Operating Systems II – Student Projects

Andreas Grapentin, Clemens Tiedt, Andreas Polze (Eds.)

Technische Berichte Nr. 142

des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam

Technical Reports of the Hasso-Plattner-Institut for
Digital Engineering at the University of Potsdam

Technical Reports of the Hasso-Plattner-Institut for
Digital Engineering at the University of Potsdam | 142

Andreas Grapentin | Clemens Tiedt | Andreas Polze (Eds.)

Operating Systems II – Student Projects

Universitätsverlag Potsdam

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet via `http://dnb.dnb.de/`.

Universitätsverlag Potsdam 2023
`http://verlag.ub.uni-potsdam.de/`

Am Neuen Palais 10, 14469 Potsdam
Phone: +49 (0)331 977 2533 / Fax: 2292
Email: verlag@uni-potsdam.de

The series **Technical Reports of the Hasso-Plattner-Institut for Digital Engineering at the University of Potsdam** is edited by the professors of the Hasso Plattner Institute for Digital Engineering at the University of Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Layout: Tobias Pape
Print: docupoint GmbH Magdeburg

ISBN 978-3-86956-524-8

Also published online on the publication server of the University of Potsdam:
`https://doi.org/10.25932/publishup-52636`
`https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-526363`

Preface

This technical report presents the results of student projects which were prepared during the lecture “Operating Systems II” offered by the “Operating Systems and Middleware” group at HPI in the Summer term of 2020. The lecture covered advanced aspects of operating system implementation and architecture on topics such as Virtualization, File Systems and Input/Output Systems. In addition to attending the lecture, the participating students were encouraged to gather practical experience by completing a project on a closely related topic over the course of the semester. The results of 10 selected projects are covered in this report.

Projects 1, 5, 6 and 9 were focused on extending different aspects of the “EV3 NinjaStorms”[1] embedded real-time operating system. The system was developed at our group as a lightweight alternative to the Linux distribution running on the LEGO Mindstorms EV3 programmable brick. Although marketed as a toy, the Mindstorms EV3 contains a powerful ARM processor, 64 MB of main memory and standard interfaces such as Bluetooth and network protocol stacks.

Project 1 extended the Memory Management capabilities of NinjaStorms by implementing a virtual memory layer. The group has understood and applied multiple textbook operating system principles such as Page Tables, Page Fault interrupt handling and User- and Kernel Mode separation on a detailed level, and were able to fully implement a process abstraction and memory isolation through managing the virtual memory page tables.

Project 5 implemented a networking stack on NinjaStorms, including a driver for the e1000 networking hardware, as well as an entire functioning “Address Resolution Protocol” (ARP) stack, which was capable of completing address resolution handshakes with other devices on the network. This work required a detailed understanding of the functionality of a networking device, as well as RFC 826.

Both Project 1 and Project 5 needed to rely on extensions to the architecture of the NinjaStorms operating system, such as a clean separation of User- and Kernel mode, as well as a driver infrastructure. By working closely together with Project 6, which implemented these architectural changes in the kernel, all teams were able to complete their ambitious projects with impressive results within the time frame of the semester.

Project 9 also extended NinjaStorms with support for additional LEGO auxiliary periphery devices that were communicating with the original LEGO drivers through a serial interface. This functionality was previously unavailable in earlier version of NinjaStorms. The group has successfully reverse-engineered the behavior

[1] `https://github.com/ninjastorms/ninjastorms`.

of the periphery devices from the schematics and the upstream source code, and were able to produce working drivers for devices such as the LEGO Color Sensor.

Project 2 and Project 8 focused on the GNU/Linux operating system. Project 2 explored and evaluated one possible way of extending the Linux kernel with modules written in the programming language "Rust". Rust is expected to bring a number of benefits compared to traditional kernel source code, which is usually written in C. Rust promises more memory safety and easier error management, although not all of these features are available outside of the comfort of the process abstraction. The group has done extensive work to integrate Rust cleanly into the kernel and have reimplemented an existing kernel module as a proof of concept.

Project 8 extended the kernel packet filtering APIs to userspace in order to make deep packet inspection and filtering available to applications running in user mode. For this, the group have developed a custom module for the Linux kernel capable of modifying the stream of network packages belonging to the network connections of the applications, and extended an API for dropping, modifying or resending packets. Afterwards, a thorough and extensive analysis compared the approach with the capabilities of the "extended Berkeley Packet Filter" (eBPF).

Project 3 revived the "Multicomp" project by Grant Searle, creating a platform for running historical 8-Bit software on a Zilog Z80 "Soft-Core" implemented on a "Field-Programmable Gate Array" (FPGA). The group has successfully installed and booted CP/M on the hardware they configured, solving multiple challenges in the process. Additionally, they extended the original Multicomp project by implementing a more reliable means of deploying software on the CP/M instance.

Project 4 implemented an extension to the ELF executable binary format which makes it possible for an executable file to be run on different "Instruction Set Architectures" (ISA), either by partial recompilation from immediate code, or by relinking from object files. To embed both the object files and intermediate code in the binary, the group have developed the PEX format. As a proof-of-concept, the group has compiled the GNU stream editor sed into a PEX file capable of running both on x86_64 and aarch64, followed by an analysis of the caveats of this approach.

Project 7 applied state-of-the art static analysis tools to the Windows Server 2003 research kernel source code to try and confirm a previously published "Common Vulnerability and Exposure" (CVE) report. In their work, the group have evaluated the efficacy of static code analysis for vulnerability analysis and discuss the additional challenges presented by analyzing code running in kernel space.

Lastly, Project 10 approached quantum computing with the question how to integrate the new paradigm into existing software engineering techniques. Best practices such as reusability, maintainability and the integration of quantum algorithms are presented. To show applicability and limiations, a hybrid implementation of classical and quantum computing is discussed.

It should be recognized that over the course of the semester all of these projects have achieved outstanding results which went far beyond the scope and the expectations of the lecture, and we would like to thank all participating students for their commitment and their effort in completing their respective projects, as well as their work on compiling this report.

Contents

Implementing Memory Management in the NinjaStorms Operating System

Marcel Garus, Rohan Sawahn, and Jonas Wanke

Hasso Plattner Institute for Digital Engineering
marcel.garus@student.hpi.de
rohan.sawahn@student.hpi.de
jonas.wanke@student.hpi.de

Memory management is a central component of modern operating systems. Our goal was to create a reference implementation for the NinjaStorms operating system for educational purposes. Due to the time constraints of a single lecture in a single semester, we could only lay the foundation by enabling the two-level paging capabilities of the MMU. This includes the handling of page faults to assign physical memory on-demand and creating a memory layout for the OS. The paper also describes newcomers' struggles with low-level programming in C, e.g., creating structs suitable for the binary interface, handling errors, and debugging kernel faults. Finally, we examine limitations of our system, such as using fixed page types, we outline next steps, e.g. compiling processes separate from the kernel and maintaining a translation table per process, and we compare our work to real-world implementations by summarizing features that make Linux's memory management more complex.

1 Overview & Scope

This report will present the results of our project "NinjaStorms Memory Management"[1] on which we worked in the context of the lecture "Operating Systems II" in the summer term 2020 at the Hasso Plattner Institute (HPI) in Germany.

NinjaStorms is a real-time operating system (OS) for the Lego Mindstorms EV3 that was developed by the "Operating Systems and Middleware Group" at the HPI. The Lego Mindstorms EV3 is a programmable brick that is mainly used to create and program robots and other interactive systems. It is based on an AM1808 CPU and features 64 MB of memory.

The CPU comes with a 32-bit ARM9 coprocessor (ARM926EJ-S), which provides the memory management unit (MMU), with features such as a translation lookaside buffer (TLB), domain access control and paging. Currently, NinjaStorms provides basic OS functionalities such as scheduling of (basic) processes, a minimal C library, and some drivers for hardware such as sensors and buttons. Processes are

[1] Available at `https://github.com/hpi-bs2-st2020-ninjastorms-memory/ninjastorms`.

scheduled preemptively via timer interrupts and represent programs in execution. The source code of processes is hereby directly compiled together with the kernel.

In an operating system, executed programs need access to required parts of the memory. This is where memory management takes place: Memory management is the part of the operating system ensuring that processes have access to the memory areas they need, without the process having to worry about how to access these required areas. Memory management allows for a much larger address space by abstracting physical addresses using virtual addresses. Moreover, memory management is needed to reach a higher level of security in the operating system. With memory protection and process isolation, processes can only access selected parts of the memory and are thereby prevented from affecting other processes or – even more dangerous – the kernel itself. In addition, memory management provides the means for more efficient usage of the given hardware. RAM is fast, but expensive, whereas storage on a disk is cheap, but slower. By allowing currently unneeded pages to be swapped out to the disk instead of keeping them in memory constantly, the operating system can ensure more efficient use of given resources.

The aim of our project was to add memory management to the existing operating system. As outlined above, the system does not have fully isolated processes yet, so we will only focus on the implementation of two-level demand paging, requiring usage of the MMU. In the following, we will describe our development process and give insights on our design choices and difficulties we encountered. Furthermore, we will outline the limits of and possible extensions to our system.

2 Development Process: Problems & Design Decisions

After installing and getting comfortable with NinjaStorms, it quickly became apparent that trial and error approaches are insufficient for setting up low-level hardware because you need to get a lot of things right upfront. That's why, in the beginning, we spent a great deal of our time reading the Technical Reference Manual provided by ARM [1]. Our first goal was to activate the MMU without crashing the system. The MMU offers several page layouts to choose from:[2]

Root: Translation table. The table where each page lookup starts. It contains 4096 entries pointing to sections or coarse or fine page tables.

Level 1: Sections. These directly map 1 MiB of memory without the need of a second-level fetch.

Level 1: Coarse page tables. These contain 256 entries pointing to large or small pages.

Level 1: Fine page tables. These contain 1024 entries pointing to large, small, or tiny pages.

[2]See the manual [1], Figure 3.2 "Translating Page Tables".

Level 2: Large pages. These map 64 KiB of memory.

Level 2: Small pages. These map 4 KiB of memory.

Level 2: Tiny pages. These map 1 KiB of memory.

As our implementation is for education purposes, we explicitly decided against implementing all available page layouts and instead focus on a few that comprise a structure representative of real-world memory management systems. Most MMUs and OSs support multi-level paging, so we decided to go with two-level paging, opting for the medium size in each layer: Having a translation table that points to coarse page tables, which in turn point to small pages.

Our first step was to implement the structs for those three layouts. Naturally, we started by implementing two structs for each layer – one for the entry and one for the table itself that just contains an array of entries. So in total, we ended up with six structs.[3]

Actually though, that's one layer too much: The lowest layer – the small page – does not have rich structs as entries but instead is merely an array of pointers to physical addresses. What had happened is that we duplicated the entry definition of the two lowest layers. We fixed this problem by combining the two lowest layers back together so that the structs match the specification.[4]

This problem arose because we assumed a mental model of how the paging works too early. Also, some of us were inexperienced in reading technical hardware documentation, so we easily got lost in there.

Another problem was that we did not get bit packing to work. By default, GCC (the C compiler we used) will choose a memory layout that is performant (for example, by padding and aligning fields). However, we need our structs to exactly match the entry definitions in the reference manual – after all, it defines a binary interface so we must adhere to it bit-by-bit. We attempted to use bit packing, which is a technique that lets us more explicitly control the memory layout of structs by telling the compiler how many bits to allocate for each field and where to insert padding.

By now, we figured out we simply missed adding `__attribute__((packed))` annotations to the structs. In the meantime, we worked around this issue using raw `uint32_t`'s as types and writing accessors (getters and setters) as macros[5] as shown in Listing 1.

This problem occurred because we were relatively inexperienced with writing C code, let alone writing low-level C code. At least, we now know how to implement bit packing for future projects.

After our mental model of the required page tables matched the actual model described in the reference model and our implementation produced the correct data structure in memory, there were still some problems left to solve.

[3]`memory_small_pte_t`, `memory_small_pt_t`, `memory_coarse_pte_t`, `memory_coarse_pt_t`, `memory_translation_table_entry_t` and `memory_translation_table`. See commit `8ec7c09`.

[4]See commit `a35c255`.

[5]See commit `23022c9`.

Listing 1: Page table entries as raw `uint32`'s with macros as accessors

```
typedef uint32_t mem_lvl2_entry_t;
// ...
#define LVL2_ENTRY_GET_BASE_ADDRESS(entry) ENTRY_GET(entry, 12, 20)
#define LVL2_ENTRY_SET_BASE_ADDRESS(entry, value) \
    ENTRY_SET(entry, value, 12, 20)
```

As the translation page table needs to be aligned to a 16 KiB boundary[6] and the coarse page tables have to be aligned to a 1 KiB boundary, we initially hard-coded their memory addresses to `0x4000000` and `0x4001000`, respectively.

Because the translation table has a size of 4096 entries $\times$ 4 B/entry = 16 KiB, it occupies the space from `0x4000000` – `0x4003FFF`. This means that our tables overlap in the `0x4001000` – `0x4003FFF` range. We first solved this problem by adjusting the offset of our coarse page tables and later on had their offsets set by the linker directly (v.i.).

Another small mistake happened when we set the translation table base register (TTBR) of the MMU. Instead of writing the value of a pointer to our translation table, we temporarily assigned the pointer *to another pointer* to the translation table.

After these problems were solved, NinjaStorms worked correctly with paging enabled for the first time. However, our project was not done yet. Only enabling the address translation with an identity mapping has no benefits but only slows down memory access. Hence, the next step on our way to full memory management was demand paging, i.e., only adding entries to page tables when they are actually required by the currently executed program. This initially saves space for the page tables and also allows us to remove unused entries in the future.

When a program tries to access a memory location that is not yet mapped by the MMU, a data abort or prefetch abort exception is generated. These can be handled by registering the corresponding interrupt handler. To test this behavior, we initially removed a single coarse page table entry, meaning a 4 KiB block of memory is not mapped anymore. We then registered a data abort handler function that added the missing entry and then returned execution to the program, re-executing the exception-triggering instruction. Next, a simple set and get of a single byte inside this block was added, intending for our registered data abort handler to add the missing entry and then retry the access. However, our handler function wasn't executed and the CPU stuck with a Page Translation Fault.

The solution to this problem came in the form of an external contribution,[7] pointing us to the missing data abort stack that is required to execute a data/prefetch abort handler.

[6]See the reference manual [1], chapter 3.2.1.

[7]`https://github.com/hpi-bs2-st2020-ninjastorms-memory/ninjastorms/pull/1`.

With this working, we have now managed to implement basic demand paging. But to efficiently use the given memory it makes sense not to identity map the whole available memory. Instead, we want our kernel to initialize the page table with as little pages as possible and then map new pages later when they are required.

This quickly raises the question of how many pages actually are needed in order to set up the system. We only need to initialize the pages that are needed for setting up the memory management. This is needed in order to then handle incoming page faults and assign new pages on demand.

We first tried to just guess how much memory will be needed to set up the system and just assigned half of the possible virtual address space and then tried to approximate how little memory will be needed to keep the system working.

We realized that about $\frac{1}{8}$ of the virtual address space seems to be needed to keep the system working. $\frac{1}{8}$ of the virtual address space is 256 MiB which obviously cannot be accurate, as we only have 64 MB of physical memory.

So instead of trying to approximate the needed memory, we started looking for other ways to find the exact amount of memory needed to initialize the page table with the functions required for paging. We discovered that the linker script offers the possibility to find out how much memory the compiled functions need that are needed for memory management.

We defined a section that includes the compiled functions that are needed for memory management. By using variables that contain the address before and after this section we can find out the required size for these functions as shown in Listing 2.

Listing 2: Excerpt from the linker script packing everything required to initialize memory management into a common section and storing its start and end

```
.minimal_memory_management_part :
{
    . = ALIGN(4K);
    _minimal_memory_management_part_start = .;
    kernel/interrrupt.o;
    kernel/interrupt_handler.o;
    kernel/main.o;
    kernel/memory.o;
    kernel/syscall.o;
    kernel/syscall_handler.o;
    _minimal_memory_management_part_end = .;
}
```

Now we are able to access the variables `_minimal_memory_management_part_start` and `-end` from our memory management .c-file. The variables contain the following values:

- Start-Address: `0x25000`
- End-Address: `0x42921f`

Therefore, the space needed to compile these functions is about 0.5 MiB, which seems reasonable. Our next step then was to initialize 0.5 MiB of pages for the memory management functions.

Working with the linker script we realized that we should think of how the memory layout should be first. We needed to define where our page tables should be placed in memory and hence find out how the whole memory is structured. Therefore, we looked into existing projects such as the Linux kernel to find out how the memory layout is designed there. We then decided to stick to the most common design we could find, which is also implemented in our linker script as shown in Figure 1.

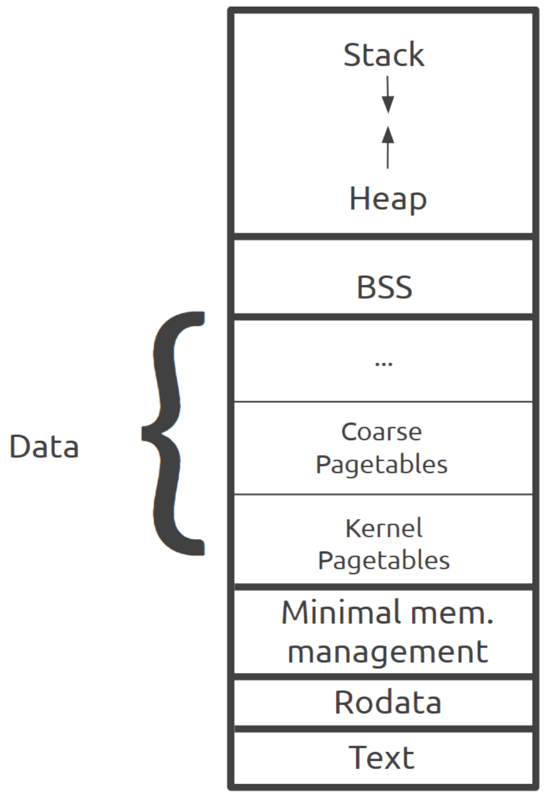


Figure 1: Memory layout

We decided to have a special section where the compiled code for a minimal memory management setup lies, which will be linked to the beginning and gives us information about how much memory is required to set up the minimal memory management. We then decided to place our page tables at the beginning of the data section. The `Text` section contains executable instructions. The `Rodata` section contains read-only variables and the `Data` section contains initialized variables. The `BSS` section contains uninitialized variables.

As soon as tasks are compiled by themselves, they will have their own memory layout, which isn't possible yet, as they are currently compiled together with the operating system. Hence, the memory layout we have designed is only used for the kernel yet.

