Proseminar "The Rust Programming Language"

Interprocess Communication in Rust

Vasil Sarafov
sarafov@cs.tum.edu
Technische Universität München
Department of Informatics

10.06.2017

Abstract

This paper is a general introduction to the basic interprocess communication facilities that can be found on every UNIX-like operating system. We examine how mechanisms such as shared memory, signals, regular files, pipes, UNIX domain and network sockets can be implemented in Rust, which is a new systems programming language developed by Mozilla. Furthermore, we show how existing OS APIs can be wrapped in safe Rust bindings so that we can take full advantage of Rust's strong type system and compile-time checks and hence prevent serious failures that could occur in C and C++ codebases. In addition, we briefly explore three projects from the Rust community that utilise IPC as their core concept, one of which is a full fledged operating system with a microkernel design.

Keywords:
Rust, Interprocess Communication, IPC

1 Introduction

For the past two decades we have been observing the continuous rise of the internet in many completely separate domains. In fact, the internet and the computer as a general technology have been popularised to a degree where they are now part of our everyday lives. Distributed computing, network communication and smart use of the computing power of today's multicore (even multiprocessor) systems are only a small fraction of the mechanisms that make this luxury possible. What these and many more techniques share in common is that they all rely on **interprocess communication** (IPC in short).

Interprocess communication is a way for processes to cooperate with each other and exchange data. Depending on the desired type of cooperation (e.g. narrow-band, low latency, time critical), different IPC mechanisms can be applied.

A large portion of systems that utilise IPC is written in low level programming languages such as C and C++ because of execution speed concerns. However, speed is not the only relevant property. Safety is of great importance too. It is a well known fact that manually managing the working memory, which is the case with C and C++, reduces the system's safety because of lower protection against memory leaks, incorrect pointer arithmetic, reuse of already freed memory segments and many more types of bugs [13, 18]. On the other hand using a programming language with full garbage collection runtime system can be critical for the performance [23]. Therefore we are interested in simultaneously achieving speed and safety without sacrificing one of those properties.

In 2015 Mozilla Research released the first stable version of their new programming language Rust [4]. Rust is a systems programming language which promises of being a safer, more performant alternative to C and C++. Moreover, it eliminates the must of manually managing the memory. In the core of this achievement is Rust's *ownership system* that leverages *affine types*, *region analysis* and *privacy* [14]. It allows deriving the needed memory management operations at compile time and furthermore it makes possible capturing many critical runtime errors at compile time.

This paper examines whether and how exactly Rust can be used for applying the basic IPC mechanisms that are supported by a traditional UNIX-like operating system. The following work is intended as a general introduction to IPC and thus does not cover advanced aspects of IPC such as security, performance optimisation and best practices for designing interprocess communication systems. It guides the reader through the elementary IPC concepts without assuming any deep prior knowledge in this field.

Chapter 2 covers the basics that are needed to follow the paper. This section can be skipped by readers who already have fundamental knowledge in operating systems and are familiar with Rust.

Chapter 3 reviews separately each IPC technique by explaining how it is handled by the operating system kernel and then presenting its Rust implementation. This section starts with shared memory and continues with message passing mechanisms. Additionally, it is shown how the C implementation of some interprocess communication facilities can unintentionally lead to serious issues.

Chapter 4 gives an example of systems written in Rust that utilise IPC.

At the very end a conclusion is made which in addition points out possibilities for future work on this topic. An extra appendix is included, listing useful community crates that can be used when applying IPC concepts in Rust.

Related Work

Many learning resources and scientific papers have been written for IPC. For example, an in-depth introduction to interprocess communication and detailed comparison between the different mechanisms can be found in [12].

Most of the resources about IPC doubtlessly apply the C programming because of obvious reasons. Others use higher level programming languages such as Java [21]. However, only a few, even none at the time of writing, have tried to use Rust and show its advantages as a new systems programming language in the field of IPC.

In [22] is shown that Rust offers a tremendous productivity gain over C with negligible performance losses in high-performance computing systems. [15] describes how Rust can be used for implementing an operating system module for UNIX-like process management. Furthermore, [8] presents and examines a Rust implementation of a low level networking stack.

2 Basics

Processes

Informally speaking, a process is a program in execution [19]. It is the main work unit on a computer system. Strictly defined, it is an active entity that needs a subset of the system's hardware resources in order to complete its task. Nowadays it is presumed that the management of the system's resources is done by the operating system. One of its main jobs is provide an isolated, fair and secure environment for each running process.

