FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A micro-kernel API for Linux

Joaquim Monteiro

DISSERTATION

Mestrado em Engenharia Informática e Computação

Supervisor: Pedro Souto

July 25, 2024

A micro-kernel API for Linux

Joaquim Monteiro

Mestrado em Engenharia Informática e Computação

July 25, 2024

Abstract

Writing device drivers, and other programs that interact with hardware, is, typically, hard and error-prone. Due to the architecture of many operating systems in use today, such as Linux, it usually requires writing kernel level code, which is of higher difficulty than regular user space code. Additionally, kernel level code runs in kernel space, meaning that a problem in one kernel component can affect the entire system, making it crash, behave incorrectly, or open up security vulnerabilities.

This dissertation proposes an interface for writing these kinds of software in user space on Linux, similar to the interfaces provided by micro-kernel operating systems. This is implemented with a kernel module that communicates with user processes through Netlink sockets (and an accompanying user space library) and executes privileged commands on their behalf. With it, user processes can perform operations such as reading and writing to memory-mapped devices, and receiving interrupt notifications. It also provides fine-grained control over the resources a process can access to the system administrator. As such, it allows writing programs that can interface with hardware and that, at the same time, run in an isolated user space process with the least amount of privilege required to perform their tasks.

A simple PS/2 keyboard driver and a framebuffer program were written to test and demonstrate the solution's capabilities.

Contents

Abbreviations

API	Application Programming Interface
IPC	Inter-process communication
IRQ	Interrupt request
OS	Operating System

Chapter 1

Introduction

Writing low-level code that interacts with hardware, such as device drivers, is not easy. On typical operating systems, these kinds of programs run at the kernel level, and faults in this type of code can affect the rest of the system, and cause it to malfunction or crash. Historically, we've observed that driver code is responsible for a large part of operating system errors [2, 3]. Part of the difficulty of writing correct driver code could be attributed to the difference in structure and requirements compared to regular user space code, which require some kernel expertise to get right.

Micro-kernel operating systems let us write these programs in user level code, facilitating development and making systems more reliable and secure [1]. However, using them implies moving away from the commonly used OSes, which are well known by most programmers and have rich software support.

This work proposes a new micro-kernel API for Linux to allow writing programs that interact with hardware in user space to avoid these issues, while still using a traditional Linux environment.

1.1 Document Structure

Beyond this introduction, this dissertation contains five more chapters. Chapter 2 provides the background and presents the state of the art. Chapter 3 analyzes the MINIX 3 kernel API to determine the features required for writing user space drivers. Chapter 4 describes the design and implementation of our solution. Chapter 5 evaluates our solution. Chapter 6 presents the conclusion of this work.

Chapter 2

State of the art

2.1 Background

Computer operating systems provide a common programming interface and environment for writing applications, including features such as process scheduling, virtual memory, disk I/O and networking. They also manage and provide abstractions over computer hardware, through pieces of software called device drivers.

One possible way to build an operating system is to implement all these components in kernel code (or in modules that can be loaded into the kernel as needed). This is called a monolithic operating system. Most common OSes, such as Windows or Linux, are monolithic.

Alternatively, a micro-kernel operating system has a small kernel with minimal features, providing support for I/O operations, virtual memory, process scheduling and IPC, with the bulk of the functionality required to implement these services implemented in user level code. MINIX 3 [1] is an example of such an operating system. In it, features like file access, networking and process management are implemented in user level servers, and each driver also runs as an individual user space process.

Monolithic architectures are conceptually simpler and can have better performance, as, inside the kernel, there is no need to perform context switches or use IPC mechanisms. Micro-kernels isolate each component of the OS, so a fault in one component can't affect the rest of the system, both in terms of stability and security. This allows MINIX to have a "self-repairing property": it monitors the state of all drivers and services, and performs an appropriate recovery procedure (such as restarting the affected server) in case of failure. Micro-kernels OSes also have much less kernel code, making them easier to verify and reason about.

2.1.1 Device drivers

Device drivers are pieces of software that implement support for a certain input/output device, so that it can be used by the operating system and by other programs without the need to deal with the specifics of that hardware.

These represent a large portion of operating system code, and are responsible for the majority of operating system errors [2, 3]. In monolithic kernels, as drivers reside in kernel space, a fault in the driver can cause issues with the whole system.

Thus, it has long been argued that micro-kernels are a superior design to monolithic kernels in terms of security and reliability. A study of the critical Linux CVEs [4] concluded that 96% of critical exploits wouldn't reach critical severity in a formally verified microkernel (such as seL4 [5]), and 57% would be reduced to low severity. It also concluded that even without formal verification, a micro-kernel design would prevent 29% of exploits.

2.1.2 Inter-process communication

Micro-kernels use IPC mechanisms, such as message passing, to communicate between the kernel and user space. MINIX 3 uses both synchronous message passing and notifications (for when blocking isn't an option, such as when dealing with interrupts).

However, some monolithic kernels have similar facilities. Linux has an extensible, socket based communication system named Netlink [6] [7], allowing communication between Linux kernel modules and user space processes. It supports both unicast and multicast messages. Netlink users can define protocol families with custom attributes and commands for use in their programs.

2.2 Related work

2.2.1 In-kernel isolation

There have been many attempts at isolating device drivers.

Nooks [8] is a new kernel subsystem that isolates drivers within lightweight protection domains inside the kernel address space, through the use of software and hardware checks. It works with conventional hardware and operating systems, and works with existing drivers. However, it is limited to uniprocessor systems.

XFI [9] provides software-based fault isolation through inline guards to perform runtime checks, which can be inserted by a binary rewriter, and a verifier that performs static analysis to check if a program contains the necessary guards.

Byte-Granularity Isolation (BGI) [10] is a fault isolation technique that can run drivers in separate protection domains in the same address space. It uses a library that intermediates communications between the driver and the kernel and a compiler plugin to generate instrumented code in order to apply this technique to existing, unmodified drivers.

LXFI [11] improves on XFI and BGI by allowing modules to partition their privileges into principals (allowing multiple instances of the same module to be isolated from each other) and by introducing annotations to enforce API integrity for complex kernel interfaces, implemented with a transpiler and a runtime component.

2.2.2 User space drivers

The Linux kernel has some mechanisms that can be used to write simple drivers in user space. Memory-mapped devices can be controlled by `mmap()`ing `/dev/mem`. On x86, the I/O ports can be accessed through the `ioperm` and `iopl` system calls and the various I/O port functions in glibc [12]. However, this approach is very limited both in the kind of devices it supports (for example, interrupt handling isn't possible) and in the isolation it provides, as it requires that the drivers run with an elevated permission level (either executed as the `root` user, or with the `CAP_SYS_RAWIO` capability).