Due to temporal constraints and the workload of the exam period we were unable to finish a dynamic solution that adapts to the actual size of functions needed for memory management. We spent a lot of time trying to find out why the system stopped working when we initialized less than $\frac{1}{8}$ of the virtual address space, but could not find the reason. Therefore, we have now submitted a system that statically maps $\frac{1}{8}$ of the virtual address space, which is more than enough to initialize the basic kernel functions needed for memory management. The next step here would be to properly initialize the page tables with as little memory as possible.

3 Analysis

Here we reflect on what we achieved in relation to our initial goals. We also outline further improvements that could be made in the future and list what distinguishes our system from real-world implementations.

3.1 Limitations

Our implementation of memory management for NinjaStorms was developed to the same basic requirements that NinjaStorms itself has: It should run on the Lego Mindstorms EV3, as well as on a QEMU emulator.[8] Both systems used the ARM926EJ-S coprocessor for memory management, and hence our implementation automatically uses a subset of its functionality. Supporting a different platform/MMU would likely involve a lot of refactoring.

Also, we only generate coarse page table descriptors for the first level and small page descriptors for the second level. However, as noted above, the MMU also supports other options.

As those descriptor types can be mixed and matched, we could dynamically decide which type is a better fit for the current allocation, e.g., by using a heuristic. When implementing `malloc` and a program requests 32 KiB of memory, we can directly map a large page instead of eight small pages. Also, we know that our kernel has some minimum size and can, e.g., use a section or a coarse page table combined with large pages, depending on that concrete size.

This decreases the size required for storing all page tables, and should also slightly increase the final performance as fewer entries have to be retrieved and

[8] `https://github.com/ninjastorms/ninjastorms#introduction`.

hence the TLB can store additional information. In the case of sections, the MMU also does not have to go through two layers of indirection, but only one.

3.2 Next Steps

When continuing with the goal of "fully-featured" memory management for NinjaStorms, an important step is strictly separating different processes by compiling them separately (so their memory areas do not share any pages) and storing page tables per process rather than one global table. Context switches then also have to invalidate the TLB cache and change the TTBR value to match the translation table of the new process.

For increased security, we should also mark different memory areas as readable/writable/executable. This allows code pages to be marked as executable but not writable, guarding against remote code execution or just accidental modifications.

Another handy feature with regard to the limited memory size of a Lego Mindstorms EV3 (60 MiB) is swapping, i.e., moving older memory pages to a persistent storage medium (e.g., an SD card) to make space for newer data. When access to the old information is required, it has to be read back from storage. This can also be combined with memory-mapped files: Offering an API for reading from / writing to files in the same way you would access RAM while the OS takes care of buffering and pre-loading files from the underlying storage.

Another feature that may benefit multi-process systems is shared memory: One process can allow other processes to read/write to parts of its memory for quicker communication (without the overhead of pipes or sockets, for example). Also, copy-on-write pages could be used when different pages (coincidentally) contain the same contents, as is common when implementing Linux's `fork` syscall. Instead of storing both copies, only one page has to be stored and marked as copy-on-write, and both usages refer to that shared page. As soon as one tries to write to the page, a copy is created and modified, saving storage in the meantime.

3.3 Comparison to existing works

Our implementation is pretty close to the typical textbook implementation of memory management. If one would look at the slides of the OS I lectures and attempt to create a memory management system, the system would be pretty close to what we have.

By contrasting our implementation with Linux memory management, it becomes even more apparent that our work is an education project and that real-world implementations quickly become much more complex. Here are just some of the features of Linux's memory management that are out of scope for this project:

Hardware diversity. Linux needs to run on a variety of hardware configurations – even ones without an MMU, in which case the "nommu" memory management is applied, which works totally different. [2]

Non-Uniform Memory Access (NUMA). Linux supports architectures with multiple processors and has optimizations for NUMA. Each processor is physically closer to a different range of memory and different caches, so the memory management maintains separate page lists for each processor. [2]

Zones. Some ranges of memory might restrict access patterns. For example, you might have a chunk of memory that hardware can directly write to or other parts that are only available temporarily. [2]

Compaction. Some drivers actually require contiguous chunks of memory, for example for direct memory access. Enabling allocation and deallocation of contiguous chunks re-introduces the fragmentation issue solved by pages. That's why Linux also needs a compaction mechanism. [2]

Kernel Samepage Memory (KSM). If multiple pages have the same data in them (for example, by starting a program multiple times and thereby loading it into memory multiple times), Linux recognizes this and de-duplicates the data. [3]

Memory Hotplug. Sometimes it makes sense to increase/decrease the amount of memory during runtime – for example, if you want to improve your system by physically plugging in another NUMA node or if your OS is running in a VM / cloud instance and the load changes. [4]

4 Conclusion & Outlook

During the project, we were able to learn a lot about several topics, such as interrupt handling, advantages and disadvantages of one- and two-level paging, and much more. When we first started the project we only had mixed (mostly little to no) experience with reading technical hardware documentation, programming in C, and developing low-level software that interacts directly with the hardware. Due to this fact we needed some time in the beginning to properly understand the existing NinjaStorms OS and how the CPU works. With this knowledge, it will now be a lot easier to dive into existing operating systems in future projects. We improved our knowledge of the C language and GNU standards enormously and also learned a lot about debugging hardware-related systems with tools such as gdb.

In general, we deeply improved our knowledge about operating systems and how theoretical OS concepts can be implemented for working systems. This was especially interesting because in last year's lecture "Operating Systems I", we learned a lot about the underlying concepts of how operating systems work, but did not really get in touch with the implementation of these concepts in real-world operating systems.

At the beginning of the project, our goal was to create a complete memory management for the Mindstorms EV3 hardware, but while getting more familiar with the system, we quickly realized that this might be a bit out of our reach. Due to the temporal constraints and our difficulties with getting the MMU enabled (such as debugging a kernel), we decided to refine our goal to provide an implementation of two-level demand paging for the EV3 hardware. Looking back, we were able to implement this and have created the fundamental basis for an implementation of memory management for the NinjaStorms OS. We also had to realize that unexpected errors occurred frequently, so we had to spend a lot of time finding their origins and fixing them. Of course, our system is not feature complete, but this wasn't the goal of the project. On the basis of our work and the work of the other groups improving NinjaStorms, it now should be possible to implement memory management that dynamically distributes memory across all processes. Possible future features could be isolated processes, proper access control, and swapping, which would make the system more efficient and secure.

References

[1] *ARM926EJ-S Technical Reference Manual*. `https://developer.arm.com/documentation/ddi0198/e/`. Online; accessed 24-November-2020; referring to revision E.

[2] *The Linux kernel user's and administrator's guide: Memory Management: Concepts overview*. `https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html`. Online; accessed 24-November-2020.

[3] *The Linux kernel user's and administrator's guide: Memory Management: Kernel Samepage Mapping*. `https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html`. Online; accessed 24-November-2020.

[4] *The Linux kernel user's and administrator's guide: Memory Management: Memory Hotplug*. `https://www.kernel.org/doc/html/latest/admin-guide/mm/memory-hotplug.html`. Online; accessed 24-November-2020.

Creating Linux kernel modules in Rust

Rewriting a file system in a safe language

Clemens Tiedt and Clara Granzow

Hasso Plattner Institute for Digital Engineering
`clemens.tiedt@student.hpi.de`
`clara.granzow@student.hpi.de`

The `rsramfs` project explored the viability of Rust for the development of Linux kernel modules. We found that while the tooling and ecosystem for kernel development are not on par with C yet, Rust has the potential to make it easier to reason about code and increase trust in it.

1 Introduction

One of the biggest problems in kernel development are memory safety issues. For example, since 2006, approximately 70% of all security vulnerabilities Microsoft fixed are related to memory safety. [1] Additionally, C as a programming language does not prevent these kinds of vulnerabilities. Over 30% of security vulnerabilities in the most important open-source C projects are because of buffer errors, making this the most common type by a large margin. [13]

A promising approach to circumvent this problem is the use of a memory-safe language like Rust. Because of safe and unsafe blocks, possibly unsafe code is contained to a smaller block and thus easier to review. Run-time garbage collection is replaced by static checks. This means that Rust does not have to trade memory-safety for speed. [2]

Rust is currently used at a kernel level only in a very limited fashion. There is an operating system, RedoxOS, written entirely in Rust. [10] For Linux, there are currently no modules written in Rust with an actual use-case, only some that serve as a proof of concept. [4, 5, 6, 7, 12]

2 Contribution

Since there are currently very few Rust kernel module projects, we want to contribute our own project. Where other modules were mostly proof of concepts, we wanted to create one with actual functionality. In particular, our goal is to take an already existing module and translate it into Rust. This gives us the possibility to write a kernel module that is a little larger while still being able to do this in the span of one semester. The module we chose is the file system ramfs.

3 Theory

Before we discuss our architecture and implementation, we provide a short overview of the theoretical background of our work.

3.1 File Systems in Linux and ramfs

When we started the project, we only knew that we wanted to write a kernel module in Rust. But we were still unsure what type of kernel module would be the best. Since we agreed that it needed to be relatively self-contained, the choices were somewhat limited. After discarding the idea of a device driver due to the difficulty of interacting with hardware, we settled on a file system, specifically ext2. This gave us the advantage of an existing implementation in C that we could base our work on. Additionally, the code for ext2 is relatively short compared to other file systems.

However, it still proved to be too much for one semester of work. This became apparent the moment we started to actually try to implement it. Because of this realization, we decided to go with a simpler filesystem, ramfs, instead.

3.1.1 Introduction to ramfs and the VFS

Ramfs is a very simple Linux file system and can be seen as a barebones virtual file system (VFS) module historically used to implement the `/tmp` directory. The code is relatively short, because most of the work can be done using the existing VFS caches and does not require any hardware interactions.

The Linux kernel provides an abstraction layer above the actual file system – the VFS. This allows the Linux user to choose between different file systems and gives a basic structure that a file system implementation may follow. As such, there are some important objects that an implementation of a file system like ramfs uses. Firstly, there is the superblock. This is a struct in which the metadata about the mounted file system is stored. The superblock contains the general information about the file system like its type, mount flags and a reference to the superblock operations. These can, but do not have to be, customized by the file system. They include things like `show_options`, which displays the mount options specific to the file system. Another important object is the index node or inode. This is also a struct and contains all the information about a specific file, but not its actual data. A reference to a list of all inodes can be found in the superblock. An inode contains for example the file's access rights, its size or the time it was last accessed. There are also customizable inode operations like `mknod`, which creates a new file by allocating an inode. The file system treats a directory like a file that knows a list of files and other directories. When a directory or file is used, a dentry (directory entry) object is created. There exists one dentry object for each part of a path name. A dentry is associated with an inode. There is a dentry cache through which directory access becomes faster. [3] Ramfs is a file system that is used, as the name implies, to save files on RAM. This works by using the existing caching infrastructure. However, the data is never put in storage. Because no access to the hard drive is required, ramfs is fast. But this also means that its data is temporary. A similar filesystem to ramfs that can be seen as its successor is tmpfs. As such, there is the option to limit tmpfs's filesystem size and memory use. Additionally, tmpfs allows for swapping out pages. Since none of this is possible with ramfs, ramfs poses the danger of running out of memory. [9]

3.2 A short primer on the Rust language

To be able to properly discuss Rust code and concepts, we now provide a short introduction to some Rust features relevant to our project. An important concept is that of attributes. Rust uses attributes (which are written as `#[attr_name]`) for various purposes. They can be global (in which case they use a shebang instead of a hash symbol) to affect the entire crate (i.e. a Rust package) or attached to pieces of code like a struct or function. They are comparable to annotations or decorators in other languages. Rust offers a form of object-oriented programming using interfaces, here referred to as *Traits*. These make Rust's type system flexible and allow for polymorphism as they allow for restrictions on generic parameters. Some simple trait implementations such as `Copy` and `Clone` which handle moving data implicitly and explicitly can also be automatically generated using the `#[derive(Trait)]` attribute. Probably the most important feature to make writing kernel modules in Rust feasible is its foreign function interface (FFI). Using the FFI, functions from C libraries can be made available in Rust and Rust functions can be exposed with a C ABI. It should be noted here that C declarations allow some features not available in other places in Rust such as variadic functions. Rust also supports raw pointers which give up safety guarantees and behave like C pointers. When more complex data types such as structs are used, they need to be replicated in Rust and marked with the `#[repr(C)]` attribute which ensures that they are laid out in memory the same way they would be in C. Unsurprisingly, Rust-specific types such as enum variants with fields are not FFI-safe.

3.2.1 Memory safety in Rust

An important concept in Rust is safety. Generally, Rust can guarantee that any code that is not explicitly unsafe will behave correctly in terms of memory usage, i.e. not produce undefined behavior (such as memory leaks or double frees). Rust furthermore comes with some features that complement these safety guarantees. Firstly, there are only two ways for a Rust value to contain a null pointer. Either it was explicitly constructed with the `core::ptr::null()` function or it was passed in through an external function via FFI. In fact, only raw pointers can be null. Rust differentiates between references and raw pointers. The compiler statically checks that references are valid and do not outlive the value they reference, whereas raw pointers make no such guarantees. Secondly, all variables are immutable by default. This makes it easy to see where values are changed as the relevant variables must be explicitly mutable. In general, only four kinds of operations are considered unsafe by Rust. These are dereferencing raw pointers, calling unsafe functions (all functions called via FFI are considered unsafe), accessing static mutable variables, and implementing unsafe traits. Unsafe operations may be performed within an `unsafe {}` block. However, seemingly paradoxically, unsafe blocks are considered safe. The reason for this is that unsafe code is not strictly wrong, the compiler just cannot guarantee its safety – for example not every raw pointer is a null pointer. So, by writing an unsafe block, the programmer steps in to guarantee that the specific instance of a generally unsafe operation is safe in the context. This is important because (to borrow a mathematical term) safe code in Rust forms a closure, i.e. any combination of safe operations will be safe. With unsafe blocks meaning that the programmer guarantees the safety instead of the compiler, we can write safe code that builds on unsafe code.

3.3 Running Rust code in kernel space

Getting code in languages other than C to run in the Linux kernel is generally not a trivial task. Firstly, not all languages are fit for this task. For example, the Go language is often named as a competitor to Rust. However, Go requires a runtime for memory management (e.g. garbage collection) making it extremely impractical for use in kernel mode. Even with languages that are better suited to kernel development usually two central issues come up. The first one is that the standard libraries of many languages (including Rust) link against libc which is strictly user mode only. The second is the lack of bindings to kernel interfaces. Thankfully, the first issue is comparatively simple to solve in Rust. Rust has a `#[no_std]` attribute at the crate level to disable linking against the Rust standard library. Without the standard library the core library, the dependency-free basis of the standard library is still usable. This way, many of Rust's language features and types are still available. When writing a crate without the standard library one needs to provide implementations for a few language features, e.g. the behavior when panicking. Most of these can be implemented just as stubs. With these steps we could already create a library that could be linked into a kernel module. However, seeing as the core library does not provide any memory management, this library was not very useful yet. Specifically, we could not use any heap-allocated data types such as `Box<T>` or most importantly collection types such as Rust's list type `Vec<T>`. Fortunately, using the bundled crate *alloc* it is possible to define custom memory allocation which we use to allocate and deallocate heap memory with the `krealloc` and `kfree` functions respectively. We will discuss how we got access to these functions later, right now we just assume these were imported using an `extern "C"` block. Finally, we wanted the ability to print debug output since Rust's `print!` macro relies on libc. For this we used the implementation provided by souvik1997's example module *kernel-roulette* which uses `kprintf` to redefine Rust's `print` macro.

3.3.1 Build process and access to kernel interfaces

At this point, we had a Rust library that could theoretically be linked into a kernel module. Linking proved surprisingly simple. Linux kernel modules use `kbuild` which itself uses makefiles. So, we could just write a kernel module using C that links against our Rust library in the kbuild process and provided an entry and exit point which would respectively hand over control to our Rust code. The Rust compiler can compile code for different platforms using target specifications. Our kernel module target requires a number of special settings such as aborting instead to attempting to unwind the stack in the event of a panic. The target specification from *kernel-roulette* gave us the ability to compile our module for an x86-64 architecture. However, without kernel bindings we could not hope to build any functionality. A naïve approach would be to write the necessary bindings ourselves as `extern "C"` function declarations and `#[repr(C)]` structs, but this way we could not have finished the project in one semester. Another idea would have been to use opaque representations which would have reduced the number of interfaces we needed to copy, but it would also have made checking any safety guarantees a lot harder. What we decided to do instead was to use the tool *bindgen* which can create Rust bindings for a C library. Since this process requires linking against kernel libraries, it is not trivial. Thankfully, *fishinabarrel* already solved this issue for their kernel module. Their solution extracts the required compilation flags from a mock module to then run bindgen using these flags. This causes a few issues mainly caused by Linux using *gcc* whereas bindgen relies on LLVM/clang instead. This restricted us to only being able to create bindings on Linux 4.x due to incompatibilities on kernel version 5.x. We also had to manually edit the generated bindings as bindgen failed to derive the `Copy` trait on some structs.

4 Porting C to unsafe Rust

By now, we had the C source code of ramfs and an empty Rust library capable of being linked into a kernel module. The next step for us was to port this C code to Rust. We did this in two steps. In the first one we rewrote the C code almost as a verbatim translation in Rust. Due to Rust's FFI, we could rewrite our code piece by piece and test that each translated function still behaved the same way. This is the main reason we chose to start with an unsafe implementation. We could have started with a more complex, but safe implementation, but then differentiating between errors caused by behavioral differences between C and Rust and actual defects caused by our implementation would have been significantly more difficult. As many functions from the original ramfs implementation simply call other functions with specific arguments or flags, we started with those. In almost all cases, porting the functions was a matter of translating C types and control structures to the equivalent Rust ones. There were some exceptions, notably `switch` in C behaves differently from Rust's `match` with `match` only matching against literals or patterns where `switch` allows for variables. Another issue that occurred was that on all systems except for Windows Subsystem for Linux 2 assigning a struct field that should contain a pointer to a `const` global instance of another struct caused a panic. While we were unable to determine the root cause, we were able to create a workaround by passing the struct over to the C side of our module and doing the assignment there. At this point we also did not port all functions. Notably, we initially left out the parsing of mount options as we believed it would be easier to completely rewrite the relevant function using idiomatic Rust from the beginning. We also decided to only support systems with a memory management unit whereas the original ramfs implementation provides code for systems without one. Since we were already limited to x86-64 we considered this a sensible limitation.