However, there are cases in which the isolation between the different independent processes can be a

drawback [19, 6]. For instance we might want to achieve:

- **Information sharing** - Several processes might be interested in the same piece of information (for example, a shared file).
- **Computation speedup** - If the computation of a given task is expensive but it is already done by another process, it is not worth repeating the calculation. We can simply share the result and save valuable CPU time.
- **Modularity** - Because of software design decisions we might be forced to construct our system in a modular fashion, dividing the different functionalities into separate processes.
- **Convenience** - It is assumed that on a typical desktop system the user should be able to perform multiple tasks in parallel. An example is editing a file and automatically printing it every time a change has been made.

In that case we say that the participating processes are cooperating. This can be achieved by applying different *interprocess communication techniques* provided by the operating system kernel. For more detailed information on how a process is generally managed by the kernel, please refer to [19, 6].

Rust

The Rust programming language is a fairly new systems programming language, designed and developed by the Mozilla Foundation. Its aim is to provide *efficiency*, *guaranteed memory and thread safety* with zero-cost abstractions and a minimal runtime [2].

Despite the fact that first stable version of the language was released only in 2015, Rust has been heavily used in the industry [4, 1]. The zero-cost abstractions and emphasis on secure and reliable multithreaded code make Rust very attractive also for the scientific community with high performance needs [7].

Since a large portion of the team behind Rust is heavily influenced by C++, many similarities between both languages such as unique pointers and operator overloading can be found. However a lot of fundamental functional programming concepts, typical for languages like OCaml and Haskell, can be found in Rust too [17]. For a top-down overview of the language's syntax please refer to [3].

Something differentiating Rust from many other programming languages is the `unsafe` keyword. It is used to make an explicit separation between *trusted* code, which is bound to the type system, and *untrusted* code, which can potentially violate the type system (e.g. dereference a raw pointer) [14]. One reason why Rust heavily relies on trusted code is because it makes possible for the compiler to derive precise information about the lifetime of the different data entities. Hence it can automatically determine when they have to be deallocated at compile time and resources can be freed without garbage collection (e.g. freeing heap memory, automatic closing of file descriptors).

As already mentioned, Rust compiles to machine code and aims to operate efficiently and ergonomically on top of the operating system layer. Because of that it preserves the abstract machine model known from the C programming language and similarly has no runtime environment [17]. The communication between Rust's facilities and the host operating system is achieved via the `libc` crate [1], which exposes the raw platform specific system calls. Hence `libc` can be thought of as "the glue" between our code and the operating system kernel, and as we will see, it plays a huge role in understanding how IPC is implemented in Rust.

The Rust code that can be found in Chapters 3 and 4 is compiled and tested with `rustc 1.17.0 (0x56124baa9 2017-04-24)`. All external crates and their versions are explicitly named when used.

[1] Appendix A1

3 Mechanisms for IPC

In this chapter we will examine the main techniques for interprocess communication. For the sake of simplicity, we are targeting a UNIX-like operating system and thus no "bare metal" process execution is being considered.

Starting with shared memory we will afterwards take a closer look at the different message passing mechanisms.

3.1 Shared Memory

The easiest and fastest way for two or more processes to communicate with each other is to share memory. It is fast because there is no additional action performed by the kernel to orchestrate the communication. The only thing that is needed is to correctly manage the memory, which is already done since the involved processes are executing. However, the developer must ensure a secure and safe communication between the processes by utilising synchronisation mechanisms and/or defining a specific set of messaging rules.

Direct addressing of the shared memory blocks is not possible because every process is assigned a virtual address space managed by the kernel. Thus the shared memory has to be allocated and then mapped to the virtual address space of each process (Figure 1).

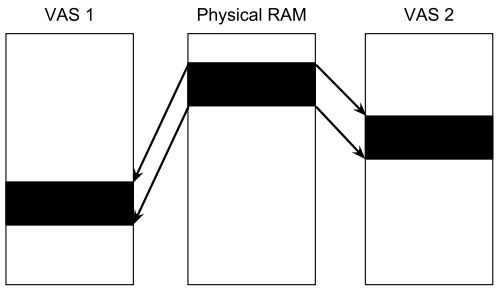

Figure 1: *Mapping a region of the physical RAM to two separate virtual address spaces.*

Theoretically speaking there are two types of memory mappings:

1. **Anonymous** - Used when sharing memory between processes with common ancestry. The shared segment is created by the root of the process family and thus can be *implicitly* identified and accessed by the children.