To fill the gaps with these existing Linux mechanisms, [13] builds upon them by adding system calls for accessing PCI devices and setting up direct memory access (DMA), as well as implementing interrupt notifications by mapping IRQs to file descriptors.

The Userspace I/O system [14], which is part of the Linux kernel, allows writing user space drivers for devices that can be controlled completely by writing to mapped memory (or through mechanisms such as the x86 I/O ports). It still requires using a kernel module to register the device, and, optionally, to implement an interrupt handler. Compared to the other approaches, it uses a declarative way to describe device mappings instead of using system calls to set it up, and it supports devices that require actions to be taken after an interrupt is raised, as it allows writing custom interrupt handlers.

The Microdrivers architecture [15] splits drivers into a user space part and a kernel part. The user space part contains management code (such as initialization and configuration), while the kernel part contains performance-critical code (such as I/O or interrupt handling). This approach isolates the kernel from bugs in user space code, and it has performance comparable to regular kernel space drivers. The architecture was evaluated on the Linux kernel, and was implemented through a tool that splits existing drivers and generates marshalling code to perform communication between kernel space and user space. Additionally, it requires adding annotations to some parts of the driver code.

SUD [16] is a system that runs existing, unmodified Linux drivers as user space processes. It runs each driver in a separate process, with its own user ID, and it uses User-mode Linux [17] to emulate a Linux kernel environment in user space. To isolate access to the hardware, it uses the IOMMU and transaction filtering in PCI-E bridges (on x86, access to the I/O ports is controlled through the task state segment's I/O port bitmap).

There also exist some user space driver frameworks specific to certain domains. FUSE [18], a framework that is part of the Linux kernel, allows implementing file systems in user space. Importantly, it allows non-privileged mounts, where the file system process runs with the privileges of the user that mounted the file system. User space block devices [19] and network drivers [20] also exist, but these are mainly motivated by performance, not isolation.

Chapter 3

A look at the MINIX 3 API

We'll start by taking a look at the APIs provided by MINIX 3 and determine which ones are important for writing drivers. With that information, we'll obtain a target feature set that should give us parity to a real micro-kernel operating system.

Before that, a small note: most operating systems use the term system call (syscall) for calls to services provided by the kernel, such as ones defined by POSIX. In MINIX 3, user processes don't make direct requests to the kernel; POSIX syscalls made by user processes are translated into messages to the appropriate server process, and it's those servers that make requests to the kernel to implement the needed functionality. To distinguish the user syscalls from the requests made by server processes and device drivers, MINIX 3 calls the latter "kernel calls". It should also be noted that kernel calls cannot be used by regular user processes, only privileged processes (servers and device drivers) have access to them [22, p. 193].

Table 3.1 lists the process management kernel calls. Of these, `SYS_FORK`, `SYS_EXEC`, `SYS_CLEAR`, `SYS_EXIT`, `SYS_UPDATE`, `SYS_TRACE` and `SYS_RUNCTL` are used solely by the process manager [21] [22, pp. 194–197]. `SYS_SCHEDULE` and `SYS_SCHEDCTL` are related to scheduling, and are thus uninteresting. `SYS_PRIVCTL` is used by the reincarnation server (which is tasked with starting device drivers and other servers, and restarting them if they fail [22, pp. 114, 118]) to set the privileges of system processes [22, p. 195]; while we may want a mechanism to grant privileges to applications, we don't need to replicate this one exactly. `SYS_SETGRANT` is used to set up memory grants, which are an IPC mechanism for sharing large amounts of data between processes [23]; while it has some advantages over similar mechanisms present in Linux, such as POSIX shared memory [24], it is not very important for our purposes. Looking at the MINIX 3 source code, we find that `SYS_GETMCONTEXT` and `SYS_SETMCONTEXT` is used for getting and setting a process's machine context, which includes items such as CPU register values; we aren't interested in manipulating processes, so these kernel calls aren't relevant.

Table 3.2 lists the signal handling kernel calls. `SYS_KILL` has a direct Linux equivalent, the `kill` syscall. The other kernel calls are used by the process manager to implement signal handling [22, p. 196], while Linux handles this internally.

Table 3.3 lists the memory management kernel calls. `SYS_MEMSET` exists due to performance

Table 3.1: Process management kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_FORK	Fork a process; copy parent process
SYS_EXEC	Execute a process; initialize registers
SYS_CLEAR	Exit a user process; clear process slot
SYS_EXIT	Exit a system process
SYS_UPDATE	Update state of a system process
SYS_SCHEDULE	Scheduler
SYS_SCHEDCTL	Change scheduler control
SYS_PRIVCTL	Change system process privileges
SYS_TRACE	Trace or control process execution
SYS_SETGRANT	Tell kernel about grant table
SYS_RUNCTL	Set/clear stop flag of a process
SYS_GETMCONTEXT	Get context of a process
SYS_SETMCONTEXT	Get context of a process

Table 3.2: Signal handling kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_KILL	Send a signal to a process
SYS_GETKSIG	Check for pending kernel signals
SYS_ENDKSIG	Tell kernel signal has been processed
SYS_SIGSEND	Start POSIX-style signal handler
SYS_SIGRETURN	Return POSIX-style signal

Table 3.3: Memory management kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_NEWMAP	Install new or updated memory map
SYS_MEMSET	Fill a physical memory area with a constant pattern byte
SYS_VMCTL	(undocumented)
SYS_PADCONF	(undocumented)

Table 3.4: Memory copy kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_UMAP	Map virtual to physical address
SYS_UMAP_REMOTE	Map virtual to physical address
SYS_VUMAP	Batch map virtual to physical addresses
SYS_VIRCOPY	Copy data using virtual addressing
SYS_PHYSCOPY	Copy data using physical addressing
SYS_SAFECOPYFROM	Copy from a grant into own address space
SYS_SAFECOPYTO	Copy from own address space into a grant
SYS_VSAFECOPY	Handle vector with safe copy requests
SYS_SAFEMEMSET	Fill a grant with a constant pattern byte

Table 3.5: Device I/O kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_DEVIO	Read or write a single device register
SYS_SDEVIO	Input or output an entire data buffer
SYS_VDEVIO	Process vector with multiple requests
SYS_IRQCTL	Set or reset an interrupt policy
SYS_IOPENABLE	Give process I/O privilege
SYS_READBIOS	Copy from the BIOS area

Table 3.6: System control kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_ABORT	Abort MINIX: shutdown the system
SYS_GETINFO	Get a copy of system info or kernel data
SYS_DIAGCTL	(undocumented)

Table 3.7: Clock kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_SETALARM	Set or reset a synchronous alarm timer
SYS_TIMES	Get process times, boot time and uptime
SYS_STIME	Set boot time
SYS_SETTIME	Update time
SYS_VTIMER	Set or retrieve a process virtual timer

Table 3.8: Profiling kernel calls in MINIX 3 [21]

Kernel Call	Purpose
SYS_SPROF	(undocumented)
SYS_CPROF	(undocumented)
SYS_PROFBUF	(undocumented)

reasons [21] and can thus be ignored. SYS_NEWMAP is obsolete and no longer used [21]. Looking at the source code, SYS_VMCTL seems to be used for performing operations on virtual memory areas, which is also uninteresting; SYS_PADCONF is ARM-only, and is used to implement support for a couple specific devices. Due to this latter kernel call being very specific, we'll choose to ignore it.