4.1 Rewriting in safe Rust

Unfortunately, C code that has only been blindly translated is not safe Rust code. So far, our code still contains things like dereferencing of raw pointers and directly calling C functions. But since Rust's properties of memory safety are one of the main reasons we want to use this language in the first place, we need to fix this – the unsafe code must become safe.

But before going into the details of how we did this, it is helpful to understand the overall structure our project had after completing this process. In the end, we had seven files of actual code; six Rust files and one C file. They can be arranged into two different groupings of three and two files, leaving two files separate. The

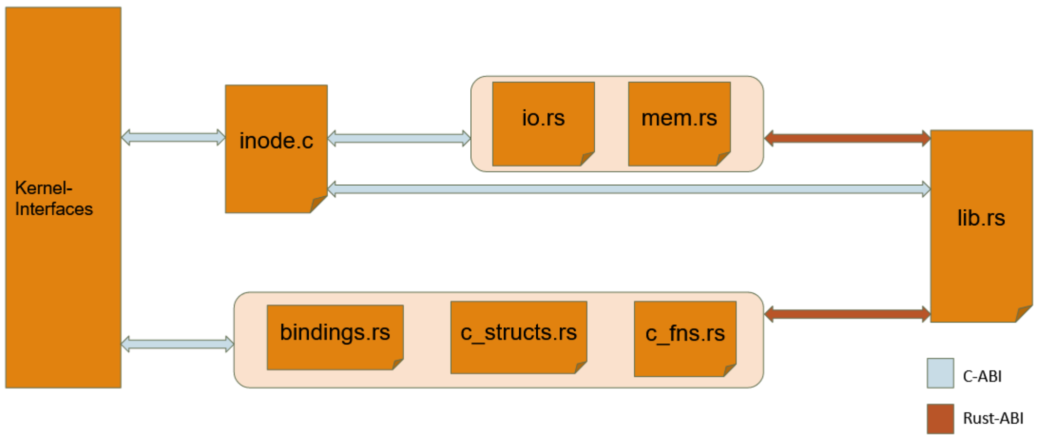


Figure 1: The architecture of rsramfs

first of those separate files is inode.c. This forms the starting point of the module. In essence, this file is just a copy of the inode.c-file from the original C-version of the file system, but with everything taken out that we moved elsewhere. What remains is what is needed to initialize the module. Additionally, the file contains some C-code that would be hard to call via FFI, like inline functions and macros.

Most of the actual functionality of the file system is contained in lib.rs, the second of the stand-alone files and the root of our Rust library. The implementations for the file system's functions can be found here. To be more specific, the file contains the implementations for `ramfs_get_inode`, `ramfs_fill_super` and `ramfs_mknod`, among others. Those functions can be called via the C-ABI, as if they were C-functions, by the operating system. Aside from those functions, there are a few helper functions, mainly to provide Rust wrappers.

But that is not the only place where we can find Rust wrappers. The first grouping of several files, consisting of bindings.rs, c_structs.rs and c_fns.rs, can be summarized as providing Rust wrappers for kernel interfaces.

Bindings.rs is a file that is automatically generated by the tool bindgen, which generates Rust FFI bindings to C libraries. [11] This allows us the use of the C libraries used in the original implementation of ramfs without having to actually write the bindings by hand.

In c_fns.rs, Rust wrappers are put around the functions from bindings.rs for which this was necessary.

C_structs.rs contain the wrappers around certain C-structs, like inode. The process for this was slightly challenging and will be explained in more detail in a later paragraph.

The second file grouping contains mem.rs and io.rs, which, as is obvious from the name, are responsible for memory allocation and input-output. If we want to do things like printing or allocating pointers on the heap (which is required by certain data types) from our Rust functions, we need to write this functionality ourselves based on the kernel versions of these mechanics. Fortunately, we could take this from previous Rust kernel modules. [4]

Now we can finally tackle the actual question – how do we make unsafe code safe?

It is not possible for us to get rid of every unsafe line of code. We have to do those unsafe operations *somewhere*. But what we can do is hide them.

Dealing with a function call of a C function is relatively straightforward. All we need to do is put a wrapper function around the unsafe foreign function call – provided we are sure the code we want to wrap is actually safe. Since Linux kernel code is written with a high code quality in mind, we do not have to worry about this.

```
pub fn rs_kill_litter_super(sb: SuperBlock) {
    unsafe { kill_litter_super(sb.get_ptr()) };
}
```

There are some naming conventions we kept to: Our wrapper functions are called 'rs_function_name', with 'function_name' being the original name of the C function. There are some places where it was beneficial to change the return type to one of the Rust-specific types Option and Result, which facilitate error handling. We use an Option when the original return type contains a pointer: It returns `Some(element)` if element is not null and `None` if it was not. This prevents us from creating wrappers containing null pointers.

```
pub fn from_ptr(sb: *mut super_block) -> Option<Self> {
    if sb == core::ptr::null_mut() {
        None
    } else {
        Some(Self { ptr: sb })
        }
    }
```

If the C-function may return an integer that represents an error code, we use a Result. Either `Ok(())`, which can be seen as equivalent to returning `0`, or `Err(error)` is returned.

```
    fn rs_ramfs_mknod(
        dir: Inode,
        dentry: *mut dentry,
        mode: umode_t,
        dev: dev_t,
    ) -> Result<(), cty::c_int> {
        use bindings::ENOSPC;
        use c_fns::{rs_d_instantiate, rs_dget};

        match rs_ramfs_get_inode(dir.get_sb(), dir, mode, dev) {
            Some(inode) => {
                rs_d_instantiate(dentry, inode);
                rs_dget(dentry);
                dir.set_mctime_current();
                Ok(())
            }
            None => Err(-(ENOSPC as i32)),
        }
    }
```

Thus, we have created a safe interface that we can now interact with like we would with any other safe Rust code. If we want to use any of these functions, we will not have to worry about safety anymore.

This whole process is simple enough that we attempted to automate it. For that purpose, we started to write a tool, wrapgen, [14] that would do this. It is still rudimentary and has a lot of issues. But because we only had the idea for wrapgen after we already finished creating most of the wrappers by hand, and because we prioritized our main project, we put its development aside for now.

There are, of course, a lot of places where we have to deal with more than just function call but need to use a struct from one of the C libraries instead. We mainly needed to access various fields from inode and superblock. Dealing with structs proved to be a little trickier than dealing with functions. The problem were the raw pointers, since accessing them is almost always unsafe.

Our first idea to solve this problem, defining a trait that would dereference a `*mut c_type` to a `c_type`, was unfortunately impossible. This solution would have violated Rust's ownership rules and we would have needed pointers for the C-interface anyway.

What we did instead was to create lightweight wrapper structs. The only field they have is for the pointer we want to wrap. The pointer is of the type of the struct we want to wrap, for example inode.

```
#[derive(Copy, Clone)]
pub struct Inode {
    ptr: *mut inode,
}
```

If we want to access this struct, we have to do it via an associated function, like `get_ptr` and `from_ptr`, which both Inode and SuperBlock have. In addition, Inode has two special constructors, `new` to create a new inode using the `new_inode` function and `null` to create one containing a null pointer. We need some associated functions to access certain fields of the struct we point to, like `set_ino`, and functions that have the struct as pointer as an argument need to be added as associated functions as well.

```
impl Inode {
    pub fn new(sb: SuperBlock) -> Option<Self> {
        Self::from_ptr(rs_new_inode(sb))
    }

    pub fn null() -> Self {
        Self {
            ptr: core::ptr::null_mut(),
        }
    }

    pub fn from_ptr(inode: *mut inode) -> Option<Self> {
        if inode == core::ptr::null_mut() {
            None
        } else {
            Some(Self { ptr: inode })
        }
    }

    pub fn get_ptr(self) -> *mut inode {
        self.ptr
    }


    pub fn set_ino(&self) {
        unsafe { (*self.ptr).i_ino = get_next_ino().into() }
    }
    .
    .
    .
}
```

Finally, there were a few things we added to the structs ourselves, like the RamfsSuperBlockOpts, that gives us the possibility to add a custom debug mode. They were added as an extension trait.

```
pub trait RamfsSuperBlockOpts {
    fn is_in_debug_mode(&self) -> bool;
}
```

```
impl RamfsSuperBlockOpts for SuperBlock {
    fn is_in_debug_mode(&self) -> bool {
        unsafe {
            (*((*self.ptr).s_fs_info as *mut RamfsFsInfo)).mount_opts.debug
        }
    }
}
```

After we finished the rewrite to safe Rust, our project ended.

5 Possible future work

We consider rsramfs only a short exploration project to show what the state of Rust for Linux kernel module development is right now. As such, we see many possibilities for future work to expand on this topic.

5.1 Future work on rsramfs

At this point, rsramfs is feature-complete. There are however some remaining issues we would like to solve. We needed to split off a version that only ran in the Windows Subsystem for Linux 2 kernel because due to what we can only assume is a memory management bug assigning a reference to a global struct created in the Rust code causes a kernel panic. Other than that, we would like to separate the kernel bindings from our filesystem code. For a small project such as this one, it made sense to keep these in one crate, but the bindings and wrappers could be useful to other projects as well. By splitting them off into their own crate, they could be used by different projects and updated or expanded independently from our project. Beyond that, there are only some code style changes that might be useful. For example, the wrappers could be unified into a `Wrapper<T>` type that comes with functions to create one from a pointer of type `*mut T` and accessing that pointer when interacting with C code. We did not make this change in our project because the little code duplication was not an issue for us and other issues took priority.

5.2 Opportunities for improved tooling

Our project focused mainly on making C interfaces easily accessible through Rust code. Future work could also consider making it easier to export Rust functions into C code. The ones we expose take raw pointers as arguments and need to construct wrappers from them manually. It would be easier for programmers to be able to write their function using Rust semantics and automatically generate an `extern "C"` function that takes raw pointers for each wrapped argument of the Rust function and converts them. This would have taken too much time during our project, but it is achievable using Rust's procedural macros. In fact, we built a prototype that given a function that takes some arguments of type `Wrapper<T>` can create a function that instead takes arguments of type `*mut T` and generates the required wrappers. However, we believe that this tool requires more development before it could actually be used productively.

5.2.1 Performance considerations

Another concern here is performance. In general, we noticed no performance differences between the original ramfs implementation and rsramfs. However, our wrappers are not completely free of performance overhead. Almost every action is wrapped in a function call and while this may not be an issue with smaller tasks (such as writing or reading short text files), the overhead might become noticeable in larger or more complex tasks. There is some performance overhead we cannot get rid of, e.g. conditionals for null checking. We believe that by inlining function wrappers we could get rid of the bulk of the overhead. With this overhead gone, we believe that a larger kernel module using wrappers such as ours could actually be faster than a C one as e.g. null checks would only have to happen at the creation of a value instead of every time it is accessed.

5.3 Future work on other kernel modules

We see great potential in using Rust to create or rewrite more kernel drivers. Given the nature of ramfs our project had virtually no direct interactions with physical devices. Therefore we are very interested in seeing which challenges and opportunities arise when using Rust to create a proper device driver. Firstly, it could be interesting to build upon the abstractions we already wrote to create or recreate a file system that operates on an actual disk. Another project could be to write a driver for some sort of peripheral (e.g. PCIe) device. This should provide some insight into how well interacting with devices at a low level, e.g. having to use precise timings in communication protocols, works in Rust. Given that Rust is also used in embedded systems we have high hopes in this regard.

5.3.1 Testing Rust kernel modules

Another aspect our project did not explore is testing. Generally, testing kernel modules is not trivial and we resorted to a kind of simple integration test. However, kernel testing frameworks such as *KUnit* exist and it should be possible to create Rust wrappers around them. Rust's built-in testing support also works in a `#[no_std]` environment and we are eager to see how further work in this area can help make the functionality of Rust kernel modules more verifiable.

5.4 Possible official Rust support in the Linux kernel

Rust is still a relatively new programming language. Still, when we started the project, there were several other projects on GitHub for a Rust module for the Linux kernel. [4, 5, 6, 12] None of them are very big and all of them definitely experimental, but it is clear that there is some interest in the topic. Of particular note is a project that is attempting to create a framework for writing a Linux kernel module in Rust. Its authors are actively pushing the topic of using Rust in the Linux kernel into more official spaces: As a step in this direction, they presented their project on the Linux Security Summit North America 2019. [7]

But despite these valuable efforts, every developer that wants to delve into projects like this at the current moment in time will inevitably run into the problems that come with the lack of support.

This summer, however, offered a spark of hope that such a thing is not that far off into the future. Linus Torvalds himself stated in an interview that he could see Rust being used in the Linux kernel in the future – the kernel team was already looking into having interfaces to write things like drivers in Rust. On an exchange on the Linux kernel mailing list, he sounded a little more cautious but still ultimately receptive to the idea. On the Rust side of things, Josh Triplett, Rust language team leader, seems open for collaborations and enhancing the Rust language itself in order to facilitate kernel integration. [8]

This culminated in a discussion about in-tree Rust support at this year's Linux Plumber's Conference at the LLVM microconference. All in all, there seems to be a lot of enthusiasm for the idea, but there are still many problems that need to be solved. For example, when creating bindings with bindgen, macros and inline functions are currently not supported very well – a problem that we encountered in our project as well. Related to this is the issue of needing to write a large amount of wrappers by hand, which we also noticed during our project. On a more abstract note, since the only currently mature Rust compiler, rustc, uses a LLVM backend, there are support questions with kernel architectures that are not supported by LLVM, as well as possible ABI compatibility issues with a kernel built with gcc. [2]

All in all, it does not seem like there is going to be any actual official support anytime soon. However, we are confident that Rust support in the Linux kernel (be it official or unofficial) will improve in the coming years.

5.5 Conclusion

Writing a Linux kernel module in Rust is, at the current moment in time, viable but challenging. In our project, we successfully created such a module by translating the existing module ramfs from C to Rust. This proved that while there is no official Rust support in the Linux kernel so far, Rust is already usable in this context and may become a viable option for productive use in the future.

There are, however, some unresolved problems, particularly the need to split off a version for Windows Subsystem for Linux 2 kernel, whose root cause remains uncertain.

There were also some issues relating to the lack of support of Rust in the Linux kernel. There exist no official bindings to kernel interfaces, and while bindgen is a very useful tool to circumvent this, it cannot help with inline functions and macros, and the resulting kernel interfaces need to be manually wrapped. This is what we spent most of our work time on, but could (and probably should) be at least partially automated to ensure that future projects can focus on the actual work to be done. The further development of the tool wrapgen we started for this purpose is a possible avenue to achieve this.

References

[1] *A proactive approach to more secure code.* https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. Online; accessed 29-September-2020.

[2] *Barriers to in-tree Rust.* https://www.youtube.com/watch?v=FFjV9f_Ub9o. Online; accessed 29-September-2020.

[3] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* Sebastopol: O'Reilly, 2006.

[4] *kernel roulette.* https://github.com/souvik1997/kernel-roulette. Online; accessed 29-September-2020.

[5] *kmod.* https://github.com/saschagrunert/kmod. Online; accessed 29-September-2020.

[6] *linux kernel module rust.* https://github.com/lizhuohua/linux-kernel-module-rust. Online; accessed 29-September-2020.

[7] *linux kernel module rust.* https://github.com/fishinabarrel/linux-kernel-module-rust. Online; accessed 29-September-2020.

[8] *Programming languages: Now Rust project looks for a way into the Linux kernel.* https://www.zdnet.com/article/programming-languages-now-rust-project-looks-for-a-way-into-the-linux-kernel/. Online; accessed 29-September-2020.

[9] *Ramfs, rootfs and initramfs.* https://www.kernel.org/doc/html/latest/filesystems/ramfs-rootfs-initramfs.html. Online; accessed 29-September-2020.

[10] *RedoxOS.* https://gitlab.redox-os.org/redox-os. Online; accessed 29-September-2020.

[11] *rust bindgen.* https://github.com/rust-lang/rust-bindgen. Online; accessed 29-September-2020.

[12] *rust.ko.* https://github.com/tsgates/rust.ko. Online; accessed 29-September-2020.

[13] *What are the most secure programming languages.* https://resources.whitesourcesoftware.com/research-reports/what-are-the-most-secure-programming-languages. Online; accessed 29-September-2020.

[14] *wrapgen.* https://github.com/ctiedt/wrapgen. Online; accessed 30-September-2020.

Discovering Grant Searle's Multicomp
Implement the Zilog Z80-Processor on an FPGA Board

Tim Kuffner and Jannis Rosenbaum

Hasso Plattner Institute for Digital Engineering
tim.kuffner@student.hpi.de
jannis.rosenbaum@student.hpi.de

Anyone who today still wants to physically plugin and start a Zilog Z80 processor from 1976 must overcome a many hurdles. However, emulators do not offer a satisfactory solution either: they often simplify, leave out historical hardware restrictions, and make it difficult to connect peripherals. Therefore, a good middle course is to map the processor "gate exact" on an FPGA. Any peripherals can then be connected directly to the FPGA or implemented directly on the FPGA. The authors have investigated this approach using the *Multicomp* project of Grant Searle [18] and were also able to solve problems with memory connection, CP/M compilation, and program transfer.

1 Introduction

If one wants to run historical 8-bit software today, the main problem is to meet the hardware requirements. There are two solutions to this problem: On the one hand, the reconstruction of a computer from historical components or, on the other hand, the use of virtualization/emulators.

Obtaining and installing physical components is time-consuming and costly. However, emulators are not an optimal solution either, as they often abstract hardware details that have been exploited by some programs or simply do not provide the same connectivity as a physical hardware implementation. In the following, this paper examines a middle course: The implementation of historical processors – taking the Zilog Z80 as an example – on *Field Programmable Gate Arrays (FPGA)*. In doing so, the *Multicomp* project of Grant Searle [18] is elaborated and enriched by specific problem solutions and context. Multicomp enables the easy creation of computer systems with a choice of hardware, including the Z80, various ROMs and several user interfaces.

2 Technologies behind Multicomp

To understand the Multicomp project, one must understand what *Field Programmable Gate Arrays (FPGAs)* are. They consist of a set (an *array*) of programmable logic blocks (the *gates*) and reconfigurable connections so that these blocks can be wired together. From this grouping of logic blocks, almost any logic circuit can be designed and reconfigured "in the field" (in a few seconds). FPGAs, therefore, have a considerable advantage over classical *ASIC*s, application-specific integrated circuits, which have to be produced anew at great expense for each change. [23] FPGAs can be programmed using the hardware description language *Very High-Speed Integrated Circuit Hardware Description Language (VHDL)*.