2. **Memory mapped files** - Used when sharing memory between arbitrary processes. The shared segment is created and bound to an empty file handle. Then it can be *explicitly* identified and mounted by other processes. No content is being written to the file system when manipulating the memory map, although it is identified by a file handle. For example, under Linux a memory mapped file would be an empty INode, pointing to a specific memory region.

On POSIX compatible operating systems only the second type of memory mapping exists [9].

At the time of writing Rust has surprisingly no support for memory mapped files in its standard library. In version 0.8 [2] cross-platform support for memory maps was introduced with the `std::os::MemoryMap` module. Two years later, when the first stable version of the language (1.0) was released, the module was removed. Luckily, the Rust community has managed to compensate the absence of an API for cross-platform memory-mapped IO and developed the `memmap` crate [3].

Below we present a short implementation of a unsynchronised producer that creates a memory map and writes data in it. A consumer opens it and reads the written data. We are using version 0.5.2 of the `memmap` crate.

[2] `https://github.com/rust-lang/rust/blob/master/RELEASES.md#version-08-2013-09-26`

[3] Appendix A3

```rust
static MAP_LEN: u64 = 4096;
static MAP_ID: &'static str =
    "/tmp/ripc-shared.memory";

pub fn producer() {
    let handle = OpenOptions::new()
            .read(true)
            .write(true)
            .create(true)
            .truncate(true)
            .open(MAP_ID)
            .unwrap();
    handle.set_len(MAP_LEN).unwrap();

    let mut map = Mmap::open(
        &handle, Protection::ReadWrite
    ).unwrap();

    unsafe { map.as_mut_slice() }
        .write(b"TUM")
        .unwrap();
    println!("Wrote {:?}", b"TUM");
}
```

On the producer side, a file handle for exactly 4096 bytes is created with read/write permissions. Then it is mapped to a memory segment and the binary representation of the string `"TUM"` is written. Hence the mutable binding `let mut map`. The `unsafe` block is needed to cast the memory map into a byte slice because a direct manipulation of the underlying memory is performed, which is considered unsafe by the Rust type system. An important note is that the operating system will discard the memory page containing the map as soon as the producer process exits.

```rust
static MAP_LEN: u64 = 4096;
static MAP_ID: &'static str =
    "/tmp/ripc-shared.memory";

pub fn consumer() {
    let handle = OpenOptions::new()
            .read(true)
            .open(MAP_ID)
            .unwrap();

    let map = Mmap::open(
        &handle, Protection::Read
    ).unwrap();

    let mut buf: [u8; MAP_LEN]
        = [0; MAP_LEN];
    unsafe {
        let mut mem = map.as_slice();
        mem.read_exact(&mut buf).unwrap();
    }
    println!("Read {:?}", buf);
}
```

On the consumer side, the memory map is opened only with read permissions because no writing is performed. Hence the immutable `let map` binding. After that the whole content of the map (4096 bytes) is *unsafely* read and printed to the standard output. The first three bytes are exactly the binary representation of the string `"TUM"`.

As we see mapping memory with Rust is possible and not that hard. Unfortunately, this is not the case when working directly with the physical memory address space, which is often the case with embedded systems. Embedded platforms are highly event-based and Rust's memory safety mechanisms largely presume threads. Thus Rust's ownership system conflicts with the reality of sharing memory-mapped hardware resources in a single threaded event-driven application, where race conditions are not possible [14].

3.2 Signals

Signals are one of the fastest and most simple forms of IPC. They are short asynchronous notifications that are sent from the OS kernel to a given process (or a specific thread inside a process) when an event occurs (Figure 2).

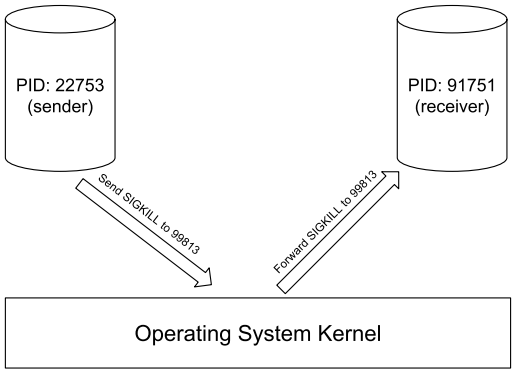

Figure 2: *When a process sends a signal, it is received by the kernel and forwarded to the receiver.*

Signals are fast because they do not have any data attached to them. They are just a code number that is sent to the receiver. Because of this their usage can be limiting. It is up to the receiving process to decide how to interpret the incoming signal code and to register a custom signal handler when the default response action defined by the kernel is not suitable.