Table 3.4 lists kernel calls for copying memory around, and is where we find the first interesting kernel calls. SYS_UMAP and SYS_UMAP_REMOTE are used to map physical addresses into a process's virtual memory. This is an important capability for device drivers, as many devices are controlled by reading and writing to physical memory that's been mapped to that device. As such, this is an interface we should provide as well. SYS_VUMAP does the same thing but in bulk, which could be helpful in some cases, but it's not vital. SYS_VIRCOPY and SYS_PHYSCOPY are used to copy memory, but they shouldn't be necessary on Linux. The SYS_SAFE* kernel calls relate to memory grants [23], so we have no use for them (see the reasoning for SYS_SETGRANT).

Table 3.5 lists kernel calls for handling I/O devices, and, as such, is also of interest. SYS_DEVIO is for reading and writing to I/O ports [22, p. 196], which are used by many (older) x86 hardware devices. Thus, it would be good to implement this as well. SYS_SDEVIO and SYS_VDEVIO do the same thing, but in bulk, with slightly different interfaces [22, p. 196]. These are helpful in some cases, but they aren't vital. SYS_IRQCTL allows installing an interrupt handler and enabling/disabling interrupts [22, p. 196], allowing the process to receive interrupt notifications, which is another essential operation for device drivers, so we need to provide it as well. SYS_IOPENABLE gives a process the privilege to perform I/O operations directly; this is probably unnecessary if we have a SYS_DEVIO equivalent. SYS_READBIOS is used to read data from the BIOS area. This is x86 only, not available in newer systems, and is of limited usefulness, so implementing this isn't a priority.

Table 3.6 lists system control kernel calls. SYS_ABORT shuts down the system, not very useful. SYS_GETINFO can be used to obtain various kinds of information about the system; as this information is MINIX specific, and Linux already provides various ways to obtain system information, providing an interface like this one doesn't make much sense. SYS_DIAGCTL, by looking at the source code, seems to be a debugging tool, for obtaining information such as stacktraces. As such, it's not relevant for our purposes.

Table 3.7 lists clock related kernel calls. As Linux already provides interfaces to control the system clock, and to use timers, none of these kernel calls need to be replicated.

Table 3.8 lists profiling kernel calls. As Linux already provides profiling tools, these kernel calls can be ignored.

3.1 Summary

In short, MINIX 3 provides the following functionality for driver authors:

3.1.1 Interrupt handling

One of the ways devices can communicate with the operating system is through interrupts. Typically, they're used to notify that an event happened. When triggered, the CPU suspends the task it was performing to execute the interrupt handler, a function registered for a specific IRQ line, that has the task of performing the appropriate action for the event that just occurred.

3.1.2 Memory mapped I/O

Many devices can be controlled by reading and writing data to sections of memory that are mapped to those devices.

3.1.3 Port I/O

In some architectures, such as x86, some devices are controlled by reading and writing to I/O ports assigned to them. This is done through special CPU instructions.

Chapter 4

Design and Implementation

4.1 Objectives

We want to design a system that meets the following requirements:

- It must provide the functionality selected in section 3.1.
- It must allow the user to use this functionality from a user space process.
- It should minimize the ability of a program to disrupt the stability and security of the operating system and of other programs, both intentionally and unintentionally. This includes, for example, giving clients access to only the resources they need to operate (principle of least privilege).

4.2 Design

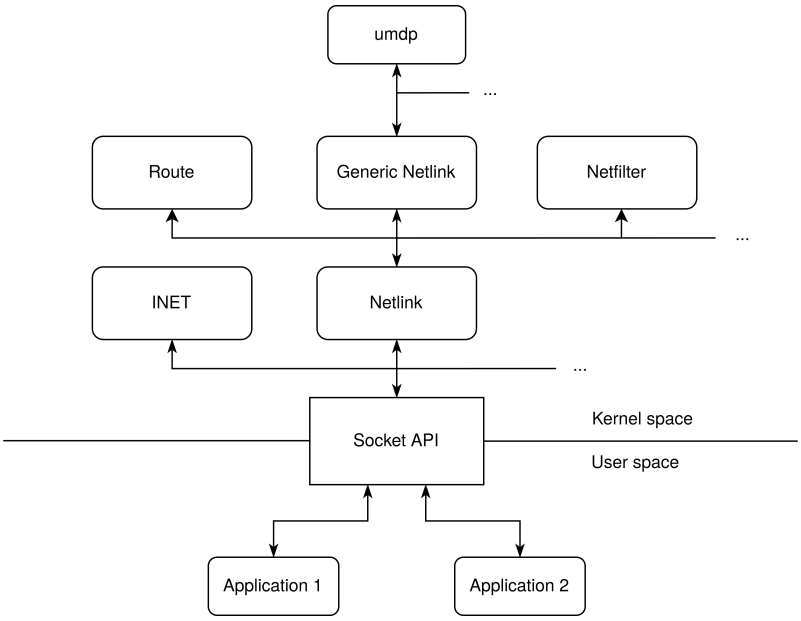
To add this functionality to Linux, we'll need to write kernel level code. As such, our system, which we named Usermode Driver Platform, or umdp for short, will be composed of a kernel module, which will provide the functionality, and a user space library, which will handle the communication with the kernel module, and will provide a nicer interface for the user.

4.3 Implementation

4.3.1 Netlink

We need a way for the kernel module and the user space to communicate, and to allow a user space program to send requests to the kernel module. Normally, Linux programs mainly talk to the kernel through syscalls, but that's not an option for us, since kernel modules can't add syscalls (the kernel itself would need to be patched), and it wouldn't be compatible with future kernel releases (where a new syscall is added). There is, however, another option.

Figure 4.1: Overview of Netlink communication



Netlink [6, 7] is a message passing protocol that allows communicating between kernel and user space over a socket connection. It provides a mechanism called Generic Netlink, which allows registering new protocols (Generic Netlink families) on top of Netlink dynamically.