Now, *Multicomp* builds on this technology and offers a *VHDL* template into which the desired components can be copied as *VHDL* source code. Multicomp supports the processors *Motorola 6809*, *MOS Technology 6502* and *Zilog Z80*. Communication with these processors can either take place via up to two serial interfaces or via a keyboard connected to the FPGA and an analogue, *VGA* or *SCART* video output.

This paper focuses on the *Z80*, a microprocessor from the Zilog company from 1976, which was binary compatible with the *Intel 8080*. This meant that most of the programs written for the successful *8080* could also be executed on the *Z80*. Furthermore, the *Z80* was even more straightforward to integrate, faster and cheaper. It is still produced in different forms, and over the years it was found again and again as (auxiliary) processors in numerous technical devices, such as the *SEGA Megadrive*, *Commodore C128* and several *Texas Instruments* pocket calculators. [12]

However, probably the most important application of the Z80 was of course as the main processor (*CPU*) in home computers. One of the earliest operating systems *CP/M* also contributed to this. It was first introduced in 1974 for the *Intel 8080*. This OS contributed significantly to reducing the configuration effort required to install software on new hardware. Thus, it is certainly one of the driving forces behind the further spread of "home" computers. So it is not surprising that many major software products had their first versions under *CP/M*. These include *WordStar* (the first, widely used word processing program), *dBase* (a popular *DBMS*), and *Microsoft Multiplan* (a predecessor of Excel). [11]

3 Discovering Multicomp

3.1 The Goal

The goal of this project is to implement a *Zilog Z80* with *Multicomp* and then to install and start *CP/M* and run various software.

Furthermore, the necessary process should be documented in detail and, where possible, simplified so that everyone – even without previous knowledge – can configure a *Z80* with *CP/M*.

3.2 The Setup

To achieve the goal, the first step is to connect the hardware. There are various possibilities for this, which can be used depending on the desired range of functions.

The most basic variant uses only the serial interface for communication and has an external RAM and an SD card. The *FPGA* – more specifically a *Altera Cyclone II* Board – handles the ROM and all connections from and to the processor. This basic structure is shown in Figure 1.

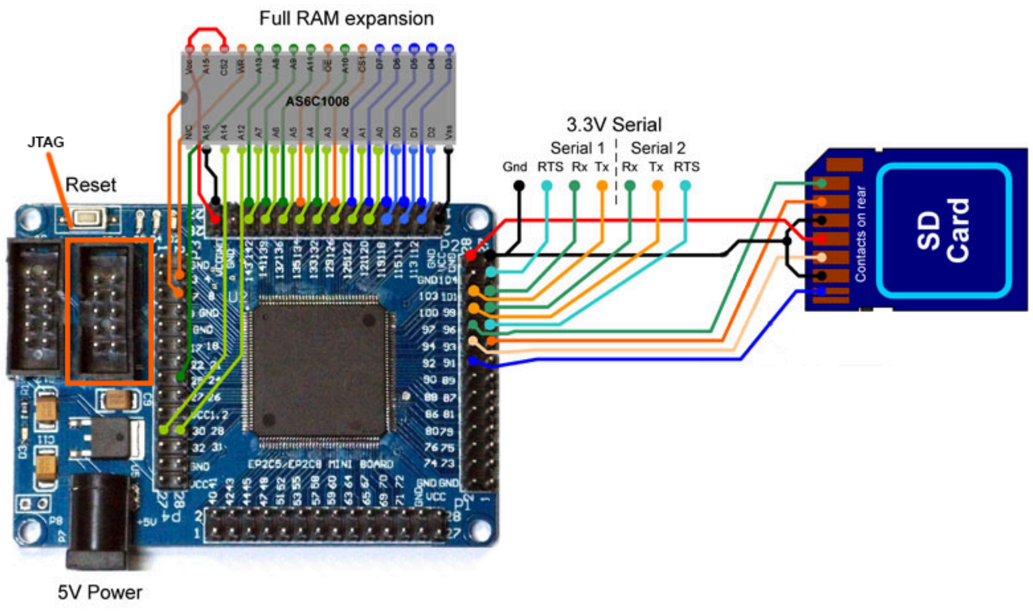


Figure 1: Basic setup [19]

With this structure, the whole project can be implemented. With additional components, the setup can be extended by a monitor connection and a keyboard. Figure 2 shows the *FPGA* with all supported peripherals. One can connect an external monitor in different ways; it is even possible to output a video signal via *Analog Video-Out*, *VGA* or *SCART*. When choosing the components, one has to pay attention to the compatibility among each other. For example, it is not possible to use an *SDHC*, *SDXC* or *SDUC* card. It is possible to use three different processors, the *Zilog Z80*, the *MOS Technology 6502* and the *Motorola 6809*. The *Multicomp* kit provides the necessary resources for each processor to simulate it on the *FPGA* and to make all connections correctly. The *FPGA* also simulates one component for serial communication and one for SD card access. *CP/M* later perceives the SD card as 16 floppy disks, each with 8 MB memory.

3.3 The Challenges

The first, expected challenge, came with the minimum requirements of *CP/M*. For *CP/M* 2.2 at least 20KB of memory are needed, [2] but with *Multicomp* it is only possible to implement 4KB of memory directly on the FPGA. So an external RAM chip had to be purchased and connected. The choice fell on the fast available *AKM6264ALP-12* 64KB *SRAM* chip.

After this was connected, the next problem appeared. Formatting of the SD card as described in the Multicomp instructions was not possible. Apparently, our chip was either faulty or not compatible with Multicomp, because a *BASIC* Memory-Check (to be found in the project repository [9]) showed at irregular intervals bytes that could not be written. An unusually large number of these faulty bytes were in the area where the disk formatting program was written.

So it was necessary to write the program into a different memory area, but the memory area was hardcoded into the assembler program code; so it was necessary to recompile the program, the *TASM Assembler* from Speech Technology Incorporated had to be used for this. The required version was only available for *DOS*. Fortunately, the *DOSBox* [3] project allows to run DOS software on modern Windows, Linux and macOS computers. So we could recompile the program for a different memory area. Afterwards, the disk formatting program was transferred and was then executable.

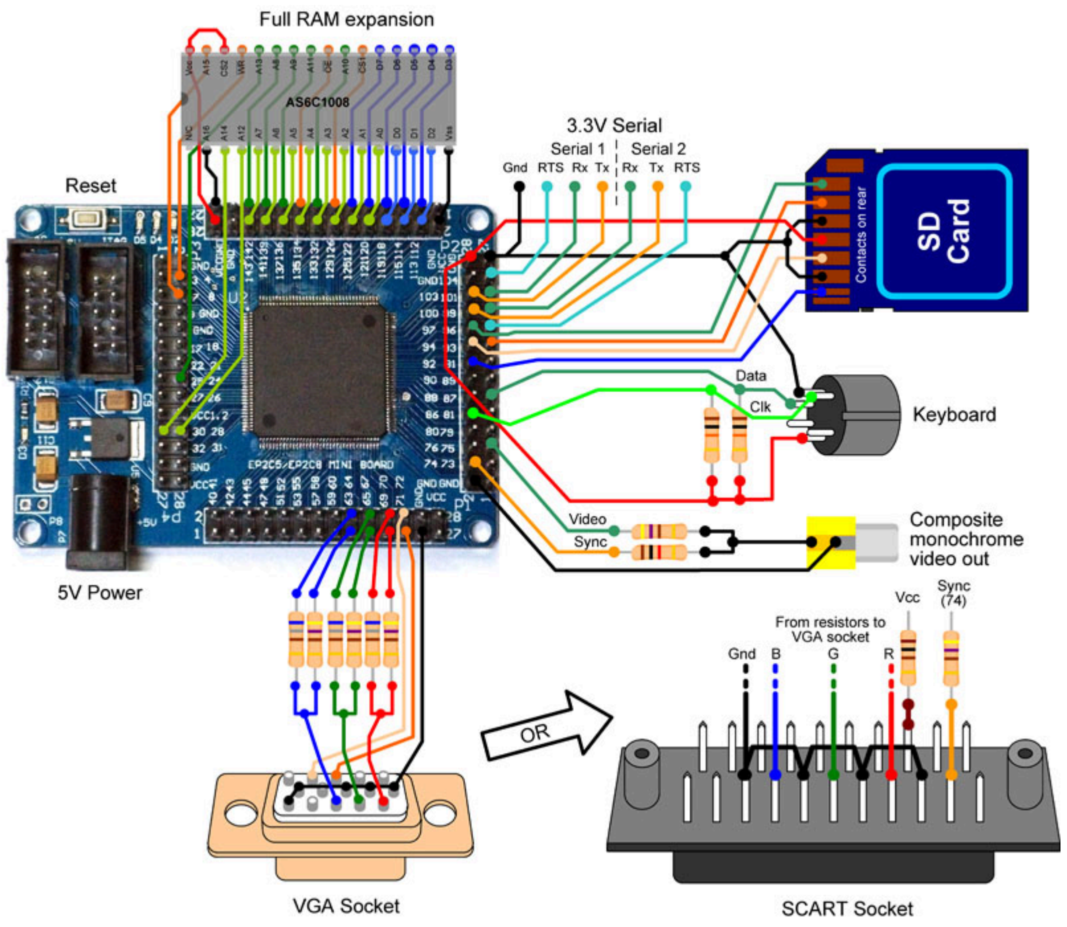


Figure 2: Setup with all supported hardware extensions [19]

A minor problem was then only the connection of the SD card. It seemed to be reasonable to solder only a micro SD card adapter to avoid soldering a real SD card and possibly destroying it. Unfortunately, the adapters we tested were not compatible with Multicomp, and finally, we had to solder an SD card. Afterwards, the formatting ran without further problems.

So we assumed that we could also install *CP/M* with a changed memory area, but *CP/M* needed much more memory than the formatting program and there was no sufficient error-free memory segment to be found. So we decided to order the exact *SRAM* chip (*AS6C1008-55PCN*), which was also used for the original *Multicomp* project.

Apart from the considerable delivery time of this chip, we were then able to complete the *CP/M* installation as described.

Now, a *CP/M* Installation by itself does not do much: Only the absolute fundamentals programs for the "navigation" of the file system are included. As mentioned, our goal was, the installation of additional software; but how does one get the software onto the FPGA? We did not connect a floppy disk drive, and the SD card was soldered. If one would unsolder it, clean it and connect it to a modern computer, one could, for example, transfer Oscar Vermeulen's Demo Disc. [21] However, this solution is very inflexible. Better is the approach Grant Searle had already taken: the transmission over the serial interface. For this purpose, Multicomp also provided a packager and download client. The download client could be loaded into memory in *Intel-HEX*[8] format via *CP/M-ROM* and then saved in *CP/M* on the SD card. Afterwards one could convert *.COM* program files into an *ASCII* format using the packager, which was then transferred via copy-paste into an open serial terminal running the Downloader.

This process posed a big problem: The transfer via copy-paste is very unreliable: modern terminals like to insert text in whole chunks, i.e. several bytes at the same time, and the *CP/M* system does not keep up. The terminal client must, therefore, be configured in such a way that only individual characters are pasted. Besides, the Downloader must be waited for, especially after line breaks, because a checksum is still calculated and compared.

If the checksum was incorrect, the entire file must be manually retransmitted. Moreover, that can – due to the character by character transfer – mean another waiting time of 10-20 minutes. This process is by no means satisfactory, especially considering how many factors can influence the transfer.

Furthermore, the packager's user experience was anything but good. The installation did not run smoothly and only worked under Windows. The Drag&Drop insertion of files did not work; additionally, there were almost no configuration options. (See 3a)

True to the goal of this project to make Multicomp accessible to everyone, we decided to write a new Packager. The result is the Z80-Uploader [10] (See 3b). With this packager anyone can easily upload programs to the Z80 resp. *CP/M*; more details can be found in the next section.

4 What we achieved

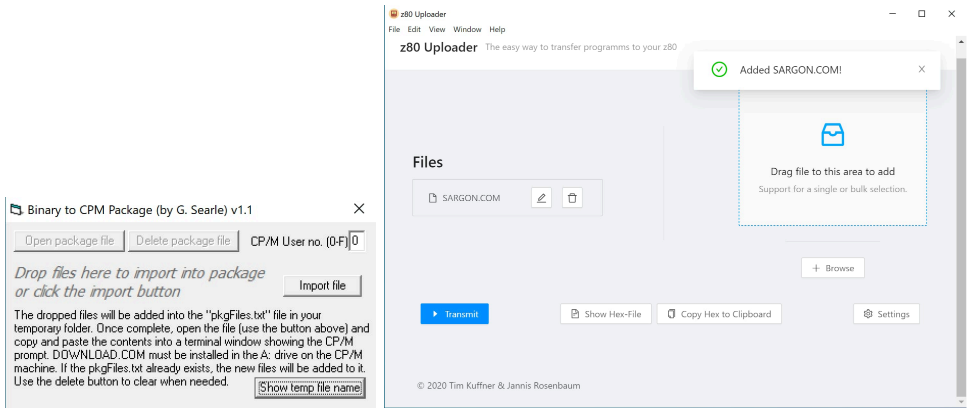
After overcoming these problems, we had a functioning *CP/M* installation. Furthermore we could transfer and run software like *TurboPascal*, *dBase* or *Sargon*. We were also able to document our project extensively and publish this documentation open-source on Github: [9] In doing so, we wrote more than 5000 words and several code examples creating two guides, several contextual classifications, as well as numerous recommendations on how to use *BASIC* and *CP/M*. Thus, the inclined retro-computing beginner will find everything he needs to learn a lot about the historical *Z80* processor and *CP/M* software – without losing the fun when encountering the numerous stumbling blocks.

No less elaborate is the Z80-Uploader, [10] whose architecture we would like – as announced – to explain in more detail at this point:

The "Uploader" is an electron application and is mainly based on *Typescript* and *React*. It can translate (/packetise) any files (e.g. executables or text files) into the *INTEL-Hex* [8] format and then transfer them to the Z80 via the serial interface of the host computer. Our corresponding downloader client can read *Intel-Hex* files under consideration of multiple checksums and save them as a file. The Downloader is written in *TurboPascal* and is specially designed for *CPM-80* systems, but should also run under e.g. *DOS* with no/minimum adjustments. Of course, the Downloader and the Uploader can also be used separately, but when used together, they have automatic error detection and correction, which makes it easy to send large amounts of data over the unreliable serial connections.

4.1 Which enhancement potentials are there ?

The most straightforward extension would be to try out more software. It would also be interesting to research how the user experience of software has changed over the years, or which user interaction concepts were first tried out under *CP/M*.


Binary to CPM Package (by G. Searle) v1.1
Open package file
Delete package file
CP/M User no. (0-F) 0
Drop files here to import into package or click the import button
Import file
Show temp file name
z80 Uploader
File Edit View Window Help
z80 Uploader The easy way to transfer programms to your z80
Added SARGON.COM!
Files
SARGON.COM
Drag file to this area to add
Support for a single or bulk selection.
Browse
Transmit
Show Hex-File
Copy Hex to Clipboard
Settings
© 2020 Tim Kuffner & Jannis Rosenbaum

The dropped files will be added into the "pkgFiles.txt" file in your temporary folder. Once complete, open the file (use the button above) and copy and paste the contents into a terminal window showing the CP/M prompt. DOWNLOAD.COM must be installed in the A: drive on the CP/M machine. If the pkgFiles.txt already exists, the new files will be added to it. Use the delete button to clear when needed.

(a) The Multicomp Packager **(b)** The Z80-Uploader

Figure 3: Comparison between the old and new "Packager"

A more in-depth examination of *BDOS*, the *kernel* of *CP/M* is also conceivable; for example, in the form of a hardware-related software project.

Concerning the keyword "hardware-related", other accessory devices such as printers or floppy disk drives are of course also of particular interest. Obtaining these and then connecting them to the FPGA would undoubtedly be an educational experience.

Finally, *Multicomp* also offers – as described – other processors; these in turn, of course, offer other operating systems that could be installed. However, the biggest challenge would be to write a separate ROM to install the operating system. This programming would be most likely possible with reasonable effort since the *Multicomp*-assembler-source code is mostly available.

4.2 What will the future bring for similar FPGA Projects ?

Like much in the technology industry, *FPGA*s have become cheaper and more potent in recent years. *FPGA*s suitable for retro computing are available for about 20€. Also, the developer tools and documentation e.g. for *VHDL* programming, have become increasingly better in recent years. [16]

However, this is also necessary because newer processors are of course more complex than the 8-bit processors of the Multicomp project. Furthermore, reverse engineering of processors is hardly possible, at least of current processors, so it is therefore up to the manufacturer or license holder to publicize their processor.

This publication often happens in the form of a so-called *Softcore*, that is a processor (core), which is available as software and can be easily integrated into a *ASIC* or *FPGA*. Depending on the license conditions, everyone can integrate the processor into their hardware-projects or, for example, for retro-computing on an *FPGA*. [5]

Due to the value of such an architecture, this happens very rarely; nevertheless, IBM, for example, has recently published its *power* (known from the *PowerPC*-Processors) architecture as open-source. [14] Furthermore, there are also other approaches about which we will report in the following section.

4.3 Related Work

The consensus is that open CPUs and soft microprocessors – including soft cores – offer an excellent opportunity for the OpenSource and Maker community. Therefore there have been constant attempts in recent years to implement or emulate other historical/current OpenSource architectures. At this point, we want to introduce a few of these projects.

Directly comparable to Multicomp and documented with similar effort is the project of S. Edwards to implement a *Apple II* on an FPGA. [4] Further there are projects for Z1013, [6] ZX Spectrum and BBC Micro. [20] Usually, if there is a softcore, there is probably also an FPGA implementation attempt.

What Multicomp is for processors 8-bit from the 70/80s is of course also available for newer architectures.

There is the *Zet* processor, which is machine-code compatible with 16-bit *x86* hardware, allowing it to run *DOS* or *Windows 3.0*. [13]

Or the 32-bit *ARM*-compatible amber core, but due to the lack of an MMU it can only run special Linux variants. [17]

More interesting is the *OpenSPARC* project of Sun Microsystems [22] or the comparable OpenRISC project, [15] which provide various 32-/64bit multicore/threading processors, that however have their own architecture, i.e. can only execute software that has been explicitly adapted for *OpenSPARC/-RISC*. This software includes several major Linux distributions.

Nevertheless, the probably best-known project of the Open-/Softcore community is certainly *RISC-V*. [1] *RISC-V* is, first of all, only an open instruction set architecture; however, there are already numerous softcores that implement the instruction set and are thus mostly compatible with software developed for commercial instruction set implementations.