Very similar to memory maps (Chapter 3.1), at the time of writing Rust has no support in the standard library for working with signals. Back in the early days when the language had a runtime system with support for green threads [4], the module `std::io::signal` was providing cross-platform API for sending and handling signals. After the core team took a decision to completely remove the runtime environment [5] and support native kernel threads only, everything that was implemented with green threads was deleted, including the `std::io::signal` module [6]. Nevertheless, there is currently an ongoing proposal [7] for solving this issue and bringing native support for signals back in `std`.

At the moment the only solution is to use raw and thus unsafe `libc` bindings to directly execute the respective operating system calls. A couple of community crates exist which have already achieved this in different forms. An interesting package is `chan-signal` [8] that provides synchronous signal handling using channels.

Below we present a short implementation of how to safely send a signal to a child process in a typical `fork()` scenario with Rust using version 0.8 of the `nix` crate [9]. Furthermore, we compare it with its respective implementation in C to show how a raw system call can be wrapped in a safe Rust API [16].

The C code for our example is the following:

```
1  int main() {
2      pid_t child = fork();
3      if (child) { // in parent
4          // do work
5          kill(child, SIGKILL);
6          // do work
7      } else { // in child
8          // do work
9      }
10     return 0;
11 }
```

At first glance it looks like there is nothing wrong with the above code. It even compiles without any warnings. However, it has one serious flaw that will be exploited on very rare occasions. If the `fork()` system call fails, for example because the system cannot allocate enough memory for the child process, it will return -1 [10]. Thus the if statement on line 3 will be entered and after the parent process has finished its work, a `SIGKILL` will be send to the process with `pid` -1. According to the POSIX specification the following will happen [11]:

[4] The term "green threads" is used to describe threads running in user space and managed by the programming language's runtime environment. More information can be found in [19, 6].

[5] `https://github.com/rust-lang/rfcs/pull/230`

[6] `https://github.com/rust-lang/rust/pull/17673`

[7] `https://github.com/rust-lang/rfcs/issues/1368`

[8] Appendix A4.

[9] Appendix A2.

> If **pid is -1, sig shall be sent to all processes** (excluding a unspecified set of system processes) for which the process has permission to send that signal.

This basically means that if the above code is executed with `root` privileges and `fork()` happens to fail, all processes on the system will be killed, which probably is not the intended behaviour of the program.

The root cause for the problem with the above implementation, which is very common for a lot of C and old C++ code bases, is that the return value of `fork()` is assigned three different meanings [10]:

- A positive integer, indicating the child's pid, is returned to the parent process
- Zero is returned to the child process
- Negative one is returned when the call fails

Of course we could fix the bug by simply inserting an additional `if` statement to check whether the returned value is less than zero. This, however, will be normally done after we had executed our program and saw it misbehaves. Hence we capture the bug at *runtime*.

Thankfully with Rust we can detect this kind of failures at *compile time*:

```
1  pub fn fork() -> Result<ForkResult, i32>{
2      // wrapping of the libc's fork()
3  }
4
5  pub enum ForkResult {
6      Parent { child: pid_t },
7      Child
8  }
9
10 fn main() {
11     match fork().expect("failed") {
12         ForkResult::Parent{child} => {
13             // do work
14             kill(child, SIGKILL)
15             .expect("failed")
16         },
17         ForkResult::Child => {
18             // do work
19         }
20     }
21 }
```

By wrapping `fork()` inside the Rust's specific `Result<>` enumeration on line 1, we enforce the caller to unpack the result and check for errors (line 11). This inspection is made at compile time, meaning the program will not compile if the needed error checks are not made. In our case we are using the `expect()` function, which will cause the program to exit immediately if `fork()` fails.

Moreover, we separate the parent/child return values with the additionally defined `pub enum ForkResult`. It helps us make our code more expressive, easier to read and hence less error-prone (line 11 to 19).

The same technique for safe wrapping is applied to the `kill()` system call, used on line 14.

3.3 Regular Files

The contents of regular files are written to the file system which can be stored on an external storage such as HDD, SSD, Flash Memory, etc. Hence regular files can be used as persistent message buffers inside an interprocess communication pool. A typical example is when downloading a file from the network [10]. However, the fact that the operating system writes to the file system makes them a slow communication medium for time-critical process cooperation.