Our kernel module registers its own Netlink family, through which it provides its functionality. A description of this protocol can be found in appendix B.

On the library side, we use the `libnl` library to handle the Netlink protocol and communicate with the kernel module.

4.3.2 Port I/O

To read and write to I/O ports, Linux provides the `in*` and `out*` family of functions, respectively. These are mainly for use in the kernel, but user space programs can use them, if they have the appropriate permissions [25].

To grant a user space program the ability to use these functions, one needs to call `ioperm` while having the `CAP_SYS_RAWIO` capability (or while being a privileged process [26]) [27]. Ideally, we'd like to avoid this, since a process with this capability can read and write to any I/O port, so we added an interface to delegate this to the kernel module, as shown in listing 1.

Internally, the kernel module simply calls the corresponding `in*` or `out*` function.

There is, still, another part to this, as the kernel module also needs to request access to a specific port (or, rather, a region of ports) before being able to read and write to it, through the use of the `request_region` kernel function. Thus, a couple of functions were added (the last two in listing 1) to allow the user to request (and later release) access to the ports they want to use, after which they'll be able to use the other functions.

```
int umdp_devio_read_u8(umdp_connection* conn, uint64_t port, uint8_t* out);
int umdp_devio_read_u16(umdp_connection* conn, uint64_t port, uint16_t* out);
int umdp_devio_read_u32(umdp_connection* conn, uint64_t port, uint32_t* out);

int umdp_devio_write_u8(umdp_connection* conn, uint64_t port, uint8_t value);
int umdp_devio_write_u16(umdp_connection* conn, uint64_t port, uint16_t value);
int umdp_devio_write_u32(umdp_connection* conn, uint64_t port, uint32_t value);

int umdp_devio_request(umdp_connection* conn, uint64_t start, uint64_t size);
int umdp_devio_release(umdp_connection* conn, uint64_t start, uint64_t size);
```

Listing 1: The port I/O API

4.3.3 Interrupt handling

Dealing with hardware interrupts is slightly more challenging. Our kernel module has to receive interrupts on the behalf of the user, then send a notification to the program that an interrupt occurred.

Upon receiving a request to subscribe for interrupt notifications of a certain IRQ, we need to install an interrupt handler. The interrupt handler is a function that's executed whenever an interrupt occurs in the corresponding IRQ line, and it should perform any necessary tasks in response to that event. For our purposes, we just need to send a notification to user space.

However, we cannot do this from the interrupt handler itself, as "a handler runs at interrupt time and, therefore, suffers some restrictions on what it can do. [...] A handler can't transfer data to or from user space, because it doesn't execute in the context of a process. Handlers also cannot do anything that would sleep, such as calling `wait_event`, allocating memory with anything other than `GFP_ATOMIC`, or locking a semaphore. Finally, handlers cannot call `schedule`." [28, p. 269] There are no guarantees that the Netlink implementation obeys these restrictions. Sending a message to user space directly is, thus, out of reach.

What we *can* do is have the handler store the IRQ number and hand the task of sending a notification to a workqueue. A workqueue is a mechanism of the Linux kernel for performing work asynchronously [29].

To deliver the interrupt notification to the user space program, it is multicasted to a dedicated multicast group of our Netlink family, sending it to every program using our interface. In the user space side, our library subscribes to this multicast group, then processes the received interrupt notifications and filters out the ones that have IRQ numbers that that program didn't subscribe to.

Multicast was chosen because it was simpler to implement, as it doesn't require keeping track of the port IDs of the clients, and it only requires sending a single message. With the work done in section 4.3.5, this could be revisited and changed to send unicast notifications to only the right clients.

```
int umdp_interrupt_subscribe(umdp_connection* conn, uint32_t irq);
int umdp_interrupt_unsubscribe(umdp_connection* conn, uint32_t irq);
int umdp_receive_interrupt(umdp_connection* conn, uint32_t* out);
```

Listing 2: The interrupt handling API

4.3.4 Memory-mapped I/O

Finally, we want to allow processes to map physical memory regions (where a hardware device is mapped to) to their own address space.

Linux already allows processes to access (and thus map) physical memory through `/dev/mem` [30]. However, it's only available to privileged processes [26], and some kernel configuration options, such as `CONFIG_IO_STRICT_DEVMEM`, further restrict this. Providing our own interface to map physical memory would let us avoid these restrictions, avoid requiring privileged processes, and allow better control over physical memory accesses.

With access to process information, we can get to the corresponding virtual memory structures, and, then, we could possibly edit them to add the physical memory mapping we want. Setting up the memory map ourselves, though, isn't feasible. "When a user-space process calls mmap to map device memory into its address space, the system responds by creating a new VMA to represent that mapping. [...] Note that the kernel maintains lists and trees of VMAs to optimize area lookup, and several fields of `vm_area_struct` are used to maintain this organization. Therefore, VMAs can't be created at will by a driver, or the structures break." [28, p. 420]

If we must go through the `mmap` syscall to set up a memory map, we need to provide our own character device that the user process can use `mmap` with. Thus, the character device needs to implement `mmap`, and in its implementation, it should interpret byte addresses as physical memory addresses like `/dev/mem` does [30].

With this done, we added an interface to our library that opens our device and calls `mmap`, thus allowing users to map physical memory.

```
int umdp_mmap_physical(umdp_connection* conn, off_t start, size_t size, void** out);
```

Listing 3: The mmap API

4.3.5 Mapping Netlink ports to processes

When the kernel module receives a Netlink request, it has no easy way to determine the process that sent it. The main information available to it about the sender is the port ID of the used Netlink socket. This port ID is specific to Netlink, and there is no available interface to obtain anything given the port ID number.

```
struct partial_netlink_sock {
    struct sock sk;
    unsigned long flags;
    u32 portid;
    // ...
};
...
struct sock* socket = ...;
struct partial_netlink_sock* nl_socket =
container_of(socket, struct partial_netlink_sock, sk);
u32 port_id = nl_socket->portid;
...
```

Listing 4: Hack to obtain the port ID out of a `struct sock*`

It would be nice to know keep track of the process associated to each port ID, as it would allow, for example, releasing its held resources if the process exits without doing so, or for implementing restrictions on the uses of the various interfaces.

One approach to accomplish this, as we can't get the process given its port ID, would be to go through the running processes and figure out which one has an open Netlink socket with that port ID.

To figure out if a process has a Netlink socket with a certain port ID, we need to look at its open file descriptors, find out which of those are sockets, and which of those are Netlink sockets. Once we get to this point, though, we face a similar issue: the port ID of the Netlink socket is stored in a private part of the socket structure (the full `struct` definition is in a private header of the Netlink implementation).

