the kernel to userspace via user-defined 'relay channels'." [12]

"A 'relay channel' is a **kernel** $\rightarrow$ **user** data relay mechanism implemented as a set of per-cpu kernel buffers ('channel buffers'), each represented as a regular file ('relay file') in user space." [12] One of the major goals of the relay interface is to provide a low overhead mechanism for conveying kernel data to userspace. In userspace, you use 'read', 'mmap' system calls which is very convenient.

The inconvenient of the relay interface is that it serves for unidirectional communication only. This means, you can not use it to convey information from user land to kernel land.

The relay interface is commonly used with debugfs. Thus, relayfs files have been replaced by debugfs files[5] stored in */sys/kernel/debug*. After *create_buf_file()* relay callback is executed with *debugfs_create_file()*, four new files named upon filename argument appear in */sys/kernel/debug* debugfs mount point.

[5] commit b86ff981a8252d83d6a7719ae09f3a05307e3592

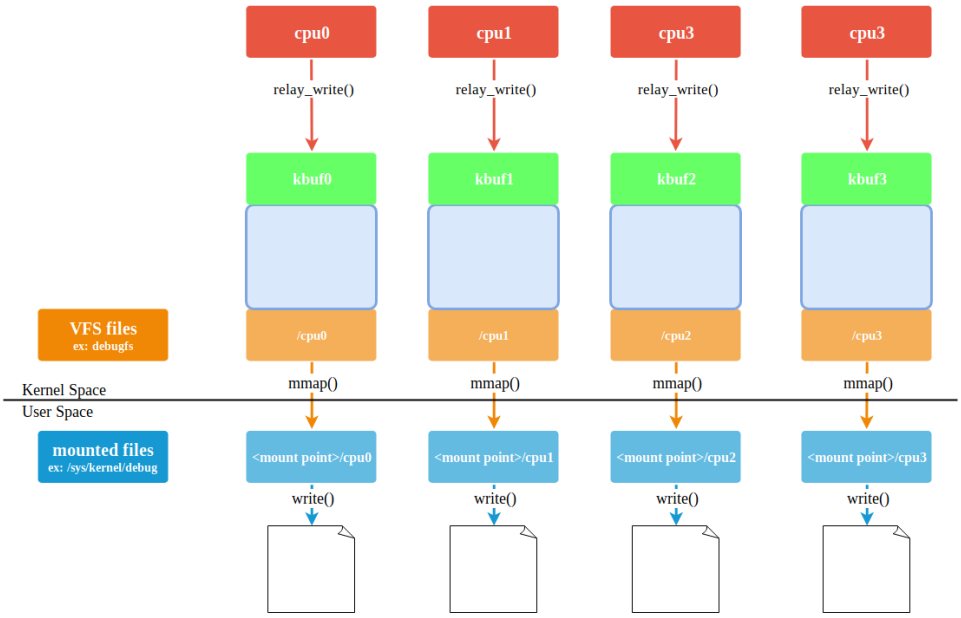


Figure 4: Relay API transmission mechanism

Then, data can be consumed in user land either by using the *read()* system call. You can also use *mmap()* which offers better performance as it requires less context switching yet debugfs files can not be mmaped. You will need to use another file system with *mmap()* support.

# 7 Inter-Process Communication primitives

This section presents IPC mechanisms because they rely on user kernel-space communications to benefit from a specific mechanism implemented in kernel-space in order to exchange informations between two processes. Another reason why IPC appears in this document is because, it is possible to implement a character device which imitate an existing IPC mechanism or even directly hack IPC mechanism to communicate as described with *named pipes*.

"Local inter-process-communication (IPC) mechanisms are traditionally used to transfer data between user space processes, but these mechanisms can be modified to accommodate user/kernel data transfer." [2]

All the following IPC mechanism presented after are standardized in two main APIs :

- System V API
- POSIX API

These two APIs provide a standardized API to use shared memory, messages queues, semaphores IPC mechanisms. We will focus on the POSIX API considered as more efficient. IPC mechanisms all rely on the same generic functioning: a user process shares some information which is handled by a kernel mechanism, then another process get the information by communicating with the kernel.

The following table gives some figures about IPC mechanism latency. "The principle of these experiences is the following : a distributing thread sends 1,000,000 messages through a communication medium to handling threads. Performances are measured with *clock_gettime()* on a quadri-core processor Intel Core i7-4600U of 2.1 Ghz frequency, 8Go memory and Linux kernel 4.4.0.".[13]

## 7.1 POSIX - Message queues

"Message queues incur the cost of a system call for each send and receive issued but provide a protocol for synchronizing messages." [2]

| IPC mechanism | Latency ($\mu$s) |
|---|---|
| Pipe for 4096 Bytes messages | 2.37 |
| Pipe for 416 Bytes messages transferred in two times | 1.27 |
| POSIX - Shared memory | 0.91 |
| POSIX - Message Queues | 0.99 |

Table 1: Comparison of IPC mechanism latency

POSIX messages queues relies on *mq_ open()* which open a message queue identified by a name. These objects are listed in /dev/mqueue. Then, *mq_ send()* and mq_receive() allows you to send and receive messages in the wait queue. Finally *mq_ close()* closes the wait queue while *mq_ unlink()* deletes the device registered in */dev/mqueue*.