The standard library of Rust provides a cross-platform support for file operations. A reference to a file on the file system is an instance of the `std::fs::File` struct which implements the core

[10] In this case the network is a distributed interprocess communication pool. That is, the participating processes could be on different hosts.

`std::io::Read` and `std::io::Write` traits.

The following code shows how we can create and manipulate a sample log file using Rust (import statements omitted for brevity):

```
static LOG_FILE: &'static str
        = "/tmp/ripc-printer.log";

pub fn inspect_log() {
    let buf = &mut String::new();
    let _ = File::open(LOG_FILE)
        .expect("Could not open")
        .read_to_string(buf)
        .expect("Could not read");
    println!("Log content: {}", buf);
}

pub fn create_log() {
    let mut f = File::create(LOG_FILE)
        .unwrap();
    f.write_all(b"Log file created.");
}
```

What makes Rust special when working with files is that there is no need to worry about closing file descriptors. Again thanks to the ownership system, the compiler is able to extract the needed information and insert the closing statements. This is something which even high level programming languages with automatic memory management such as Java, Go, Ruby, Python and many more don't have.

3.4 Anonymous Pipes

A pipe is a mechanism for a process to consume byte wise the output of another process in a parallel or time-sliced fashion. Hence pipe communication is orchestrated by the operating system kernel (Figure 3).

An important note is that pipes provide unidirectional communication (the consumer cannot send data back to the producer). Another remark is the

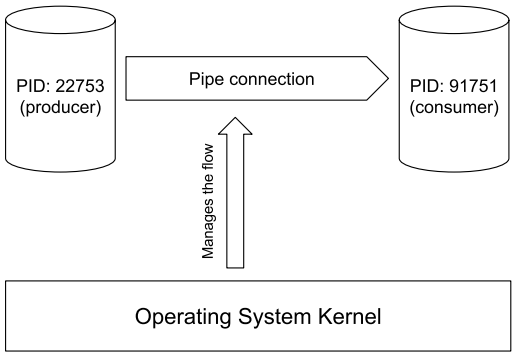

Figure 3: *Unidirectional byte stream communication between two processes via pipe connection, managed by the operating system kernel.*

difference between regular files and pipes: while files store their contents on the file system, pipes do not. They buffer the data in memory which makes them faster.

Anonymous pipes are a special kind of pipes. Their lifetime is managed by the kernel, meaning that they are created on demand and automatically destroyed when the producer has terminated and its output has been fully consumed.

In UNIX the "|" character is used to denote a pipe connection between two processes, executed from a shell. For example the `ls /usr/local/sbin | cat` command will execute the `ls` and `cat` programs in parallel. Afterwards all of the output from the `ls` program will be concurrently consumed by `cat`. As soon as `ls` has terminated and `cat` has read the produced output, the operating system kernel will free the allocated pipe resources.

In Rust we can arrange an anonymous pipe to connect a father-child process pair using the `std::process::Stdio::piped()` function. The following code translates the above example in Rust, whereas the `cat` program is replaced with a

simple `println!()` statement:

```
pub fn list_sbin() {
    let ls = Command::new("ls")
        .arg("/usr/local/sbin")
        .stdout(Stdio::piped())
        .spawn().unwrap();

    let buf = &mut String::new();
    ls.stdout.unwrap()
        .read_to_string(buf)
        .expect("Failed to read");
    println!("{}", buf);
}
```

3.5 Named Pipes

The only difference between anonymous (Chapter 3.4) and named pipes is how their lifetimes are managed. Named pipes, also known as FIFO files on UNIX, are explicitly created and destroyed.

Creation is done with the `mkfifo()` system call. Just like memory maps (Chapter 3.1), named pipes are identified by an empty file system reference. Hence in order to delete such a pipe, its reference has to be unlinked from the file system.

At the moment Rust does not support named pipes. Therefore the only way to apply them for IPC in a Rust codebase is to use a unsafe system call binding or create a safe wrapper around it as shown in Chapter 3.2. Once created, the named pipe can be opened and operated upon just as if it were a regular file (Chapter 3.3).

3.6 UNIX Domain Sockets

A socket is a communication endpoint that is provided by the operating system (Figure 4). Processes can use sockets to exchange data in different forms and by applying a variety of rules [5].