It's possible to hack our way around this (unless `struct` layout randomization is enabled, and `__randomize_layout` is set for `struct netlink_sock`, which is not the case on the current version of Linux) by replicating the `struct` definition on the side of our kernel module, then using the `container_of` macro to obtain the full structure.

The last piece of the puzzle is knowing which process's file descriptors to look through. Going through every running process isn't exactly ideal in terms of performance. We can have the user space library help the kernel module with this, by making it *tell the kernel module* what process ID (PID) its process has. The kernel module can then check, as described above, if the indicated process owns the socket that sent the request.

This is done by introducing a "connect" request (containing the PID) to our Generic Netlink protocol, which is sent automatically by the user space library when a connection is created. The kernel module keeps a list of the connected clients, associating Netlink port IDs with process IDs and other information.

A limitation of mapping Netlink sockets to processes is, in case a process with an open socket `fork`s without closing the socket afterwards, we'll end up with two processes sharing a socket,

which could cause our system to behave in an unexpected way from the perspective of said programs. We can detect this on the library side and handle it gracefully there.

4.3.6 Cleanup

By keeping a record of the client processes, we can release their resources if they exit without releasing them themselves (for example, in case of a crash or a poorly-coded client).

The Linux kernel doesn't provide a specific interface that can notify kernel modules when a process exits. It is, however, possible to accomplish this by using a Kprobe. Kprobes (Kernel Probes) are a mechanism that allow the user to register a function to be executed at a specific point (such as the start or end) of almost every kernel routine and collect information, primarily used for debugging and performance analysis [31].

We can set up a Kprobe on the `do_exit` kernel function (implemented in `kernel/exit.c` in the Linux source code) to run code whenever a process exits. `do_exit`, and thus our handler function, runs in the context of the process that's exiting, so we can identify the process by looking at global item `current`, which is effectively a pointer to the `struct task_struct` that contains information about the process [28, pp. 21–22]. We can, then, look for the record that matches this process, and if there is one, we remove it and release the associated resources.

4.3.7 Access control

With the functionality to write device drivers in place, we can shift our focus to a different topic. One of our main goals is to improve system reliability and security by minimizing the resources accessible to programs to only what they require to function. Our system doesn't impose any restrictions yet, so the next step will be to fix that.

Before that, though, let's take a quick look at how MINIX 3 handles this. It makes use of a global configuration file, `/etc/system.conf`, which contains configuration for the primary system services, and can control parameters such as allowed IRQs, I/O port ranges, PCI devices and kernel calls (additionally, similar config files can be used to control specific services) [32].

Now, add restrictions to our interfaces is the easy part, as it's just a matter of placing a check in the right position and refusing the request if the check fails. The interesting parts are the methods used to implement the checks, and the interface used to configure them.

4.3.7.1 Path-based permissions

There are many valid options that could be used for modeling permissions. To keep things simple, we chose a global table where the system administrator can specify the permissions of a program given its (canonical) file path.

The downside is that a poor configuration, such as granting permissions to a file in a globally accessible directory, could compromise the security of the system, requiring the system administrator to use caution in configuring it. Still, we consider this trade-off worth it for our purposes.

```
# cat /proc/umdp/permtab
Executable path          IRQ lines   mmap regions      I/O port regions
/home/vbox/timer         0           none              0x40-0x41,0x43-0x44
/home/vbox/framebuffer   none        0xA0000-0xC0000   none
# echo '/home/vbox/keyboard 1 none 0x60-0x61,0x64-0x65' > /proc/umdp/permtab
# cat /proc/umdp/permtab
Executable path          IRQ lines   mmap regions      I/O port regions
/home/vbox/keyboard      1           none              0x60-0x61,0x64-0x65
```

Listing 5: Reading and writing to `permtab` (formatted for readability)

4.3.7.2 Exposing the configuration

As we saw earlier, MINIX 3 makes use of a config file. Such an approach is, however, frowned upon in Linux kernel programming [33]. We'll instead use procfs to control our permission settings.

procfs is a pseudo-filesystem containing data structures exposed by various kernel modules and parts of the kernel. While most of it is read-only, some parts are writable, allowing users to change some kernel settings [34].

We implemented a file, `/proc/umdp/permtab`, using the `seq_file` interface [35], which displays the current rules in table form when read, and replaces the existing rules with new ones when written to, as shown in listing 5.

4.3.7.3 Applying the rules

Lastly, we need to be able to map these executable paths to our client processes in order to enforce the rules. Fortunately, the hard work has already been done (in section 4.3.5), so we can always identify the process and access its `struct task_struct`, from which we can extract a pointer to the corresponding executable file, which we can then resolve into a file path.

With this, we are able to implement the proposed restrictions, as demonstrated in listing 6.

The documentation for the full user space API can be found in appendix A.

```
$ ./interrupts 12
umdp_receive_interrupt: failed to receive reply: Operation not permitted
umdp_receive_interrupt returned -28
-----
# dmesg -W
umdp: /home/vbox/interrupts not allowed to access IRQ 12, refusing request
```

Listing 6: Example of running a program without the necessary permissions

Chapter 5

Evaluation

To test the functionality described in section 3.1, we wrote two test programs: a PS/2 keyboard driver, which makes use of port I/O and interrupts; and a framebuffer program, which makes use of memory mapped I/O.

The tests were run in a VirtualBox virtual machine running Arch Linux (updated on 2024-07-08), under the 6.9.8 and 6.6.37 kernels.

5.1 PS/2 Keyboard

First, a simple PS/2 keyboard driver. It uses port I/O to set up the PS/2 controller and keyboard, as well as to read data coming from the keyboard, and it uses interrupt notifications to detect when the keyboard has sent data. A sample of the code is shown in listing 7, and a small demo is shown in figure 5.1.

```
static int handle_ps2_interrupts(umdp_connection* conn) {
    while (!should_exit) {
        uint32_t irq;
        TRY_UMDP(umdp_receive_interrupt(conn, &irq), "error receiving interrupt");
        if (irq != PORT1_IRQ) {
            continue;
        }

        uint8_t data;
        TRY_UMDP(umdp_devio_read_u8(conn, DATA_PORT, &data),
            "failed to read from the data port");
        display_scan_code(data);
    }
    return 0;
}
```

Listing 7: The main loop of the keyboard driver

5.2 Writing to the framebuffer

Second, a small program that writes to the VGA framebuffer. It opens the framebuffer device and uses `ioctl` to obtain information about it, such as the resolution and number of bits per pixel and, most importantly, its physical address. It then closes the device, and uses the `mmap` interface to map the framebuffer into memory directly.