Of course, the last-mentioned projects are not directly comparable to *Multicomp*, because *Multicomp* allows implementing "real" historical processors. But should these processors/architectures become established and be supported by the major operating systems – it will be easy for the tinkerers and "software archaeologists" of tomorrow to continue to run this software for many years to come. However, an assessment of whether and when *RISC-V* or *OpenSPARC/-RISC* will be real competitors to *AMD64*, *Intel-64* or *ARM*, would be out of scope here.

Multicomp was also implemented for other FPGA hardware, among others the successor model *Altera Cyclone IV*. Code and hardware can be found in Doug Gilliland's GitHub repository. [7]

5 Conclusion

We were able to achieve our set goal: We have completely reconstructed the *Multicomp* project and have a working *CP/M* on a *Z80* Implementation on our *FPGA*. We have also successfully installed and executed software. In the form of the *Downloader* we also had contact with the software development under C/PM with TurboPascal. We documented the steps and learnings both in this paper and in even greater depth in an OpenSource project. Thus, a basis and a starting point for further discussions about *Multicomp* and retro-computing on *FPGA*s, in general, are laid. Based on our experience with this project, we believe that *softcores* on *FPGA*s are an excellent alternative to cumbersome hardware projects or emulators. The related projects we mentioned also show that tomorrow's hardware will perhaps be much easier to implement on *FPGA*s. Progress we will watch with great interest.

References

[1] *About RISC-V*. `www.riscv.org/about/`. Online; accessed 30-September-2020. 2020.

[2] *CP/M 2.2 Operating System Manual*. Online; accessed 30-September-2020. Digital Research Inc., 1983.

[3] *DOSBox: An x86 emulator with DOS*. `www.dosbox.com`. Online; accessed 30-September-2020.

[4] S. A. Edwards. "Retrocomputing on an FPGA". In: Online; accessed 30-September-2020. 2009.

[5] *FPGA Soft Core*. `www.mikrocontroller.net/articles/FPGA_Soft_Core`. Online; accessed 30-September-2020. 2020.

[6] Fpgakuechle. *Retrocomputing auf FPGA*. `www.mikrocontroller.net/articles/Retrocomputing_auf_FPGA`. Online; accessed 30-September-2020. 2013.

[7] D. Gilliland. *Spins of Grant Searle's MultiComp*. `www.github.com/douggilliland/MultiComp`. Online; accessed 30-September-2020. 2020.

[8] T. Kuffner and J. Rosenbaum. *The Intel-HEX Format*. `www.github.com/sinnaj-r/z80-on-an-fpga/blob/master/The_Intel_HEX_Format.md`. Online; accessed 30-September-2020. 2020.

[9] T. Kuffner and J. Rosenbaum. *Z80 on an FPGA Github Repository*. `www.github.com/sinnaj-r/z80-on-an-fpga`. Online; accessed 30-September-2020. 2020.

[10] T. Kuffner and J. Rosenbaum. *Z80 Uploader Github Repository*. `www.github.com/sinnaj-r/z80-uploader`. Online; accessed 30-September-2020. 2020.

[11] B. Leitenberger. *CP/M: der erste Betriebssystemstandard für PCs*. `www.bernd-leitenberger.de/cpm.shtml`. Online; accessed 30-September-2020. 2003.

[12] B. Leitenberger. *Intels Niederlage - Der Z80*. `www.bernd-leitenberger.de/z80.shtml`. Online; accessed 30-September-2020. 2003.

[13] G. Marmolejo. *Zet - The x86 (IA-32) open implementation*. `www.opencores.org/projects/zet86`. Online; accessed 30-September-2020. 2008.

[14] *Open Power CPU: Open-Source-ISA als letzte Chance*. `https://www.golem.de/news/open-power-cpu-open-source-isa-als-letzte-chance-2001-145893-3.html`. Online; accessed 30-September-2020.

[15] *OpenRISC*. `www.openrisc.io`. Online; accessed 30-September-2020. 2020.

[16] T. Richter and Y. Krasteva. "Scale-In, Then Scale-Out - MPP Postgres Database with FPGA Acceleration". `www.tele-task.de/lecture/video/7870`. Online; accessed 30-September-2020. 2019.

[17] C. Santifort. *Amber - ARM-compatible core*. `www.opencores.org/projects/amber`. Online; accessed 30-September-2020. 2010.

[18] G. Searle. *Grant's Homebuilt Electronics Page*. `www.searle.wales`. Online; accessed 30-September-2020. 2014.

[19] G. Searle. *Grant's MULTICOMP pick and mix computer*. `http://searle.x10host.com/Multicomp/index.html`. Online; accessed 30-September-2020. 2014.

[20] M. Stirling. *ZX Spectrum and BBC Micro on FPGA*. `www.mike-stirling.com/2016/01/zx-spectrum-and-bbc-micro-vhdl-on-github/`. Online; accessed 30-September-2020. 2011.

[21] O. Vermeulen. *Multicomp FPGA - CP/M Demo Disk*. `www.obsolescence.wixsite.com/obsolescence/multicomp-fpga-cpm-demo-disk`. Online; accessed 30-September-2020.

[22] D. L. Weaver. *OpenSPARC internals: OpenSPARC T1/T2 CMT throughput computing*. English. OCLC: 601007974. Santa Clara, CA: Sun Microsystems, 2008. ISBN: 9780557019748.

[23] *What is an FPGA*. Online; accessed 30-September-2020. Xilinx Inc., 2020.

Portable Executables
ISA Independent Executables for Linux

Linus Hagemann and Tom Wollnik

Hasso Plattner Institute for Digital Engineering
linus.hagemann@student.hpi.uni-potsdam.de
tom.wollnik@student.hpi.uni-potsdam.de

We present an approach to build executables that can be ported between different instruction set architectures (ISAs). Our approach draws on the LLVM Intermediate Representation (IR).[1] We make use of a wrapper around `clang`, [2] the `LLVM`[3] compiler for the `C` programming language. The IR allows for recompilation of the application if necessary. We provide a prototypical implementation that can build a portable version of GNU `sed`.[4] We observe some drawbacks (e.g. performance impact) of our approach and propose how future work might mitigate these.

1 Context and Motivation

The ability to run programs on CPUs with different Instruction Set Architectures is becoming increasingly important. The dominance of Intel x86 CPUs in modern mobile and desktop computers is being challenged by ARM.[5] Apart from this shift in consumer devices, data centers are also becoming more heterogeneous. [4] For example, one may find Intel x86, ARM, and PowerPC CPUs all in one data center. Also, there is a wide range of possible server configurations, e.g. with regard to the available accelerators. This strengthens the need for executables that can easily be ported between machines with different ISAs and different accelerator configurations.

A successful implementation of ISA independent executables could also support efforts for energy aware computing by making it easier to migrate programs and compute jobs between different machines. This aids the compute load balancing across servers in a data center based on the energy supply. [3]

[1]LLVM Language Reference Manual. `https://llvm.org/docs/LangRef.html#abstract` (accessed on 09 Sept 2020).
[2]Clang C language family frontend for LLVM. `https://clang.llvm.org` (accessed on 09 Sept 2020).
[3]The LLVM Compiler Infrastructure Project. `https://llvm.org` (accessed on 09 Sept 2020).
[4]GNU sed. `https://www.gnu.org/software/sed` (accessed on 09 Sept 2020).
[5]E.g. Apple announced in June 2020 that they will transition their computer lineup from x86 to ARM based Apple Silicon over the next two years.

2 Related Work

A multitude of approaches for portable binaries have been proposed throughout the years.

In this section we want to highlight some approaches that are strongly related to the one we chose for our project and also briefly comment on the respective differences.

2.1 Fat Binaries

The approach that arguably had the most impact for a wide range of users is known as Fat binaries, in which an executable contains code native to different ISAs. [2] This results in file sizes larger than standard executables, thus their name. Often it is possible to engineer such formats in a way that some parts of the application (e.g. assets) can be used for both ISAs. For this reason Fat binaries can be smaller than the sizes of the respective standard executables combined.

For users, the advantage of Fat binaries is that only a single file has to be provided for installation. Therefore, no special knowledge is required to identify which version of a program should be installed. Additionally, this advantage does not come with a significant performance overhead, since native code is executed and the selection of the appropriate code is supported directly by the operating system. The only disadvantage is larger file sizes.

The following implementations supported only a predetermined combination of ISAs.

2.1.1 Apple's Universal Binaries

Apple's transitions from PowerPC to Intel and from Intel to ARM are well known cases of the usage of Fat binaries. In both instances, native code for both ISAs was contained in a single file. Additionally, different word lengths could be supported.[6] In June 2020, Apple announced the new version Universal 2 for the transition of x86 to ARM for their Macs and Macbooks.[7]

2.1.2 FatELF

`FatELF` is a prototypical implementation of the Fat binary approach for `GNU/Linux` and its `ELF` (Executable and Linkable Format)[8] format. Different ELF files are concatenated with some additional header information. The proof of concept implementation is available for Ubuntu 9.04. However, since currently there is no

[6]Mac OS X ABI Mach-O File Format Reference. `https://web.archive.org/web/20090901205800/http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html#//apple_ref/doc/uid/20001298-154889` (accessed on 09 Sept 2020).

[7]Building a Universal macOS Binary. `https://developer.apple.com/documentation/xcode/building_a_universal_macos_binary` (accessed on 09 Sept 2020).

[8]Executable and Linking Format (ELF) Specification, Version 1.2. `https://refspecs.linuxfoundation.org/elf/elf.pdf` (accessed on 09 Sept 2020).

integration of this work into the mainline Linux Kernel and none is planned as of this writing, involved changes to ones OS are necessary in order to get support for `FatELF`.[9]

2.2 Binary Translation

Another approach for portable executables is the translation of binary files before their execution. Therefore, the executable itself is not strictly portable but there exists tooling that translates code native to another ISA before or while a program is executed for the first time. This process is transparent to the user, although it takes significant time. However, this translation has to happen only once. This makes it overall faster than emulation, if the program is used sufficiently often since in an emulated environment one has to deal with a constant overhead. [5] Modern approaches use different strategies such as caching and memoization in order to translate as few lines of code as possible.

Again, Apple provides a prominent example for this approach with Rosetta and Rosetta 2.[10]

2.3 Comparison to this project

We will see in section 4 that our implementation builds on both of the described ideas.

We recompile code for different ISAs from an architecture independent intermediate representation. Additionally, we store the generated `ELF` files in order to only compile once per target configuration.

This leaves us with a worse performing solution (both in terms of space and performance) than the solutions above (see 6). Additionally, we cannot bridge differences in processor word-length.

However, the on demand recompilation makes the provided prototype more flexible: Our review of "The Architecture Of Open Source Applications" [1] and the `LLVM` Documentation[11] suggest that the only requirement for an ISA to be supported is that an `LLVM` backend exists. The set of supported ISAs does not have to be predetermined and no substantial changes to code, build tools or operating system are necessary in order to use our prototype.

[9]FatELF: Universal Binaries for Linux. `https://icculus.org/fatelf/` (accessed on 09 Sept 2020).
[10]Apple–Rosetta. `https://www.apple.com/rosetta/index.html` (accessed on 09 Sept 2020).
[11]LLVM Documentation: Writing an `LLVM` Backend. `https://llvm.org/docs/WritingAnLLVMBackend.html` (accessed on 25 Sept 2020).

3 Requirements and Goals

There are three main requirements for our implementation of portable executables.

The executable should be portable and extensible. Portable executables need to be able to run on machines with different ISAs. Additionally, they should be able to run on machines where different accelerators are available. It should not be necessary to know the concrete configurations of these setups in advance.

It should be easy to create and run portable executables. Firstly, this means that users should not have to make significant changes to their existing workflows. We want to accomplish this by providing a compiler and a linker that can act as drop-in replacements for the `clang` compiler and linker in C projects.

Secondly, there should be only a minimal administration overhead. Our tools should only use standard dependencies and no configuration should be necesseary to get started.

Thirdly, users who run the portable executable should not notice a difference to running a standard executable. Any startup logic needs to be hidden from the end user.

The first language to support is C.

4 Implemented Prototype

Our approach builds on the LLVM Intermediate Representation to generate portable executables. We want to explore the use of compiler intermediate products for program portability. A working prototype using the LLVM IR can provide valuable insights into challenges and possibilities of this implementation strategy for portable executables. A reason for choosing LLVM IR is that a portable format using it could make it possible to bridge a multitude of programming languages and target architectures (including GPUs), due to the large and rising number of available front- and backends. LLVM also allows to perform architecture independent optimizations on the IR, which would enable optimizations prior to distribution and recompilations of a portable executable that uses the proposed approach.

We use a container format. The general idea is that a file in our PEX format contains a `tar` archive with the LLVM IR for the program and executables in the ELF format for different ISAs and compiler flag configurations.

4.1 LLVM Intermediate Representation

The LLVM IR is the foundation of our portable executable format. LLVM IR is an intermediate, assembly-like code format used by the LLVM compiler suite. It is designed to act as an intermediary between compiler frontends for different programming languages and backends for different target architectures. The IR is mostly machine-independent. There are some portability issues, as discussed in subsection 6.4.

As an example, listing 1 shows the C code and listing 2 shows the corresponding LLVM IR for a simple Hello World program. We will come back to this example when discussing the limitations of IR portability in subsection 6.4.

We make use of the LLVM IR's portability to build portable executables. Portable executables in our format contain the LLVM IR for the program. The specifics are discussed below in subsection 4.2. When a PEX file is executed on a machine with an ISA unknown to this PEX file, we can compile the program for the new ISA from the IR that is stored in the PEX file. If it is possible to treat the generated LLVM IR as sufficiently architecture agnostic it would always be possible to compile on any target architecture with an LLVM backend available. In theory, this does not have to be the compiler initially used to generate the IR (e.g. `clang`). This would allow for a flexible usage of the portable executables, due to less strict requirements regarding the software available for execution.

Listing 1: Simple Hello World program in C.

```c
#include <stdio.h>
int main() {
    printf("Hello World!");
    return 0;
}
```

Listing 2: LLVM IR for the simple Hello World program (shortened).

```
; ModuleID = 'helloworld.c'
source_filename = "helloworld.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [13 x i8]
            c"Hello World!\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(
      i8* getelementptr inbounds (
              [13 x i8], [13 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = {[...] "target-cpu"="x86-64"
    "target-features"="+fxsr,+mmx,+sse,
                             +sse2,+x87" [...] }
[...]
```

4.2 PEX Format

Having discussed the LLVM IR and how it can be used to achieve portability we will now take a closer look at the file format developed in this project.[12] We use a container format as depicted in Figure 1. PEX files consist of a loader Shell script and an integrated `tar` archive.

Tar archive. The `tar` archive in a PEX file contains the LLVM IR for all source files of its program, linker flags, and one or more bundles. A bundle is the compiled program for one ISA or one compiler flag configuration and its object files. Listing 3 shows the contents of the `tar` file for an example Hello World PEX file. The file `helloworld.ll` contains the IR for the single source file of the program. `LINKER_FLAGS` stores the flags that were used to link the different object files when the PEX was first created (i.e., the linker call in the Makefile, see section 4.3). The example `tar` archive contains two bundles: `aarch64-unknown-linux-gnu` and `x86_64-pc-linux-gnu`. Each bundle holds the actual executable for one ISA (file `a.out`) and the object files compiled for this ISA. The names of the bundles tell us which ISA and OS the bundles were created for.

The loader script. The loader is a shell script that is run when a PEX file is executed. It unpacks the tar archive, recompiles the program from the IR if necessary, and executes the program. Note that the need for a recompilation is determined by checking if a bundle for the current machine exists (via `clang -dumpmachine`) as a default. However, this behavior can be overridden. In that case the user specifies a bundle to create/execute. This allows for multiple configurations per ISA and operating system, e.g. due to different available accelerators.

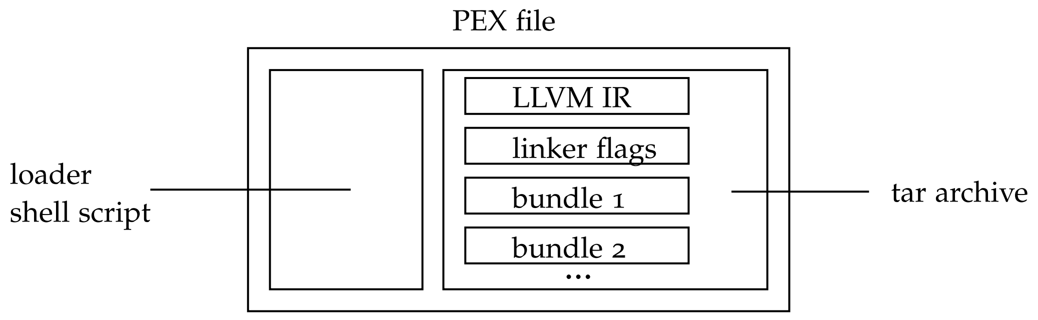


Figure 1: The proposed portable executable format PEX

Listing 3: Contents of the tar archive for a simple hello world PEX file

```
IR
   |--helloworld.ll
```

[12]Portable Binaries Project Repository. `https://github.com/linusha/Portable-Binaries` (accessed on 09 Sept 2020).

```
LINKER_FLAGS
aarch64-unknown-linux-gnu
   |-- a.out
   |-- src
       |-- helloworld.o
x86_64-pc-linux-gnu
   |-- a.out
   |-- src
       |-- helloworld.o
```

4.3 Workflow

In previous subsections we have looked at the PEX format. Now we want to describe how PEX files can be created for C projects. Given is an example Makefile that uses `clang` and compiles a simple C project with just two files (listing 4). To generate a PEX file instead of a standard executable we only need to change the compiler and linker. This is done by setting the `CC` and the `LD` variables in the Makefile appropriately. This modified Makefile creates portable executables in the PEX format.

The `pex` compiler and linker can be used as a drop-in replacements for `clang` in most projects. This makes it simple to start generating PEX files. Note that we currently only support workflows with a single linker call that only links object files (see section 6.3).

Listing 4: Makefile that compiles a small C project. Left: Original Makefile that uses `clang`. Right: Modified Makefile that uses the pex compiler and linker to create PEX files.

```
CC = clang
LD = clang

a.out: foo.o bar.o
	$(LD) -o $@ $^
%.o: foo.c bar.c
	$(CC) -c -o $@ $<
```

```
CC = pex
LD = pex

a.out: foo.o bar.o
	$(LD) -o $@ $^
%.o: foo.c bar.c
	$(CC) -c -o $@ $<
```

4.4 Managing PEX Files

To make working with PEX files more comfortable we provide a small tool that allows for the inspection and modification of PEX files. This manager program supports the following operations:

- `ls` and `tree`. These commands list the contents of the `tar` archive from a given PEX file using `ls` or `tree`.[13]

[13]tree(1) - Linux man page. `https://linux.die.net/man/1/tree` (accessed on 09 Sept 2020).