UNIX domain sockets can be accessed only from processes on the same host and offer major benefits

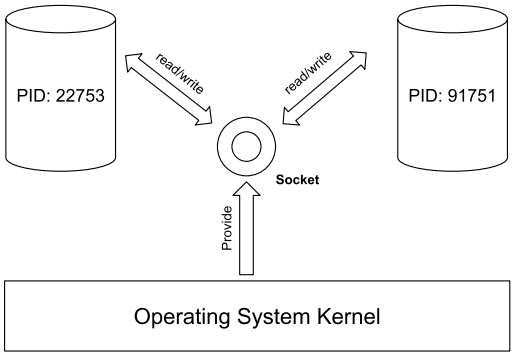

Figure 4: *Two processes communicating bidirectionally via socket, provided by the operating system.*

over the previously discussed IPC message passing mechanisms. They are very similar to named pipes (Chapter 3.5) and therefore share all of their advantages. However, compared to them, UNIX domain sockets are bidirectional and allow exchanging kernel verified data such as credentials, UIDs, GIDs, etc.

In Rust `std::os::unix::net::UnixListener` and `std::os::unix::net::UnixStream` provide full native support for UNIX domain sockets on the server and client side respectively. Once the server has accepted a connection or the client has established one, information can be exchanged by manipulating the underlying data stream which implements the `std::io::Read` and `std::io::Write` traits.

The code below demonstrates a very simple client-server demo using UNIX domain sockets. The server accepts incoming connections and immediately sends the string `"Hello"`. The client receives the string and prints it to `stdout`.

```
static UNIX_SOCK: &'static str
    = "/tmp/ripc-unix.sock";

pub fn unix_server() {
```

```
    let server =
        UnixListener::bind(UNIX_SOCK)
        .unwrap();
    loop {
        match server.accept() {
            Ok((ref mut stream, _)) => {
                stream
                    .write_all(b"Hello")
                    .unwrap();
            }
            Err(_) => {
                println!("Failed.");
                break;
            }
        }
    }
}


pub fn unix_client() {
    let mut stream =
        UnixStream::connect(UNIX_SOCK)
        .unwrap();
    let mut response = String::new();
    stream
        .read_to_string(&mut response)
        .unwrap();
    println!("Received: {}", response);
}
```

3.7 Network Sockets

One of the major differences between domain sockets (Chapter 3.6) and network sockets is obviously the domain the communication endpoint is exposed to. In most cases but not only, network sockets are used to implement a distributed inter-process communication pool. Due to the potential scale and complexity of IPC networks, there are multiple protocols that standardise this type of communication [5].

In Rust the cross-platform `std::net` module is used for utilizing core network communication concepts. On the network layer it provides API for working with IPv4 and IPv6 and on the transport layer both TCP and UDP are supported. Additional utility structures such as `lookup_host()` for issuing DNS queries to resolve a FQDN can be found as well.

A very simple IPv4 TCP echo client/server program is demonstrated with the following code example. The client reads the user's input and sends it to the server, which prints it back to `stdout`. The API for manipulating the underlying data stream is exactly the same as the one shown in Chapter 3.6. Hence the only difference is how the sockets are bound (lines 6 and 31):

```
static TCP_SOCK: &'static str =
    "127.0.0.1:4444";

pub fn echo_server() {
    let server =
        TcpListener::bind(TCP_SOCK)
        .unwrap();
    println!("Listening on {}", TCP_SOCK);

    loop { match server.accept() {
        Ok((ref mut stream, ref addr)) =>{
            let message =
                &mut String::new();
            stream
                .read_to_string(message)
                .unwrap();
            println!(
                "Client {:?} sent {}",
                addr, message
            );
        },

        Err(_) => {
            println!("Connection lost.");
        }
    }}
}

pub fn echo_client() {
    let mut stream =
        TcpStream::connect(TCP_SOCK)
```

```
        .unwrap();

    println!("Please enter your input:");
    let line: String = read!("{}\n");
    stream.write_all(line.as_bytes())
        .unwrap();
    stream.flush().unwrap();
}
```

4 Applications

In this chapter we will take a closer look how IPC with Rust is applied in a couple of real projects. Some of the examples we observe below (4.1 and 4.3) directly use IPC mechanisms in the form discussed in Chapter 3. Others utilise a fraction of these techniques in a different environment (4.2).

4.1 Remote Procedure Calls

Remote procedure calls (RPCs) allow a client to invoke a procedure on a remote host as it would invoke it locally. That is, a given function and its caller could be living on different hosts. This effect is achieved by interacting with a *code stub* on the client side, which delegates the needed data (arguments, control units, etc) in a serialised form to the remote host via a distributed IPC mechanism [19].