After that, it writes a pattern into the framebuffer using `memset`, and exits.

Listing 8 shows some of the code used, and figure 5.2 shows the result.

```
int main(void) {
    ...
    size_t screensize = variable_info.xres * variable_info.yres
        * variable_info.bits_per_pixel / 8;
    ...

    char* fb;
    int ret = umdp_mmap_physical(connection, (off_t) fixed_info.smem_start,
        screensize, (void**) &fb);
    ...

    int colors[6] = {36, 38, 54, 16, 25, 40};
    unsigned int chunk = (variable_info.yres / 6) * fixed_info.line_length;
    for (int i = 0; i < 6; i++) {
        memset(fb + chunk * i, colors[i], chunk);
    }
    ...
}
```

Listing 8: Some parts of the framebuffer program

Figure 5.1: Screenshot of the keyboard driver being executed through SSH on a virtual machine, where the text “test” is inputted

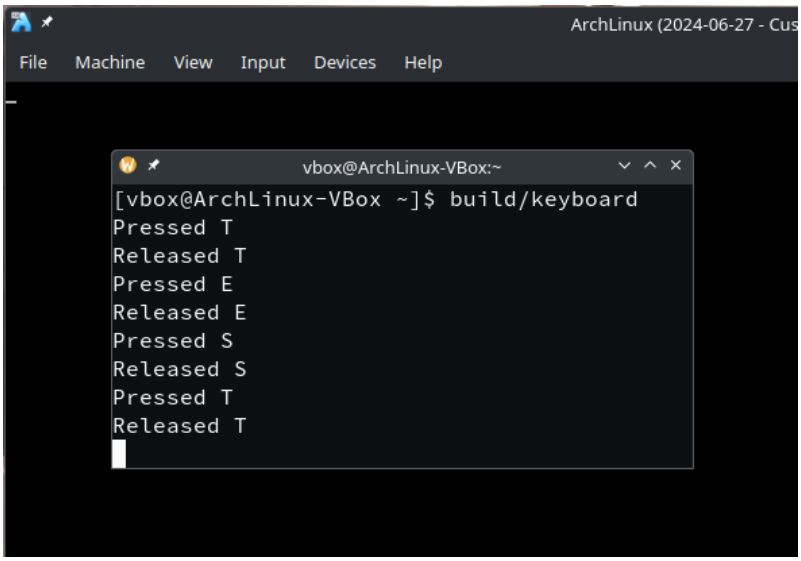
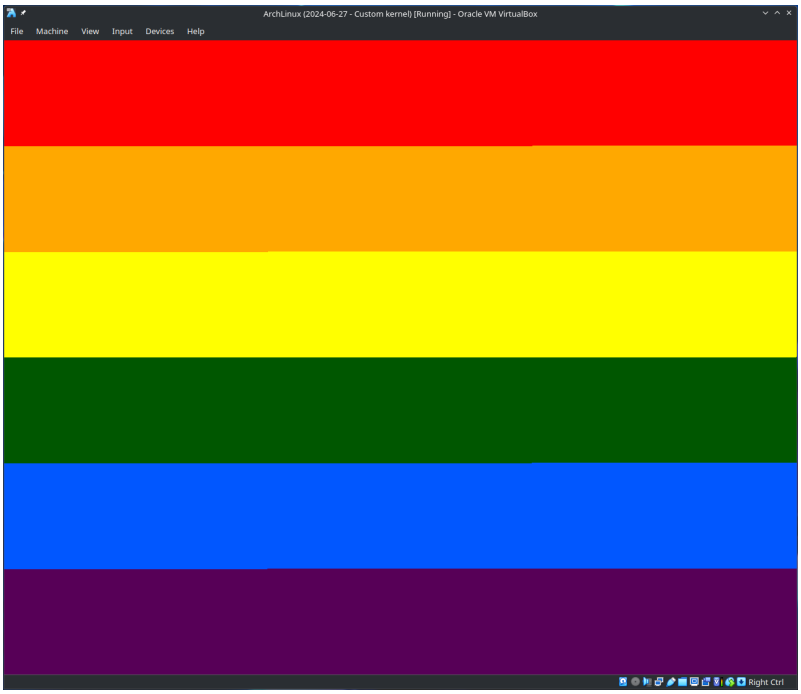


Figure 5.2: Screenshot of a virtual machine after writing a rainbow pattern to the framebuffer



Chapter 6

Conclusions

The programs presented in the last section show that umdp provides enough functionality to demonstrate that writing programs like device drivers in user space, with strong security and reliability guarantees, is possible under the Linux kernel. As such, it illustrates the possibility of moving current operating systems to architectures similar to this one, in order to gain these benefits. We also consider that this project can be, as-is, a useful tool for educational purposes and some real-world use cases.

6.1 Future Work

This system can be expanded to handle more kinds of software devices, as well as to integrate better with existing drivers and other subsystems of the kernel (by, for example, adding DMA-BUF functionality).

Another area that can be further explored is security and access controls. While the current system allows fine control over allowed resources, there's room for improvement, and other different approaches could be explored as well.

Bibliography

[1] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. "MINIX 3: A Highly Reliable, Self-Repairing Operating System". In: *SIGOPS Oper. Syst. Rev.* 40.3 (July 2006). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 80–89. ISSN: 0163-5980. DOI: 10.1145/1151374.1151391. URL: https://doi.org/10.1145/1151374.1151391.

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An Empirical Study of Operating Systems Errors". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. event-place: Banff, Alberta, Canada. New York, NY, USA: Association for Computing Machinery, 2001, pp. 73–88. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502042. URL: https://doi.org/10.1145/502034.502042.

[3] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. "Faults in Linux: Ten Years Later". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. event-place: Newport Beach, California, USA. New York, NY, USA: Association for Computing Machinery, 2011, pp. 305–318. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950401. URL: https://doi.org/10.1145/1950365.1950401.

[4] Simon Biggs, Damon Lee, and Gernot Heiser. "The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-Based Designs Improve Security". In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. APSys '18. event-place: Jeju Island, Republic of Korea. New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 978-1-4503-6006-7. DOI: 10.1145/3265723.3265733. URL: https://doi.org/10.1145/3265723.3265733.

[5] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. event-place: Big Sky, Montana, USA. New York, NY, USA: Association for

Computing Machinery, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[6] *Introduction to Netlink*. The Linux Kernel documentation. URL: https://docs.kernel.org/6.9/userspace-api/netlink/intro.html (visited on 07/01/2024).

[7] Pablo Neira-Ayuso, Rafael M. Gasca, and Laurent Lefevre. "Communicating between the kernel and user-space in Linux using Netlink sockets". In: *Software: Practice and Experience* 40.9 (2010). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.981, pp. 797–810. DOI: https://doi.org/10.1002/spe.981. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.981.