- `rm`. The manager program can remove a bundle from a PEX file. This is especially useful to force recompilation for a configuration that the PEX file already knows.
- `extract`. Extract the content of the `tar` archive of a PEX file.
- `merge`. This command adds the possibility of merging two PEX files with disjoint bundle names into a new PEX file. Users can now compile a program multiple times using different compiler flags. The resulting PEX files can be merged and users can choose which program version should be used each time the resulting PEX file is executed. This allows for a great degree of flexibility and enables PEX files to use different accelerators depending on the compiler flags that were set for the selected bundle. This feature is directly related to the requirement that the executable should be portable accross accelerator configurations.

4.5 Summary

PEX files consist of a loader script and a `tar` archive. The `tar` archive contains the IR, linker flags, and one or more bundles. The IR is used to recompile the program for previously unknown ISAs. We also provide a management program.

5 Proof of Concept

To demonstrate a more advanced usage of our prototype we built a PEX file for the text editor GNU `sed` on a x86 machine and an aarch machine.[14] Both builds then could be executed on the other architecture.

We use `sed` here as an example since it is a complex, large, and well-tested `C` project. This makes it suitable as a proof of concept, since presumably all common features of the C language are used. By successfully compiling `sed` as a PEX file we can therefore expect our tool to work for all `C` programs, within the limitations discussed later in section 6.

To compile `sed` to a PEX we set the `CC` and `CPP` variables in `sed`'s Makefile to `pex` and `pex -E`, respectively.

However, since this build process is more advanced than the processes we looked at before, another modification has to be made in order to receive a working PEX File. In the unmodified `sed` build process some object files are bundled as an `archive` (.a files). `Archives` are currently not supported by our implementation (see section 6 and 7). Therefore, we have to make a final call to `pex` with all object files

[14]Our testing setup for this section, as well as for the discussions in section 6 consists of one machine running a) an Intel x86_64 processor running GNU/Linux (4.11.0-13-generic) Ubuntu 16.04 and clang version 3.8.0-2ubuntu4 and b) an aarch64 processor running GNU/Linux (4.9.216-69) Ubuntu 18.04 with clang version 6.0.0-1ubuntu2.

generated during the unmodified build process. More details on this can be found in the project repository.[15]

6 Limitations

We want to discuss some limitations of our current implementation, as well as some limitations of the underlying approach.

6.1 Replicating `clang` Behavior

A general problem arises from the fact that `pex` replaces `clang` in compilation workflows. Since the behavior of `clang` differs depending on the given flags our implementation needs to mimic these behaviors as well. However, because of the variety of possible flags and their combinations we cannot guarantee correct behavior in all cases. We provide POSIX compliant behavior for all combinations of the `-o`, `-E`, and `-c` flags.

6.2 Performance Implications

Our approach of recompiling the entire program and bundling files in a `tar` archive has some serious performance implications.

First, a `PEX` file is significantly larger than the corresponding standard `ELF` file. For our `sed` example this means that a PEX generated on a x86 Ubuntu machine without any additional bundles is approximately 24MB large, while the on-system binary is drastically smaller than even 1MB. The size of a `PEX` file grows with each new architecture it is executed on and with each configuration that is created (i.e., the number of contained bundles).

Second, the current implementation suffers from a significant slowdown at startup time. When we execute a `PEX`, the `loader` script is executed inside a Shell process and the `tar` archive is extracted. Depending on whether a configuration corresponding to the current ISA is found, a recompilation from IR is necessary, as well as recreating the `tar` archive with the new content. For the final execution of the executable contained in the relevant bundle another Shell subprocess is started. For the example of sed we measured the time necessary to execute a simple sed call for a natively compiled sed and our PEX sed. While it took $\sim 0.004s$ for the executions of standard sed, the PEX calls took $\sim 0.02s$ on average.[16] With larger program runtimes the performance implications become less relevant, e.g. because the `tar` archive is extracted only once.

[15]Portable Binaries Repository–Proof of Concept Guide. `https://github.com/linusha/Portable-Binaries/blob/master/Proof_of_Concept_Guide.md` (accessed on 23 Sept 2020).

[16]For details see the project repository. `https://github.com/linusha/Portable-Binaries/blob/master/Proof_of_Concept_Guide.md` (accessed on 23 Sept 2020).

6.3 Complex Build Processes

Another limitation lies with more advanced build processes (as seen in section 5) and library usage.

The problem here is that the knowledge about the build process lies completely outside of our tool, e.g. in a `Makefile`. Even in the simple case of bundling multiple object files inside a `.a` file this becomes a problem. The reason for this is that our tool currently assumes the existence of one source file with `IR` per object file in the project, as described in section 4. For the case of `archives` a possible solution to this problem is discussed in section 7.

Similar problems could also arise when using shared libraries and dynamic linking. However, this was not investigated as part of this project.

6.4 LLVM IR

When LLVM IR is treated as architecture agnostic multiple problems arise.

6.4.1 `clang` Version Differences

Different versions of `clang` produce IR that is not necessarily compatible.

For example, our testing machines ran `clang` version `6.0.0-1ubuntu2` (ARM) and `3.8.0-2ubuntu4` (x86). Initially, the IR created on the ARM machine could not be used to compile on x86.

We could track the problem to a single line that could be removed without altering the functionality of the `IR`. However, it cannot be assumed that there are no other cases in which different `clang` versions may create incompatible IRs or that no such cases will be introduced in the future.

6.4.2 Non-trivial ISA Differences and Dependent Code

Since it was possible to compile a working version of `sed` both from IR created on ARM and on x86, it is justified to assume that the vast majority of C code can correctly be compiled from the corresponding IR. However, cases in which architecture dependencies exist in the C code remain problematic.[17]

Preprocessor Definitions The `C` preprocessor is run before the IR is created. Therefore, architecture dependencies within Macros,... cannot be resolved by recompilation. For example, a definition with `#ifdef __x86_64__` will not be kept in the IR if first compiled on ARM, thus leading to unexpected behavior if one later uses that IR to compile an executable on x86.

[17]Examples for these can be found in the `caveats` folder of `https://gitlab.hpi.de/osm/portable-binary`.

Data Type Sizes Differences in the bit-length of data types are hard coded into the IR, as seen in e.g. listing 2 line 3. Therefore, the current implementation does not allow for portability from e.g. a 16 Bit system to one with a 64 Bit architecture.

Inline Assembler If inline assembler code is present in the program (using `__asm__`,...) the resulting IR will not be portable due to the platform specificity of the assembler code.

7 Future Work

As seen in section 6 there are multiple shortcomings of the approach described here and the implementation provided. In this section we want to highlight some areas where substantial improvements can be made.

7.1 Operating System Integration

As we saw earlier, the independence of our software of most parts of the underlying system comes with a significant cost in performance, i.e., startup time. A part of the overhead we observed comes from the fact that we do not execute any ELF directly. Instead multiple other processes are started.

Similar to FatELF an integration via a Kernel patch could be used to solve this problem. But another, less invasive option could also be pursued: One could build an ELF file for the current architecture and then add the other parts of PEX (i.e IR,... see section 4) as data in sections. Additionally a custom handler via `binftm_misc` that triggers the recompilation if necessary should be provided.[18] This could lead to faster startup times with less necessary modifications of ones system. An approach like this also allows for PEX files to be correctly recognized as binary executables (e.g. with the `file` command).

7.2 Additional Language Support

Since we build upon the IR generated by an LLVM Compiler, it should be possible to transfer the presented approach and large parts of the presented implementation to other languages that have an LLVM frontend available.

7.3 Support for More Build Processes

The current implementation only allows for one type of simple build process, in which multiple `.c` files are compiled into multiple `.o` files which are then linked into one executable.

[18]Portability Experiments for LLVM Intermediate Representation. `https://gitlab.hpi.de/osm/portable-binary` (accessed on 09 Sept 2020).

7.3.1 Enhancement of the Presented Implementation

As we saw in section 5 our current implementation does not support linking libraries in the form of `.a` files. Since such files are just `archives` of object files it should be possible to adapt our implementation to extract the IR for each file contained in such an archive. We think this is possible without any major modifications to our format or the already existing logic.

7.3.2 Integration into `clang`

More complicated and advanced future work could also deal with the question if and how an approach for portable binaries could be supported directly by `clang` or other LLVM frontends. This would make the adoption of portable binaries easier, since no additional tooling would be necessary. It would also allow for more certainty in regards to the possible usage of flags and allow arbitrarily complex build processes.

7.3.3 How Portable is the LLVM IR?

A central question to the viability of our approach and also most of the proposed future work is: How portable is the IR generated by LLVM? Earlier, we identified some issues (e.g. data type length differences) we currently see as a hard limit for the portability of LLVM IR.

Future research could a) systematically investigate whether the problems named in this work are exhaustive and b) whether they can be mitigated, either due to manipulation of the generated IR or due to changes in the underlying language design.

8 Conclusion

We saw that it is possible to treat LLVM IR as portable between different instruction set architectures and use IR generated on one machine to compile a program on a machine with a different ISA. We presented a prototype that draws on this fact in order to build portable executables. Portability is achieved by recompiling the program if necessary. Our prototype is likely able to support any ISA with an LLVM backend without prior selection, as discussed in subsection 2.3. Integration in existing projects is easy in most cases, since our implementation is a wrapper around `clang`. We were able to demonstrate the potential of our approach by successfully applying it to GNU `sed`.

However, LLVM IR is not completely architecture agnostic and we also could observe implications on file sizes, performance and problems with complicated build processes.

Here we have identified several areas in which future research could improve the presented approach, e.g. regarding the question of how certain compatibility problems in the LLVM IR could be mitigated.

References

[1] A. Brown and G. Wilson. "LLVM". In: *The Architecture of Open Source Applications*. 2012.

[2] M. Franz and T. Kistler. "Slim binaries". In: *Communications of the ACM* 40.12 (1997), pages 87–94.

[3] B. Herzog, T. Hönig, W. Schröder-Preikschat, M. Plauth, S. Köhler, and A. Polze. "Bridging the Gap: Energy-efficient Execution of Software Workloads on Heterogeneous Hardware Components". In: June 2019, pages 428–430. DOI: 10.1145/3307772.3330176.

[4] P. Olivier, S.-H. Kim, and B. Ravindran. "OS support for thread migration and distribution in the fully heterogeneous datacenter". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 2017, pages 174–179.

[5] M. Probst. "Dynamic binary translation". In: *UKUUG Linux Developer's Conference*. Volume 2002. 2002.

Project Kraken
Implementing a network stack on the NinjaStorms OS

Lorenz Woth, Felix Auringer, and Tobias Kantusch

Hasso Plattner Institute for Digital Engineering
`lorenz.woth@student.hpi.de`
`felix.auringer@student.hpi.de`
`tobias.kantusch@student.hpi.de`

The goal of this project was to implement a network stack up to the ARP protocol on the NinjaStorms OS. First we started by writing a driver for the local PCI bus. Therefore we configured the processor to be able to use the PCI bus correctly and developed a PCI driver according to the PCI 2.2 specification afterwards. Following we virtually connected an E1000 network card using the previously configured PCI bus. We then developed a driver for the E1000 that configures it for being able to send synchronically and receive packets asynchronously using interrupts. As the E1000 requires memory addresses to be 16-byte aligned we added a basic Unix-like `malloc` to the system. We continued by implementing the Ethernet protocol as a basis for our network stack and implemented the Address Resolution Protocol (ARP) afterwards as a basis for IPv4 networks. To invalidate cached ARP entries we needed a system time that we added as an additional functionality to the OS.

As a final result we were able to start two instances of our operating system that could then request and respond each other using the ARP protocol. In following work one can use our work as a basis for implementing higher level network protocols like the IPv4 and TCP. To complete the network stack in terms of layers one could then implement the HTTP.

1 How it all began

The NinjaStorms OS is a simple real-time operating system for the Lego Mindstorms EV3 that also runs in a virtual environment. It comes with its own custom C-library that, at the start of our project, only provided a basic version of the `printf` function to output text to the console. The operating system itself contained a basic round-robin scheduler to handle concurrent tasks as well as a timer interrupt for the scheduler. Most importantly the OS was able to boot.

As all members of our team are heavily interested in networking and security, we decided to add basic network functionality to the OS.

2 Context

Before starting our work, we had to understand basic concepts that make up the building blocks for our project.

2.1 TCP/IP Stack

The Internet protocol suite is the conceptual model and set of communications protocols used in the Internet and similar computer networks. It is commonly known as TCP/IP because the foundational protocols in the suite are the Transmission Control Protocol (TCP) and the Internet Protocol (IP). [11]

The link layer defines the networking methods within the scope of the local network link on which hosts communicate without intervening routers. This layer includes the protocols used to describe the local network topology and the interfaces needed to affect the transmission of Internet layer datagrams to next-neighbor hosts. [11] **ARP** and **Ethernet** are part of that lowest network layer.

2.2 QEMU

QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). [10]

2.3 Tap device

Tap devices are a Linux kernel feature that allows you to create virtual network interfaces that appear as real network interfaces. Packets sent to a tap interface are delivered to a userspace program, such as QEMU, that has bound itself to the interface. [1]

2.4 Peripheral Component Interconnect

Peripheral Component Interconnect (PCI) is a local computer bus for attaching hardware devices in a computer and is part of the PCI Local Bus standard. Devices connected to the PCI bus appear to a bus master to be connected directly to its own bus and are assigned addresses in the processor's address space. [7]

2.5 ARM926EJ-S

The **ARM926EJ-S** is part of the ARM9 processor family with an older 32-bit RISC architecture. It is the main processor of the Lego Mindstorms EV3, for which the NinjaStorms OS is designed.

2.6 Intel 82540EM

The **Intel 82540EM** is a PCI compliant network interface card of the 8254x series produced by Intel. [3] It can easily be emulated by QEMU. Because it is commonly referred to as **Intel E1000** we will also use this name.

3 Virtual Machine Setup

Developing an operating system solely on the hardware itself is hard to debug and thus we run the OS in a virtual machine. As virtualizer we use **QEMU**, as done by the original developers of NinjaStorms.

3.1 Basic networking

Since we want to focus on the E1000 as a network device, we need to make sure that this card will be attached. An easy way to do this is using the options `-device e1000,netdev=net0 -netdev user,id=net0` when starting QEMU. This will use the E1000 as the network device and specify a simple user networking backend.

However this was very easy to set up and worked well in the beginning to test our hardware driver, this configuration will only work with TCP and UDP protocols. As a result we would not be able to test our network stack until we reach the transport layer, which is out of scope for our project. This was not feasible and we decided to switch to the more complicated to set up but yet more versatile tap networking.

3.2 Tap networking

As tap devices are supported by the Linux bridge drivers, we can also set up a network bridge and attach QEMU to it. This way we can test our network stack with multiple instances of our os in the same network. Creating the necessary virtual devices under Linux is straightforward using the `ip` tool. The whole setup process is done using the commands shown in Listing 1.

Listing 1: Setup of network devices for QEMU on Linux using `ip`

```
ip tuntap add dev tap0 mode tap   # Create tap device
ip link add br0 type bridge       # Create bridge
ip link set dev tap0 master br0   # Attach tap0 to bridge
ip link set dev br0 up            # Start bridge
ip link set dev tap0 up           # Start tap device
ip address add dev br0 10.0.2.15  # Assign ip to bridge
```

Afterwards QEMU can be started using the options `-device e1000,netdev=net0 -netdev tap,id=net0,ifname=tap0,script=no,downscript=no` instead of the ones shown in subsection 3.1. QEMU will now attach to the previously created tap device `tap0` and use it as the network interface for NinjaStorms. Our host machine can now monitor the network traffic on the `br0` interface as well as send packets via that interface to our vm.

4 PCI bus driver

Extension cards for computers like graphic cards or in our case network cards are most commonly attached via a PCI expansion slot on the motherboard. This exact procedure is emulated by QEMU and thus the E1000 network card that we will use is attached via PCI. We first have to implement a driver for the PCI bus that allows us to configure the attached component.

4.1 Configuration Space

The PCI specification provides for totally software driven initialization and configuration of each device on the PCI Bus via a separate **Configuration Address Space**. [12] All devices attached to the PCI bus are required to provide 256 bytes of configuration registers, also known as the **Configuration Space Header**. For our use it contains the following important information.

The **Device ID/Vendor ID** are used to precisely identify a device and thus the corresponding slot number.

The **Command** register controls the device's ability to generate and respond to PCI cycles, e.g. to act as a bus master or responding to memory space accesses.

The **Base Address Registers** (BARs) are used to gather information on how much and what type of memory is required by a device. They also hold the base addresses of the memory regions.

4.2 Self-configuration

Before we are able to use the PCI bus, we have to do a self configuration of our chip. Looking at the ARM926EJ-S the device id is `0x0002` and the vendor id is `0x0000`. Finding the slot where our chip is plugged in is as easy as scanning the addresses $0x41000000 + (n << 11)$, where $11 \leq n \leq 31$. For each address, we check the first 4 bytes of the Configuration Space Header to obtain the device id and vendor id. Once we find a matching pair we are done.

Afterwards we write the value of n into the `PCI_SELFID` register. This way the chip knows in which slot it is located. Furthermore we make use of the Command register in the chips configuration space header to make it the initiator on the system.

4.3 Device configuration

Now that the `PCI_SELFID` register holds the chips slot number, we scan the normal configuration space located at `0x42000000` to find devices as done in subsection 4.2. After we found a certain device we have to read how much and what type of memory is required. As defined in the PCI specification we write `0xFFFFFFFF` to the BAR to get the encoded size information. We then allocate that memory in one of the PCI memory regions and write the base address of it to the BAR.

Here our first bigger problem occurred. Having written the correct addresses to the E1000 BARs, we were still unable to read its data. The documentation states as the last step for PCI configuration, that one has to *"Set the PCI control registers at `0x10001000` **appropriately** [...]"* and that *"An example of PCI scanning and configuration is provided as an example on the CD."* [5] As the chip is quite old and so is the documentation, it is very unlikely that we find the corresponding CD lying around somewhere. Furthermore the word **appropriate** is very inaccurate.

It took us a while fiddling with different configurations, until we eventually found the **appropriate** settings. The said PCI control registers include the `PCI_IMAPx` registers. They are used to translate our machine addresses to PCI addresses. By writing `0x5` to `PCI_IMAP1` and `0x6` to `PCI_IMAP2`, we configured what values should be used for the high bits of the PCI address bus. Accesses to the memory regions `0x50000000` and `0x60000000` (which are two PCI memory regions) were now translated correctly and we were able to read data from the PCI device.