A typical usage example for RPC are distributed file systems:

```
// dfs is a dummy module for the
// purpose of the example
let s = b"data";
dfs::write(s);
// the write function is called remotely
// content is written on a different host

let block = dfs::read(0x7aff, 50)
// read 50 bytes from a remote storage
// starting at address 0x7aff
```

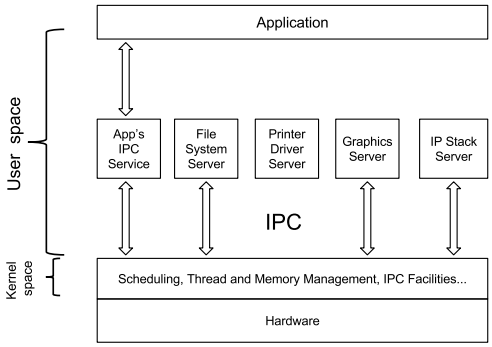

Figure 5: *Schematic overview of the microkernel architecture. The main communication between the kernel and the user space entities is achieved via IPC.*

There exists a community implementation in Rust for the Capn'Proto RPC protocol, which is known to be highly efficient [11]. Thus systems with distributed architecture implementing the protocol can potentially benefit from the rich features that Rust has to offer.

4.2 Microkernels

Microkernel design is an alternative way of structuring an operating system kernel, compared to the standard monolithic approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system- and user-space programs [19, 6]. Microkernels are possible because they provide an advanced communication facility for the user-space processes to use (Figure 5).

The result is a notably more compact kernel, hence the safety-critical surface is smaller. Moreover, a greater flexibility is achieved for exchanging critical components such as network stacks, graphic drivers, etc.

[11] Appendix A5

RedoxOS [12] is a full-fledged UNIX-like operating system written from scratch in Rust, whose kernel has a microkernel design. It is open source (MIT licensed) and currently supports only the x86_64 architecture. Redox aims to bring Rust's features to a modern, efficient and secure microkernel with a large set of applications, which is proven to be a challenging task compared to the standard monolithic kernel architecture because of performance losses [20].

4.3 Platform Communication Stack

A platform communication stack (PCS) provides a standardised way for processes that extend or operate on top of a given platform to communicate with each other by utilising IPC mechanisms. It sounds very similar to the microkernel design approach (Chapter 4.2) because it follows the same idea for decoupling the additional components from the system's core. However, it operates on a higher level - completely in user space.

A typical example for a PCS is a desktop server, where each program that needs graphical resources can connect to the server. All connected programs can then act as peers of each other and share resources if needed.

The desktop environment of RedoxOS - Orbital DE - provides a trivial implementation of a platform communication stack called `launcher` [13].

5 Summary

Rust's strong type system and compile time checks transform many runtime errors into compile time errors, which can prevent critical faults in programs applying interprocess communication. Its unique ownership system and memory safety mechanisms achievable without any runtime overhead provide many advantages over C. Because of this, Rust is suitable for implementing complex and efficiency critical IPC systems.

At the moment Rust lacks support in its standard library for essential IPC mechanisms such as signals and memory mapped IO. However, the community around the language is highly skilled and although not very big, it has managed to fill the gaps between the standard library and the developer's needs with additional open source crates. On the other hand, regular files, pipes, domain and networking sockets can be easily applied in any Rust codebase because they are present in the standard library.

There are examples for complex systems written in Rust that already utilise or implement IPC stacks. At the centre of those examples is RedoxOS - an operating system with a microkernel architecture that uses interprocess communication as its main working strategy.

6 Further Work

On the technical side, some systems calls essential for IPC such as `mkfifo()` and `signal()` are still not wrapped in safe APIs by the `nix` crate [14]. Thus implementing those safe bindings and contributing them to the already known crates could be of a huge benefit for the community. Another option would be to create a unified framework with common API for the main IPC mechanisms.

Aside from that, an experiment with possibly interesting results would be to benchmark the performance of IPC systems written in Rust and other programming languages such as C, C++, Go and Java that are nowadays often used for implementing systems- and networking software. Finding suitable software architecture designs and evaluating security best-practices when working with IPC is the next stop to bringing the basics described in this paper to complex applications.

[12] https://www.redox-os.org/
[13] https://github.com/redox-os/orbutils
[14] Appendix A2.