### 7.2 POSIX - Shared memory

"Shared memory has the advantage of direct memory access but has no synchronization protocol of its own. Typically, semaphores or signals are used to synchronize access to shared memory regions." [2]

Shared memory relies on the primitive *shm_ open()* which create/open POSIX shared memory objects. These objects are listed in ***/dev/shm***. Then, we configure the size of the memory segment with *ftruncate()*. With *mmap()* system call, we create a new mapping in the virtual address space of the calling process. Finally, *mremap()* and *munmap()* allows respectively resizing of the paging and deletion of the mapping.

Based on the comparison made in [8], **it appears, shared memory is the most efficient IPC mechanism in terms of processor cycles for user-kernel communications.** The start up delay is high because of page mapping initiated during first sending.

### 7.3 Named Pipes - FIFO

POSIX messages queues relies on *mq_ open()* which open a message queue identified by a name. These objects are listed in ***/dev/mqueue***. Then, *mq_ send()* and *mq_ receive()* allows you to send and receive messages in the wait queue. Finally *mq_ close()* closes the wait queue while *mq_ unlink()* deletes the device registered in */dev/mqueue*.

Pipes use their own filesystem named pipefs, and it is possible to write from kernel land directly to the pipefs file thanks to VFS write functions. However, it is not recommended to write to a file from kernel even if you use the VFS API:

> "In conclusion, reading and writing a file from within the kernel is a bad, bad thing to do. Never do it. Ever." [14]

There are various arguments to Greg Kroah-Hartman recommendation:

First, there is the argument for *Separation of mechanism and policy* which dates back to early operating systems. It is relevant whereas "Selecting where and in what format to read/write data is a policy and policy does not belong to kernel.". [15]

Then, if you are in atomic context, filesystem operations are not allowed. "You can't be sure you're in user context so you can't write something from (for example) an interrupt handler." [15]

## 8 Comparaison

### 8.1 Synchronous an asynchronous privmitives

Synchronous primitives are reliable but expensive for performance, however asynchronous primitives allows to perform other operations while data is being sent, yet you have to use acknowledgment or flow control algorithms to make sure data is delivered.

| Kernel to User communication | |
|---|---|
| **Synchronous** | **Asynchronous** |
| vfs_write() | nlmsg_unicast() |

Table 2: Comparaison of synchronous and asynchronous primitives from kernel space

| User to Kernel communication | |
|---|---|
| **Synchronous** | **Asynchronous** |
| ioctl, write syscall | write/send syscall with O_NONBLOCK |

Table 3: Comparaison of synchronous and asynchronous primitives from user space

### 8.2 Latency comparison

| | User sending | Kernel receiving |
|---|---|---|
| netlink API with dedicated port | 15 000 | NO DATA |
| character device with read/write ops | 52 405 | 182 |
| relay API | NOT POSSIBLE | NOT POSSIBLE |
| vfs read/write to pipefs | NOT IMPLEMENTED | NOT IMPLEMENTED |

Table 4: Comparison of mechanisms latencies in clock cycles for 1024Bytes messages exchanged from User to Kernel space

| | Kernel sending | User receiving |
|---|---|---|
| netlink API with dedicated port | 3 498 | 3 724 |
| character device with read/write ops | NOT IMPLEMENTED | NOT IMPLEMENTED |
| relay API with writing to debugfs | 158 | 6078 |
| vfs write/read to pipefs | 4 620 | 4 622 |

Table 5: Comparison of mechanisms latencies in clock cycles for 1024Bytes messages exchanged from Kernel to User space

Results of table 4 and 5 where obtained on Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz with 2 core and 2 threads per core with 1000 to 1024 Bytes. [6]

[6] You can find source code at `https://github.com/YohanPipereau/UserKernelCommunication`

## References


[1] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in linux using netlink sockets," *Software: Practice and Experience*, 2010.

[2] R. Slominski, "Fast user/kernel data transfer," Master's thesis, The College of William & Mary, 2007.

[3] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," *Usenix*, 2007.

[4] K. Du, "80386 protection mode." `http://www.cis.syr.edu/~wedu/Teaching/CompSec/Lectureurl_New/Protection_80386.pdf`.

[5] Intel, *Intel® 64 and IA-32 architectures software developer's manual combined volumes 2A, 2B, 2C, and 2D: Instruction set reference, A-Z*, May 2018.

[6] M. Davis, "Creating a vdso : the colonel's other chicken," *Linux Journal*, February 2012.

[7] "System calls in the linux kernel. part 3.." `https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-3.html`.

[8] J. Corbet, "Debugfs," *LWN.net*, 2004.

[9] J. Corbet, "Kernel summit 2006: The ioctl() interface," *LWN.net*, 2006.

[10] P. Ficheux, "Les mystères de l'ioctl," *Linux Magazine*, June 2018.

[11] J. Choate, "A gentle introduction to plan 9." `www.drdobbs.com/a-gentle-introduction-to-plan-9/199101771`, 2004.

[12] Linux Kernel Documentation, *relay interface (formerly relayfs)*, 2018.

[13] D. Conan, M. Simatic, and F. Trahay, *Concepts des systèmes d'exploitation et mise en oeuvre sous Unix - Communications inter-processus*. Télécom SudParis, 2018.

[14] G. Kroah-Hartman, "Driving me nuts - things you never should do in the kernel." `https://www.linuxjournal.com/article/8110`, April 2005.

[15] "Why writing files from the kernel is bad ?." `https://kernelnewbies.org/FAQ/WhyWritingFilesFromKernelIsBad`.