5 E1000 driver

In order to connect to a network a network interface card (NIC) is needed. We decided to use the common Intel E1000 because it is supported by QEMU and has an extensive documentation. The E1000 is connected via PCI and can therefore be accessed with our PCI bus driver. The next step was to implement a driver for the E1000 so that we can control it to receive and send packets.

5.1 Configuration of the E1000

We start with getting the PCI device for the E1000 whose vendor ID is `0x8086` and device ID is `0x100E`. We can interact with the E1000 in two ways:

Firstly the E1000's registers or flash storage is mapped into system memory via PCI. We configured this mapping by writing addresses to the BARs of the E1000 PCI device. Using the written address with the offset `0x5400` we can now read the MAC address of our network card. For handling incoming packets asynchronously we also need to enable interrupts by setting an interrupt bit mask in the register at the offset `0x00D0`.

Secondly we use a shared memory region on the host to exchange information with the E1000. Therefore we have to enable bus mastering for the E1000 PCI device by setting a bit in the configuration space. This allows the E1000 to initiate direct memory access to the host's memory over the PCI bus which is required for efficiently handling the incoming and outgoing packets.

Both incoming and outgoing packets are stored in buffers in the shared memory accessible by E1000 and host processor. Each packet has a corresponding descriptor that contains the address and length of the packet, the status and other information like a checksum. The descriptors are arranged in circular buffer queues.

This did not work for us at first. Because the buffers need to be 16-byte aligned, and we used stack addresses (which did not have any alignment guarantee so far) the packets were not written into our memory correctly. Thus we wrote a `malloc` function to allocate their memory and guarantee alignment.

We now write information about the queues like length, base address and some control bits into the E1000's registers. The E1000 then can use the buffer addresses in the descriptors to access the host's memory for writing or reading packets. The offsets for the registers and the control bits are well documented in the OSDev Wiki. [6]

5.2 Transmitting a packet

Our driver populates the current transmit descriptor from the ring buffer with the address and length of the data to be sent. It then sets the appropriate command bits for sending in the descriptor. Increasing the queue's end on the E1000 triggers a send of all new descriptors. Our driver waits until the status of the old descriptor has been changed by the E1000 indicating that the packet has been processed. Because of that the sending takes place synchronously and pauses the execution of the current thread.

5.3 Receiving a packet

When receiving a packet the E1000 writes it to the next buffer in the queue. In order to minimize the number of interrupts, it generates an interrupt only after a certain time or number of packets.

The PCI bus is not directly connected to the primary interrupt controller (PIC) and thus the interrupts didn't arrive in our first versions. By looking at the ARM documentation we found out that our processor contains two interrupt controllers. To make the interrupts arrive at the PIC we had to enable a passthrough for our PCI interrupts.

Because the interrupt is asynchronously we want the interrupt service routine to be as quick as possible. Therefore the interrupt handler only clears the interrupt on the E1000 and inserts all new packets in a new packet queue. Those elements will be processed later by a network task.

6 Ethernet

Implementing the ability to send raw packets with the E1000 driver gave us the opportunity to build more abstract layers of the network stack. To make Ninja-Storms work with Ethernet frames, we firstly had to define how the frame should be formatted. We decided to reduce the standard IEEE 802.3 Ethernet frame to the most important fields for rudimentary functionality of Ethernet. After these shortenings our final struct for Ethernet frames looked like that:

Table 1: NinjaStorms OS Ethernet Frame Structure

Destination MAC	**Source MAC**	**EtherType**	**Payload**
6 Byte	6 Byte	2 Byte	??? Byte

To further reduce the complexity, we decided to only implement three of all common EtherTypes. We added a distinction between the most common EtherTypes: `ARP`, `IPv4`, and `IPv6`. Part of our actual implementation was only ARP. Each of these EtherTypes has its own hex value transmitted within the Ethernet header field which is used to determine which subroutine we want to call.

6.1 Receiving Ethernet frames

Handling new Ethernet frames is not invoked directly via the NIC driver. We implemented a small process to handle different EtherTypes in a more structured way. Therefore we created a queue for newly received packets from the E1000 driver. A *network receiving task* runs constantly to check for new packet in the queue. If there is a new packet, the packet will be removed from the queue and starts a process that we call *PDU*[1] *Encapsulation*. It takes the buffer's data from the raw E1000 packet and casts it to an Ethernet frame.

Based on the EtherType further functions will be invoked to handle this Ethernet frame, e.g. for ARP.

6.2 Sending Ethernet frames

Sending Ethernet frames is basically just the construction of a new one based on the given arguments. A function takes source, destination, EtherType and the payload as arguments and builds an Ethernet frame out of it. Sending is performed by the E1000 driver. This task was complicated to solve at first place, because back then `malloc` was not part of our library. Implementing `malloc` made it much easier to move the payload from one function to another without much effort. This simplified and shortened the function a lot.

[1]Protocol Data Unit.

6.3 Little and Big Endianness

Since our NinjaStorms OS is working with little endian, we noticed on receiving our first packets that network related things usually work with big endian. That led to some confusion in the beginning, so we decided to implement some helper functions that translate the byte sequence order for us. As in the POSIX standard defined, we created rudimentary functions to change byte orders from little to big endianness, here called host byte order (little endian) and network byte order (big endian).

7 Address Resolution Protocol

As described in the Ethernet part, a function called *PDU Encapsulation* is responsible to decide what to do with the incoming frame. To determine if the incoming frame is an ARP frame, the EtherType needs to equal: `0x0806`. If that is the case, our ARP subroutine starts. Before explaining the functionalities of our ARP functions, a proper understanding of our ARP table is necessary.

7.1 ARP Table Implementation

The ARP table is basically an array containing so called `arp_table_entry_t` elements. They only contain the IPv4 address, the hardware address and a timestamp when the element has been added to the table. We also considered more sophisticated approaches to implement the ARP table (e.g. a hash table), but we decided to keep it simple yet. In our tests we used a maximum of 10 entries with a maximum living time of 300 seconds.

We implemented functions to read, add, and update entries from the ARP table. Once an entry is used, its timestamp is updated. Entries that are too old are removed. If the ARP table is full of entries, the oldest entry will be replaced. Iterating over the table is performed via basic linear search.

7.2 Receiving ARP Requests

After extracting the payload, we follow the algorithm for packet reception as defined in RFC 826. [9] We check if the protocol type and address length imply IPv4 and the hardware address type and address length imply Ethernet. Then the update procedure follows. The `arp_table_update`-function is invoked with the incoming ARP frame and will either return true, if the update was successful or false if not (the ARP entry will be added to the ARP table later in this function).

Now the ARP reply is constructed. Therefore we create an empty ARP frame and use our own addresses (IPv4 and MAC) as sender and the original sender as new receiver. This reply is sent via the Ethernet sending function, described in section 6.

7.3 Sending ARP Requests

Sending ARP requests is very simple to accomplish. We construct an ARP frame with the NinjaStorms IPv4 and MAC address as source addresses. As ARP Operation Code we use `0x0001`. It indicates the ARP request. This frame is then sent to the broadcast MAC `ff:ff:ff:ff:ff:ff`.

8 Evaluation

In the end, we started two instances of our operating system. Both were basically the same code, we only adjusted the IP and MAC addresses. Each instance had a simple ping task running that would periodically ping the other machine using the ARP protocol and print received packets. That this works, shows very well that our initial goal to implement a network stack up to the ARP layer was completely achieved.

All in all we have gone through the various abstraction layers in an OS and a network stack. We wrote assembler code to handle interrupts, implemented real device drivers and high level functions like `arp_send`. Additionally we created helpful library functions like `malloc` and implemented a system clock as well as a logger that shows important information more attractively. We also refined the existing interrupt handler to allow more interrupt service routines.

In following work one can use our work as a basis for implementing higher level network protocols like the IPv4 and TCP. With IPv4 and TCP one can then implement Telnet to ultimately watch Star Wars Episode IV on `towel.blinkenlights.nl`. All the necessary basics for that like the ARP protocol are already provided by our work.

9 Related Work

There are several other projects that implement drivers for PCI and NICs and even more that implement a network stack. In the following we introduce some of the projects we used as inspiration.

The most known project surely is the Linux kernel. But because it aims for full compatibility with a lot of hardware and compliance with all the standards, it is so extensive that it was very hard to understand their implementation.

Another very active project is SerenityOS. [4] It is a self-made UNIX-like operating system written in C++ that also implements drivers for PCI and the E1000 and is much smaller and better to understand than the Linux kernel.

An example for a similar, but further advanced network stack is the now inactive `netstat` project on GitHub. [2]

Furthermore there is a lab in the Operating System Engineering lecture of the Parallel & Distributed Operating Systems Group at the MIT that is very similar to our project. The students have a skeleton of an operating system that they emulate using QEMU and develop basic functionalities for this OS during the course. Lab 6 [8] provides information for implementing a driver for the E1000 and building a network stack on top of it.

References

[1] ArchLinux wiki contributors. *QEMU*. `https://wiki.archlinux.org/index.php?title=QEMU&oldid=632898`. Online; accessed 2-September-2020. 2020.

[2] J. Groocock. *netstack: TCP/IP network stack implementation in userland*. `https://github.com/frebib/netstack`. Online; accessed 19-September-2020. 2017.

[3] Intel Corporation. *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual*. `https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf`. Online; accessed 12-September-2020. 2009.

[4] A. Kling. *SerenityOS*. `http://serenityos.org/`. Online; accessed 19-September-2020. 2020.

[5] A. Limited. *PCI configuration*. `https://developer.arm.com/documentation/dui0224/i/programmer-s-reference/pci-controller/pci-configuration?lang=en`. Online; accessed 29-September-2020. 2008.

[6] OSDev Wiki contributors. *Intel Ethernet i217*. `https://wiki.osdev.org/Intel_Ethernet_i217`. Online; accessed 12-September-2020. 2020.

[7] OSDev Wiki contributors. *PCI*. `https://wiki.osdev.org/index.php?title=PCI&oldid=25084`. [Online; accessed 5-September-2020]. 2020.

[8] Parallel & Distributed Operating Systems Group, MIT. *Lab 6: Network Driver (default final project)*. `https://pdos.csail.mit.edu/6.828/2018/labs/lab6/`. Online; accessed 19-September-2020. 2018.

[9] D. C. Plummer. *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware*. STD 37. `http://www.rfc-editor.org/rfc/rfc826.txt`. RFC Editor, Nov. 1982.

[10] QEMU wiki contributors. *Main Page*. `https://wiki.qemu.org/index.php?title=Main_Page&oldid=9546`. Online; accessed 2-September-2020. 2020.

[11] Wikipedia contributors. *Internet protocol suite — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Internet_protocol_suite&oldid=974885345`. Online; accessed 2-September-2020. 2020.

[12] Wikipedia contributors. *Peripheral Component Interconnect — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Peripheral_Component_Interconnect&oldid=976234642`. Online; accessed 2-September-2020. 2020.

NinjaStorms kernel architecture
Improving a toyOS

Felix Roth and Konrad Hanff

Hasso Plattner Institute for Digital Engineering
`felix.roth@student.hpi.de`
`konrad.hanff@student.hpi.de`

We outline our progress of incremental changes to an ARM-based operating system called NinjaStorms. We describe newly implemented features such as a working syscall system, and specific syscall building upon that like `create_process` and `wait_on_pid`. We outline the initial operating systems, our goals, as well as hurdles on the way and discuss some of our implementations in comparison to other operating systems.

1 Initial Situation

We based our work on the code of the NinjaStorms operating system, which was published in a Github repository, [2] a basic operating system for the ARMv5 architecture, implemented to run on the board of a LEGO mindstorms robot and be emulated with QEMU. It is largely written in C, with some ARM assembly code inlined and in additional files. Our first steps were understanding the given project, we have outlined the initial state below.

1.1 Tasks

A task is a struct (`task_t`) that stores the 16 CPU registers and the Current Program Status Register (cpsr) of a process when the scheduler stops its execution and schedules another task. They are stored in a ring buffer that is defined in the scheduler's source. Tasks can be added during runtime with `add_task`. The tasks are executed in the user mode of the CPU by setting the corresponding bits in the saved cpsr on task struct creation.

1.2 Interrupts

A complete interrupt table is present, although we initially did not understand this section of code. Comments that we did only understand after we figured out how it worked were present but did not help initially. Only one interrupt handler is implemented, the `irq_handler`. The interrupt gets triggered periodically by a timer initialized elsewhere and is responsible for scheduling the next task.

1.3 Scheduler and Dispatcher

The scheduler logic is implemented in C, while the dispatcher, responsible for saving and loading CPU registers when necessary, is implemented in assembly which gave us trouble at first, but when implementing the later features was really no problem at all. The saving routine is executed on the periodic timer interrupt. It then starts the scheduler which changes the content of the `current_task` pointer to point to the struct of the task that should be loaded next, then jumps to the non-returning function `load_current_task_state`, which loads all saved registers including the cpsr into the CPU registers. After that, the newly scheduled tasks continues where it left off before being interrupted by the timer interrupt.

2 Goals

As the initial system was very basic, we had a lot of options to expand. The initial idea was to implement a network stack, which seemed like a daunting task. We therefore split the group in two teams, where we would be tasked with creating the fundamental architecture that would allow a network stack to be built on top of it. 1 While this was an initial motivation, we acted largely independent and set our own goals, which are outlined below.

2.1 Kernel- and Usermode

Modern general purpose operating systems use a distinction between kernel and user mode, where in many features are restricted to kernel mode. This makes an interface to the kernel necessary, if any of the kernel features are relevant for user mode programs. In Linux this is implemented with syscalls, which are user mode functions that permit processes to use kernel mode features. As we wanted to establish a clear distinction between the two fundamental operating modes, we also wanted to implement syscalls. To achieve this goal, we would have to implement a handler for software interrupts, a wrapper library as well as dispatchers on kernel mode side which then perform the appropriate actions for each specific syscall.

2.2 Processes

Tasks or Process are the fundamental units of operation in many popular operating systems. While a task struct was already used in an array of tasks, we wanted to implement many new features that would allow identification of processes and save relevant data. These goals were closely linked with the goal of syscalls. In the matter of processes, syscalls would be required to create and exit syscalls as well as gather information about the currently running or other processes.

2.2.1 Inter-process communication

With processes firmly established, we wanted to implement a way for distinct processes to communicate with each other, without breaking inter-process isolation.

2.3 Further Goals

As the operating systems can run in two different environments, the kernel code included some checks for the current running board using the C preprocessor. We found this to be a breach of abstraction layers, and wanted to push these to a lower level, a hardware abstraction layer. We also planned on going further if the time frame permitted it, with user interaction on the terminal and further implementation of the subset of library functions that were included with the operating systems.

3 Progress

Table 1: Timeline of the project in chronological order

Activity
Project selection
Start working on syscalls
Syscalls are working
Add process structure
IPC buffer
End work on fork, Start work on wait
Rewrite syscall handler in assembly
Start cleaning up and adding examples, some hardware abstraction
Implement cooperative multitasking, project finalization

3.1 Processes

After assessing the existing `task_t` struct, we noticed that some valuable information was not saved. In the initial version there was no way to exit tasks, so the position in the array of tasks was a unique identifier for every process. Since we wanted to create and exit processes during the runtime, we implemented a unique process id (pid) for every task/process. This is quite similar to the POSIX standard, implemented in Linux, where every process can be identified by a pid. Similar to that standard, in our operating system we also created an initialization process on user mode side with pid 1. With processes created by other processes, a parent structure was an obvious next step. Every process now has a parent process that is identified by pid, while the init process has itself as parent. We also saved some information that would become necessary in the implement of wait and exit, as well as stored the errno on process level, so it is predictable for each process, rather than possibly getting changed through scheduling of other processes. Inter-process communication, which requires some data in the task_t struct, is described in subsection 3.3. Tasks and processes are treated as synonyms.

3.2 Syscalls

Our implementation of syscalls was very much inspired by the standard approach, found in many textbooks. Since processes already run in user mode, we needed a mechanism to switch to kernel mode (called svc mode in the ARM architecture we worked with). Since this is already an effect of an interrupt, we just needed to generate an interrupt and get the necessary information about what we want to call in the kernel and with arguments should be used through.

ARM provides the `SVC{cond} #imm` instruction, that stands for supervisor call, which copies the cpsr into the spsr (Saved Program Status Register), switches the CPU to svc mode and jumps to the address of the interrupt vector table plus 8. It also masks interrupts so the execution of a syscall can not be interrupted by another interrupt, e.g. the scheduling of another process.

The `svc` instruction takes an immediate argument that remains in the machine code and could theoretically be retrieved, but we don't use this feature as it would require us to build the instruction at runtime, which could get messy. Instead we use the general purpose registers `r0` and `r1` to send the number of the syscall and a pointer to its argument. They are set in usermode, which then executes `svc #0`, and read in kernel mode.

As stated above, when `svc #0` is executed, the CPU jumps to the address of the interrupt vector table plus 8. There we place an instruction that loads the program counter with the address of the `syscall_handler`, the entrypoint for syscalls in the kernel.

The `syscall_handler` is an assembly function that stores the CPU registers and cpsr in the task struct, calls the `syscall_dispatcher(uint32_t syscall_number, void *data)`, passing on the values for syscall number and the pointer to the arguments it received in `r0` and `r1`.

The `syscall_dispatcher` calls the correct kernel mode function for the given syscall number with the argument. That function executes the actual syscall and returns an integer when it is finished, that may include a pid or a status number. The `syscall_dispatcher` also returns the same value.

The `syscall_handler` then either returns to the user mode process with the integer result or loads a new process, depending on the type of syscall. More on that in paragraph 3.2.3.

In the user mode, there are generally two layers of abstraction around the svc instruction. There is a syscall(unsigned int number, void *data) function that moves its arguments to the registers r0 and r1, calls svc #0 and returns the result for abstraction from the assembly instructions to execute the supervisor call. The second layer is a syscall-specific wrapper function for syscall(number, &data), it provides the correct syscall number and data argument, generally a struct containing all necessary information. The syscall function as well as the specific functions for each use case are stored in a user mode library.

3.2.1 create_process

The most fundamental and obvious use of a syscall is to create a new process from within the current running process. Therefore we adapted the existing add_task() function to be accessible from user mode. create_process, just as its predecessor, requires the passing of a function pointer that determines the start of the execution of the process, this is being passed through a struct with the syscall.

The dispatcher for create_process searches for an empty space in the tasks array, assigns a new process id by incrementing the previously assigned process id, and creates a task_t struct that represents the process on kernel side. In this struct, the pc is then set to the passed address, the process is assigned a stack (an address space) and set to the user operating mode by appropriately setting the cpsr. The process is scheduled as soon as possible, as it is initialized in the TASK_RUNNING state.