[8] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. "Improving the Reliability of Commodity Operating Systems". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. event-place: Bolton Landing, NY, USA. New York, NY, USA: Association for Computing Machinery, 2003, pp. 207–222. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945466. URL: https://doi.org/10.1145/945445.945466.

[9] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. "XFI: Software Guards for System Address Spaces". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. event-place: Seattle, Washington. USA: USENIX Association, 2006, pp. 75–88. ISBN: 1-931971-47-1.

[10] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. "Fast Byte-Granularity Software Fault Isolation". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. event-place: Big Sky, Montana, USA. New York, NY, USA: Association for Computing Machinery, 2009, pp. 45–58. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629581. URL: https://doi.org/10.1145/1629575.1629581.

[11] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Software Fault Isolation with API Integrity and Multi-Principal Modules". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. event-place: Cascais, Portugal. New York, NY, USA: Association for Computing Machinery, 2011, pp. 115–128. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043568. URL: https://doi.org/10.1145/2043556.2043568.

[12] Bryce Nakatani. *ELJOnline: User Mode Drivers*. 2002. URL: https://web.archive.org/web/20090107040456/http://www.linuxdevices.com/articles/AT5731658926.html (visited on 01/05/2024).

[13] Peter Chubb. "Get More Device Drivers out of the Kernel!" In: *Proceedings of the Linux Symposium*. Linux Symposium. Vol. 1. Ottawa, Canada, 2004, pp. 149–162. URL: https://www.kernel.org/doc/ols/2004/ols2004v1-pages-149-162.pdf.

[14] Hans-Jürgen Koch. *The Userspace I/O HOWTO*. The Linux Kernel documentation. Dec. 11, 2006. URL: https://www.kernel.org/doc/html/v6.6/driver-api/uio-howto.html (visited on 01/04/2024).

[15] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. "The Design and Implementation of Microdrivers". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. event-place: Seattle, WA, USA. New York, NY, USA: Association for Computing Machinery, 2008, pp. 168–178. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346303. URL: https://doi.org/10.1145/1346281.1346303.

[16] Silas Boyd-Wickizer and Nickolai Zeldovich. "Tolerating Malicious Device Drivers in Linux". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. event-place: Boston, MA. USA: USENIX Association, 2010, p. 9.

[17] Jeff Dike. *The User-mode Linux Kernel Home Page*. URL: https://user-mode-linux.sourceforge.net/ (visited on 01/05/2024).

[18] *FUSE*. The Linux Kernel documentation. 2022. URL: https://www.kernel.org/doc/html/v6.6/filesystems/fuse.html (visited on 01/05/2024).

[19] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. "BDUS: Implementing Block Devices in User Space". In: *Proceedings of the 14th ACM International Conference on Systems and Storage*. SYSTOR '21. event-place: Haifa, Israel. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 978-1-4503-8398-1. DOI: 10.1145/3456727.3463768. URL: https://doi.org/10.1145/3456727.3463768.

[20] Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle. "User Space Network Drivers". In: *Proceedings of the Applied Networking Research Workshop*. ANRW '18. event-place: Montreal, QC, Canada. New York, NY, USA: Association for Computing Machinery, 2018, pp. 91–93. ISBN: 978-1-4503-5585-8. DOI: 10.1145/3232755.3232767. URL: https://doi.org/10.1145/3232755.3232767.

[21] *MINIX 3 Kernel API*. MINIX 3 Wiki. July 8, 2016. URL: https://wiki.minix3.org/doku.php?id=developersguide:kernelapi (visited on 06/28/2024).

[22] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Third Edition. Upper Saddle River, NJ: Pearson Prentice Hall, 2006. 1054 pp. ISBN: 0-13-142938-8.

[23] *Memory grants*. MINIX 3 Wiki. Jan. 25, 2016. URL: https://wiki.minix3.org/doku.php?id=developersguide:memorygrants (visited on 06/28/2024).

[24] Jorrit N. Herder, Herbert Bos, Arun Thomas, Ben Gras, and Andrew S. Tanenbaum. "Memory Sharing Revisited (Work in Progress)". 4th EuroSys Conference. Nuremberg, Germany, 2009. URL: https://www.minix3.org/docs/jorrit-herder/eurosys09-wip-poster.pdf (visited on 06/28/2024).

[25] Alejandro Colomar, Michael Kerrisk, Andries Brouwer, and Rik Faith. “outb(2) - System Calls Manual”. In: *Linux man-pages*. 6.9.1. May 2, 2024. URL: https://www.kernel.org/doc/man-pages/.

[26] Alejandro Colomar, Michael Kerrisk, Andries Brouwer, and Rik Faith. “Capabilities(7) - Miscellaneous Information Manual”. In: *Linux man-pages*. 6.9.1. May 2, 2024. URL: https://www.kernel.org/doc/man-pages/.

[27] Alejandro Colomar, Michael Kerrisk, Andries Brouwer, and Rik Faith. “ioperm(2) - System Calls Manual”. In: *Linux man-pages*. 6.9.1. May 2, 2024. URL: https://www.kernel.org/doc/man-pages/.

[28] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. Third Edition. Sebastopol, CA: O’Reilly Media, Inc., 2005. 615 pp. ISBN: 0-596-00590-3. URL: https://lwn.net/Kernel/LDD3/.

[29] Tejun Heo and Florian Mickler. *Workqueue*. The Linux Kernel documentation. Sept. 2010. URL: https://www.kernel.org/doc/html/v6.9/core-api/workqueue.html (visited on 07/01/2024).

[30] Alejandro Colomar, Michael Kerrisk, Andries Brouwer, and Rik Faith. “mem(4) - Kernel Interfaces Manual”. In: *Linux man-pages*. 6.9.1. May 2, 2024. URL: https://www.kernel.org/doc/man-pages/.

[31] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. *Kernel Probes (Kprobes)*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/v6.9/trace/kprobes.html (visited on 07/01/2024).

[32] Cristiano Giuffrida. *system.conf(5) - File Formats Manual*. Minix Man Pages. URL: https://man.minix3.org/cgi-bin/man.cgi?query=system.conf&apropos=0&sektion=5&manpath=Minix&arch=default&format=html (visited on 06/29/2024).

[33] Greg Kroah-Hartman. *Driving Me Nuts - Things You Never Should Do in the Kernel*. Linux Journal. Apr. 6, 2005. URL: https://www.linuxjournal.com/article/8110.

[34] Alejandro Colomar, Michael Kerrisk, Andries Brouwer, and Rik Faith. “proc(5) - File Formats Manual”. In: *Linux man-pages*. 6.9.1. May 2, 2024. URL: https://www.kernel.org/doc/man-pages/.