IoT is a rapidly emerging domain of today's technological advance. Hence it would be useful to assess Rust's practicalness in the embedded world for implementing communication facilities such as RTOS IPC modules and/or low-level networking stacks.

Appendices

A Useful Crates

1. `rust-lang/libc` - Raw (unsafe) bindings to platform APIs for Rust.

2. `nix-rust/nix` - Safe wrappers of multiple system APIs exposed by `libc` for *nix platforms (Linux, Darwin, BSDs, etc). It is almost the de facto crate for accessing system calls on those platforms.

3. `danburkert/memmap-rs` - Cross-platform Rust API for memory-mapped IO.

4. `BurntSushi/chan-signal` - Synchronous (thread-blocking) signal handling with Rust via channels.

5. `capnproto/capnproto-rpc-rust` - Rust implementation of the Cap'n Proto remote procedure call protocol.

6. `capnproto/capnproto-rust` - Rust implementation of the Cap'n Proto type system for distributed systems.

B Additional Materials

Additional materials accompanying the paper such as more complex code examples for each presented IPC technique can be found on the following link: `https://gitlab.com/v45k0/ripc`.

References

[1] Organizations running Rust in production, 2017. `https://www.rust-lang.org/en-US/friends.html`.

[2] The Rust programming language, 2017. `https://www.rust-lang.org/en-US/`.

[3] The Rust Programming Language - Book, 2017. `https://doc.rust-lang.org/book/first-edition/`.

[4] Announcing Rust 1.0, May 15, 2015. `https://blog.rust-lang.org/2015/05/15/Rust-1.0.html`.

[5] David J. Wetherall Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, fifth edition, 2011.

[6] Herbert Bos Andrew S. Tannenbaum. *Modern Operating Systems*. Pearson, fourth global edition, 2015.

[7] Sergi Blanco-Cuaresma and Emeline Bolmont. What can the programming language rust do for astrophysics? *Proceedings of the International Astronomical Union*, 12(S325):341–344, 2016.

[8] Robert Clipsham. Safe, correct, and fast low-level networking. 2015. `https://octarineparrot.com/assets/msci_paper.pdf`.

[9] Ion Gaztanaga. Memory mapped files and shared memory for C++. *Open Standards*, N2044=06-0114, 2006.

[10] The IEEE and The Open Group. The open group base specifications issue 7, IEEE std 1003.1-2008, 2001-2016. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html`.

[11] The IEEE and The Open Group. The open group base specifications issue 7, IEEE std 1003.1-2008, 2001-2016.

http://pubs.opengroup.org/onlinepubs/9699919799/functions/kill.html.

[12] Patricia K. Immich, Ravi S. Bhagavatula, and Dr. Ravi Pendse. Performance analysis of five interprocess communication mechanisms across UNIX operating systems. *Journal of Systems and Software*, 68(1):27–43, oct 2003.

[13] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Identifying the root causes of memory bugs using corrupted memory location suppression. In *2008 IEEE International Conference on Software Maintenance*. IEEE, sep 2008.

[14] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems - PLOS 15*. ACM Press, 2015.

[15] Alex Light. Reenix: Implementing a Unix-Like operating system in Rust. 2015. http://scialex.github.io/reenix.pdf.

[16] Kamal Marhubi. Easier UNIX systems programming with Rust and nix, 2017. http://kamalmarhubi.com/blog/2016/04/13/rust-nix-easier-unix-systems-programming-3/.

[17] Raphael Poss. Rust for functional programmers. *CoRR*, abs/1407.5670, 2014.

[18] Herbert O. Rocha, Raimundo S. Barreto, and Lucas C. Cordeiro. Hunting memory bugs in c programs with map2check. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 934–937. Springer Berlin Heidelberg, 2016.

[19] Abraham Silberschatz. *Operating System Concepts*. John Wiley & Sons, Inc., sixth edition, 2002.

[20] Neal H. Walfield and Marcus Brinkmann. A critique of the GNU hurd multi-server operating system. *Operating Systems Review*, 41(4):30–39, 2007.

[21] George Wells. Interprocess communication in java. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009, Las Vegas, Nevada, USA, July 13-17, 2009, 2 Volumes*, pages 407–413, 2009.

[22] Florian Wilkens. Evaluation of performance and productivity metrics of potential programming languages in the HPC environment. 2015. https://octarineparrot.com/assets/mrfloya-thesis-ba.pdf.

[23] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. Performance analysis and optimization of full garbage collection in memory-hungry environments. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE 16*. ACM Press, 2016.