[35] Jonathan Corbet. *The seq_file Interface*. The Linux Kernel documentation. 2003. URL: https://www.kernel.org/doc/html/v6.9/filesystems/seq_file.html (visited on 07/01/2024).

Appendix A

User space API

```
typedef struct umdp_connection umdp_connection;

/// Establish a connection to the kernel.
///
/// The `umdp` kernel module needs to be loaded before this function is called.
/// The returned `umdp_connection` should be destroyed after its last use using
/// `umdp_disconnect()`.
///
/// \return Pointer to `umdp_connection` or `NULL` in case of failure
umdp_connection* umdp_connect(void);

/// Disconnect the specified UMDP connection, freeing all associated resources.
///
/// This function also frees the `umdp_connection` struct.
///
/// \param connection Pointer to `umdp_connection`
void umdp_disconnect(umdp_connection* connection);

/// Request access to an I/O port region.
///
/// If the I/O port region is already in use by another driver,
/// it will be released beforehand.
/// Linux requires that I/O regions be released as a whole,
/// so if the I/O region you want is already in use,
/// you must specify it in its entirety, even if you don't intend
/// to use all of it.
///
/// Make sure to release it when it's not necessary anymore
/// using `umdp_devio_release()`.
///
/// \param connection `umdp_connection` to use
/// \param start The first I/O port of the desired region
/// \param size The size of the region (must be greater than 0)
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_request(umdp_connection* connection, uint64_t start, uint64_t size);
```

```
/// Release an I/O port region.
///
/// It must have been previously requested using `umdp_devio_request()`.
///
/// \param connection `umdp_connection` to use
/// \param start The first I/O port of the desired region
/// \param size The size of the region (must be greater than 0)
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_release(umdp_connection* connection, uint64_t start, uint64_t size);

/// Read a byte from the specified port.
///
/// \param connection `umdp_connection` to use
/// \param port Port to read from
/// \param out Pointer to where the read value should be stored
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_read_u8(umdp_connection* connection, uint64_t port, uint8_t* out);

/// Read a 2 byte value from the specified port.
///
/// \param connection `umdp_connection` to use
/// \param port Port to read from
/// \param out Pointer to where the read value should be stored
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_read_u16(umdp_connection* connection, uint64_t port, uint16_t* out);

/// Read a 4 byte value from the specified port.
///
/// \param connection `umdp_connection` to use
/// \param port Port to read from
/// \param out Pointer to where the read value should be stored
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_read_u32(umdp_connection* connection, uint64_t port, uint32_t* out);

/// Write a byte to the specified port.
///
/// \param connection `umdp_connection` to use
/// \param port Port to write to
/// \param value Value to write
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_write_u8(umdp_connection* connection, uint64_t port, uint8_t value);

/// Write a 2 byte value to the specified port.
///
/// \param connection `umdp_connection` to use
/// \param port Port to write to
/// \param value Value to write
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_write_u16(umdp_connection* connection, uint64_t port, uint16_t value);
```

```
/// Write a 4 byte value to the specified port.
///
/// \param connection `umdp_connection` to use
/// \param port Port to write to
/// \param value Value to write
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_devio_write_u32(umdp_connection* connection, uint64_t port, uint32_t value);


/// Subscribe to interrupts from the specified IRQ line.
///
/// The IRQ line must either be free, or in shared mode.
///
/// \param connection `umdp_connection` to use
/// \param irq IRQ line to subscribe to
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_interrupt_subscribe(umdp_connection* connection, uint32_t irq);

/// Unsubscribe from interrupts from the specified IRQ line.
///
/// \param connection `umdp_connection` to use
/// \param irq IRQ line to subscribe to
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_interrupt_unsubscribe(umdp_connection* connection, uint32_t irq);

/// Receive an interrupt notification from any of the subscribed IRQ lines.
///
/// \param connection `umdp_connection` to use
/// \param out Pointer to where the IRQ number should be stored
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_receive_interrupt(umdp_connection* connection, uint32_t* out);

/// Establish a mapping between a physical memory region and the process's
/// address space.
///
/// \param connection `umdp_connection` to use
/// \param start Address of the start of the physical memory region to map
/// \param size Size of the region to be mapped
/// \param out Pointer to location to store the address of the mapped region in
/// (will be set to a valid pointer on success, or NULL in case of failure)
/// \return 0 in case of success, a non-zero value in case of failure
int umdp_mmap_physical(umdp_connection* connection, off_t start, size_t size,
                       void** out);
```

```c
/// Get description of error number.
///
/// The returned pointer should not be `free()`'d by the caller.
///
/// This function can use `strerror()` internally, so the returned pointer
/// could be invalidated by a subsequent call to this function,
/// or to `strerror()` or related functions. As such, if the caller needs
/// to use the string after the immediate moment when this function is called,
/// it should make a copy of the returned string.
const char* umdp_strerror(int error);
```

Appendix B

Generic Netlink Protocol

Table B.1: Attributes used by each command request and reply

Command	Sent by	Contains	Reply	Reply contains
CONNECT	User	PID	Value	CONNECT_REPLY
DEVIO_READ	User	READ_PORT READ_TYPE	Value	READ_REPLY_U8 or READ_REPLY_U16 or READ_REPLY_U32
DEVIO_WRITE	User	WRITE_PORT (WRITE_VALUE_U8 or WRITE_VALUE_U16 or WRITE_VALUE_U32)	ACK	n/a
DEVIO_REQUEST	User	START SIZE	ACK	n/a
DEVIO_RELEASE	User	START SIZE	ACK	n/a
INTERRUPT_NOTIFICATION	Kernel	IRQ	None	n/a
INTERRUPT_SUBSCRIBE	User	IRQ	ACK	n/a
INTERRUPT_UNSUBSCRIBE	User	IRQ	ACK	n/a

Table B.2: Attributes accepted by each command, as well as their type

Command	Attributes	Type
CONNECT	PID	s32
	CONNECT_REPLY	u8
DEVIO_READ	READ_PORT	u64
	READ_TYPE	u8
	READ_REPLY_U8	u8
	READ_REPLY_U16	u16
	READ_REPLY_U32	u32
DEVIO_WRITE	WRITE_PORT	u64
	WRITE_VALUE_U8	u8
	WRITE_VALUE_U16	u16
	WRITE_VALUE_U32	u32
DEVIO_REQUEST	START	u64
DEVIO_RELEASE	SIZE	u64
INTERRUPT_NOTIFICATION INTERRUPT_SUBSCRIBE INTERRUPT_UNSUBSCRIBE	IRQ	u32