In Windows NT one would create a process with a function of the CreateProcess family with a lot of arguments. Our system is more closely modeled after Linux, so this implementation was not a consideration, even though some similarities are shared, not only in name. The standard way of creating a new process in Linux is the use of the fork/exec pattern, which is described in the next section. Our implementation can more directly be compared to posix_spawn, even though, again, this call allows for much greater flexibility with arguments.

3.2.2 fork and exec

In Linux, as well as the POSIX standard, the syscalls fork and exec can be used to create new processes from a process and then selecting a new program for execution. This pattern has long been established and has some advantages over a seemingly more direct approach with create_process, especially in that the creating process has total control over parameters of the program execution through sequential additional instructions between fork and exec, where as all parameters would have to be supplied in a create_process, as that call immediately results in the execution of the new process.

At this point we didn't require additional parameters for execution, so that was no pressing concern. That's why there seemed to be little reason to implement fork in a system that already had a create_process equivalent to use it in conjunction with exec. Furthermore, some additional problems with that pattern were evident. [1] There are other ways to use fork, for example in combination with wait, that did warrant implementation.

Even though the use cases may be limited, we wanted to implement `fork` to increase similarity to Linux, and thereby familiarity to users. As pages were not used in the operating system at that point, we could not simply copy pages (or mark pages as copy on write) for the new forked process. Instead we had to copy the stack of the process. This was however not possible, as the stack contained not only relative addresses, which we could copy easily, but also absolute addresses. For these addresses, we would have to determine if they referenced process external data or data in the process stack. The latter would have to be translated into new addresses. This was a challenge we did not solve.

The implementation of `exec` was not possible for the simple reason that no file system is implemented that would supply the file to execute.

3.2.3 exit

Since we can create processes, we also want a way to exit a process, returning a result. This is done by the `exit(int32_t result)` syscall. It marks the task as `TASK_DONE` so it does not get scheduled anymore and removes it altogether if possible (No task waiting 3.2.4). For further convenience, we set up each process on creation so that it calls `exit` automatically when returning if it does not exit itself.

Due to the nature of `exit`, in that it does not return to the calling process, we need to schedule and load another process during the syscall. This proved problematic, since when we called the `load_current_task_state` function responsible for this from somewhere in our kernel code, all stack frames of function call leading up to this would remain on the supervisor mode stack while the execution continues in user mode. The next syscall would be executed with all the frames still on the stack and on each exit the stack would grow even more. This was a memory leak we knew of but didn't know how to prevent.

Later investigation shows that the Linux kernel gets around this problem by having a dedicated kernel mode stack for each process[3] that does not interfere with syscalls from other processes and can be cleared when a process ends. This approach also works better than the one kernel mode stack for everything when running concurrent or parallel processes, because multiple processes can execute syscalls simultaneously.

Return to user mode This memory leak existed quite a long time until, during the finalization of the `wait_on_pid` syscall, we found a solution. The solution is to not load a new process during the execution of a syscall but to only declare the wish to do so and then fulfill this request when no harm can be done. This is when the control flow of a syscall returns to the `syscall_handler` which formerly would have returned to user mode. During an instance where only the frame of the current `syscall_handler` function is on the stack (in comparison to before the syscall, there are still other frames), a variable is checked to decide whether the control flow should return to the calling process as usual, or the process pointed to by the `current_task` pointer should be loaded. The syscall can change the contents of the pointer as well as change the variable indicating a new process should be loaded or the `syscall_handler` should return to user mode in the calling process.

Since we also remove all remains of the current frame from the stack, there is nothing unwanted left on the stack when execution in user mode continues, even if it is in another process.

3.2.4 wait_on_pid

Since we could not implement fork/exec/wait, the next best thing that was possible is creating a process using `create_process` and waiting until it exited with a result.

This is implemented as follows. When a process p_1 creates another process p_2 it receives the *pid* of p_2. When it calls the `wait_on_pid(pid)` syscall, p_1 is marked as waiting on p_2 in the task struct. When the scheduler looks at p_1 for possible scheduling, it also checks if p_2 has exited already. If not, p_1 will not be scheduled, but if true, the exit result of p_2 is placed in `r0` of the saved state of p_1 and p_1 is loaded, therefor it seems to the syscall as if `syscall_handler` had returned the result. The process can then use the result.

Also, when a process in the `TASK_DONE` state is considered by the scheduler, it checks if any other process is still waiting on it. If not, it is removed, thereby freeing space in the limited tasks array.

3.2.5 Cooperative Multitasking

Since our OS can manage multiple processes, we must consider how they share the limited resources of the CPU. Since the system is designed for one CPU core, it is not possible to truly run processes in parallel. Instead, we could employ batch processing, where each program runs until it exits before another one starts to run. In our mind, this is not really useful, especially since we have at least two other options: Cooperative and preemptive scheduling.

With cooperative scheduling, processes can give control (yield) to the next process. This requires the program and therefore the programmer to actively end its current time slice.

Another option is preemptive scheduling, where programs do not contain code to yield control but the OS interrupts the running process, stores its state and replaces it with another one after a specified amount of time. This process is transparent to the processes.

In contrast to cooperative multitasking, preemption does not need the processes to be well-behaved and yield control regularly. Even if a process enters an infinite loop without giving up control, other processes will continue to run, because the OS simply interrupts after a short period of time. Also, it requires less thought from the programmer, who does not need to decide where in e.g. a long calculation the execution should temporarily interrupted, especially with a complex control flow. Preemption is also useful for interactive systems reacting to input, like the LEGO Mindstorms controller NinjaStorms was originally intended for. Depending on the length of each time slice, processes handling input can run and process input without long delays.

While preemptive scheduling is useful in a context where processes are constantly busy and starvation is not acceptable, cooperative scheduling brings the benefit of task having very little work being able to yield control if they don't need the full time slice preemption would provide. This can improve general performance of the system.

Using the return to user mode/load new process functionality, it is easy to add cooperative multitasking to the OS. A process can choose to call the syscall `yield()` and end its current time slice early. The scheduler then runs and selects a new process, also resetting the timer for the next scheduling interrupt, to guarantee the next process gets a full time slice. If this was not done, a process could always use nearly its full time, then yield, and the next process would always have very few instructions to run.

We added the ability to disable preemption when starting the scheduler, so the OS now supports full cooperative or preemptive multitasking, as well as preemptive multitasking with the possibility to yield control early.

3.3 Inter-process communication

While we could not implement memory isolation between processes as we lacked paging or segmentation, we did create ways for processes to interact and communicate with one another in isolation conforming ways. The `create_process/ return/ wait_on_pid` pattern described earlier is one way of inter-process communication. Here we describe two additional ways.

3.3.1 Buffer

Every process has some fields that can be used for inter-process communication. The general behavior is as follows: A process with pid t marks its IPC buffer as open by calling `ipc_buffer_open(0)`. Another process can then send data d to the process, in the size of words (4 bytes) $d[i]$, by calling `ipc_buffer_send(d[i], t)`. This word $d[i]$ is then saved in the IPC buffer of process t, meaning it is saved in the tasks array in the kernel. Process t can then retrieve the data at any later point, by calling `ipc_buffer_read()`, which will return one word and remove it from the buffer. t may also check the length of the buffer with a syscall. All these syscalls make use of appropriate error numbers to communicate what went wrong, failing for example when the buffer is full, empty or closed.

This implementation is very simple, and has its problems. Buffer size can not be altered, as the buffer is directly in the tasks array. This could be solved by saving an address to a growing buffer, which could be easily implemented with pages and dynamic memory. Furthermore the receiving process can not restrict access to specific processes and can not determine the sender.

We took some inspiration from other inter-process communication tools from UNIX or UNIX-like operating systems. *Pipes* also use kernel buffers to transfer data between processes. File descriptors were not implemented in the current version, but could be a logical next step to take this implementation. UNIX sockets make use of a file system, which is not provided in our operating systems and were thus not an option. In the long term, sockets would be a good idea, since we strive to create a networking operating system, which may well use a socket implementation when communicating via TCP.

3.3.2 Security in shared memory with mutexes

With processes defined as functions in code, a programmer may include variables and data outside of functions, that all functions can access. Thereby processes may interact by altering these variables, which are stored at fixed position by the compiler. To facilitate a fruitful communication between multiple processes writing or reading on the same data, we implemented mutexes. To use these, such a shared data as described has to be defined and referred to by all participating processes. This variable *m* can be used as the mutex. This common variable should not be used to store data, but to synchronize processes. A process can then call `lock_mutex(&m)`, to acquire the mutex. If the mutex has not been acquired and not freed before, the process will busy wait until the mutex is freed with `unlock_mutex(&m)` by another process. By using spinlocks, both mutex calls are handled in user mode. As NinjaStorms uses the ARMv5 instruction set, the use of `LDREX` and `STREX` is not available. Therefore, the mutex is implemented with the `SWP` instruction.

This implementation is once again quite simple, but not perfect. Busy waiting is something that should be avoided, and a better implementation may be possible with the use of syscalls, with behavior like the current `wait_on_pid` call.

4 Conclusion

During development of new features we focused on getting things running and did not pay much attention our solution's impacts on performance. Currently there are no critical and noticeable performance bottlenecks we are aware of. We did not use the OS for time critical applications but there could be problems in the future or even when introducing a large number of simultaneous processes, since the scheduler we implemented has linear worst case time complexity in the current number of tasks.

It always considers all processes, not just the currently running ones and for a process that has already exited, all other processes are checked if they are waiting for it. Also reparenting children of terminated processes takes linear time in the number of processes. Using other data structures than a one-dimensional array of processes structs to hold all relevant information could improve scheduling performance drastically.

4.1 Next steps

We mostly operated independently and look forward to merging our progress with that of the other two groups that worked on other aspects of the same operating system. With the addition of dynamic memory and paging some major improvements could be made, for example an implementation of `fork`, managed shared memory for processes to communicate and real memory isolation between processes. With the advancements of the network development group (See page 49), new syscalls can be implemented, as a timer is implemented, we may be able to implement a `sleep` call that pauses execution for a set amount of time.

Many modules of our system are not designed to be especially efficient, and can be improved. Examples are the inter process communication, that may be improved with file descriptors or an implementation of semaphores in addition to mutexes. Aside from that, the scheduler may as well be improved to understand priorities or threads added as the scheduling unit instead of processes. A major step would be user interaction on a terminal.

References

[1] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. "A Fork() in the Road". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, pages 14–22. ISBN: 9781450367271. DOI: `10.1145/3317550.3321435`.

[2] A. Grapentin, D. Korsch, C. Werling, and C. Walther. *ninjastorms*. `https://github.com/ninjastorms/ninjastorms/`. Online; Accessed 24-September-2020. 2020.

[3] R. Love. In: *Linux Kernel Development*. Addison-Wesley Professional; 3rd Edition, 2010, page 21. ISBN: 978-0672329463.

Tool based analysis of the Windows Research Kernel

Using static analysis and mapping of CVEs to source

Niklas Schilli

Hasso Plattner Institute for Digital Engineering
niklas.schilli@student.hpi.de

The Windows Research Kernel (WRK) mirrors the kernel used in Windows Server 2003 and XP SP1. In this paper we aim to apply modern static analysis techniques to evaluate the usefulness of user-space focused analyzers with the kernel as it was when the WRK was released by using the PVS-Studio C analyzer. Furthermore we try to map Common Vulnerabilities and Exposures (CVEs) on to the WRK using PVS-Studio (PVS), a user space based C and C++ analyzer. We examine CVE-2005-2827 in detail, as the entire defect chain is located inside of the WRK.

1 Introduction

The released WRK source code includes code for processes, threads, virtual memory management, scheduling and other core NTOS functionality. It was originally licensed for academia, which enabled swaths of universities to use the material for teaching. While the source code is by now widely available even outside of academia, the challenge still remains of injecting modern analyzers in the by now dated toolchain. Analyzing the WRK with a user-space analyzer requires a lot of legwork. While the WRK was itself originally analyzed by Microsoft PreFAST, the tool has since long been discontinued and does not work with newer toolchains. Also, the choice of PVS as analyzer precludes the use of many of its features, mainly those involving memory management analysis. As such, only a subset of relevant results can be considered relevant from PVS.

1.1 Contributions

I would like to thank Andreas Grapentin and Andreas Polze for allowing me to investigate this during the 2020 Operating Systems II course, even though it was not a pre-approved topic. Furthermore I would like to thank Raymond Chan for explaining the intricacies of the 16-Bit Windows remnants of CS_CLASSDC and CS_OWNDC. His blog, *The Old New Thing*[1] is an invaluable resource for people looking into insights of Windows development and historical information on remnants of older Windows versions.

[1] devblogs.microsoft.com/oldnewthing.

2 Context

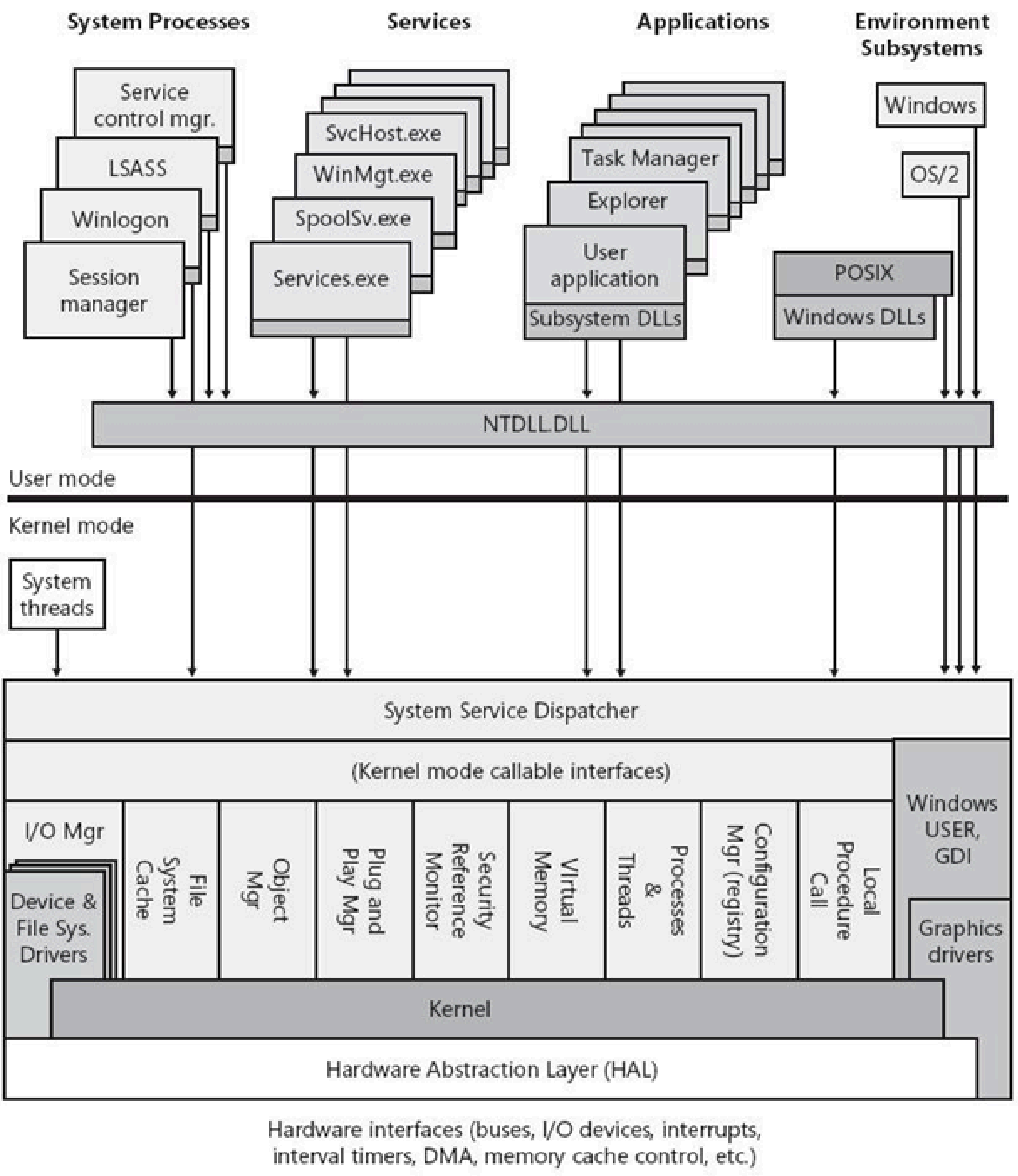


Figure 1: Windows Kernel - Source https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-components

The WRK does not include all features listed in shown Diagram. Features such as the Plug and Player Manager , Power Manager, the Device Verifier and the Virtual DOS Machine. Furthermore there are some components which come precompiled, so static analysis has far from the complete picture when applied to the WRK. As such, any CVE mapping may easily cross available source boundaries.

2.1 Background

PVS is a static analyzer first released in 2006 which offers support for multiple programming languages. Their C analyzer is focused on user space programs which is expressed by their reliance on malloc and free for memory management routines, and unfortunately alternative memory management routines can not be substituted for analysis. Other analyzers support this functionality allowing for custom memory management functions to be considered. CLang even created its own tool with only memory management analysis features, MallocChecker.

3 Problem

Modern analyzers do not integrate well with batch based compilation. Solution formats have changed over the years, the makefile does not offer a well formed entry point to include static analysis down the line. Additionally the actual tools are so out of date that one can be happy to find a accurate documentation for a at best twenty year long maintained tool in an ancient version. toolset used for the WRK either requires diligent setup on Windows 10 machines or getting a Windows XP VM with tools which have been deprecated for 15 years, such as Visual Studio 2003. Another concrete problem arising from the CVE mapping is that the WRK at best mirrors the windows kernel at the 12th July 2006, the date where the license file was last modified. As such, some exploits may have already been fixed in the WRK source which is hard to verify due to the fact that the exploits do not include the windows source code at the time of their writing.

3.1 Specific Problem

PVS results come in three severity degrees: advice, warning and critical. The advice results are largely in line with pedantic compiler warnings, such as comparing a signed with an unsigned value (even when the comparing value is a constant) or branch simplification where two consecutive checks against the same variable may be placed in an outermost if block. They mostly do not offer a lot of insight, and as such are not the primary target of this investigation. Warning and critical results in this application refer mostly to missing parameter validation, an unfortunate byproduct of not having control flow analysis which can trace argument validation to the callsite. These cases are all annotated with comments describing that the parameter will not be verified as the caller has already done that.

4 Solution

Modern toolchains for analysis are invaluable and are used at Microsoft to affect static analysis. PreFAST, the long discontinued tool has been absorbed into the

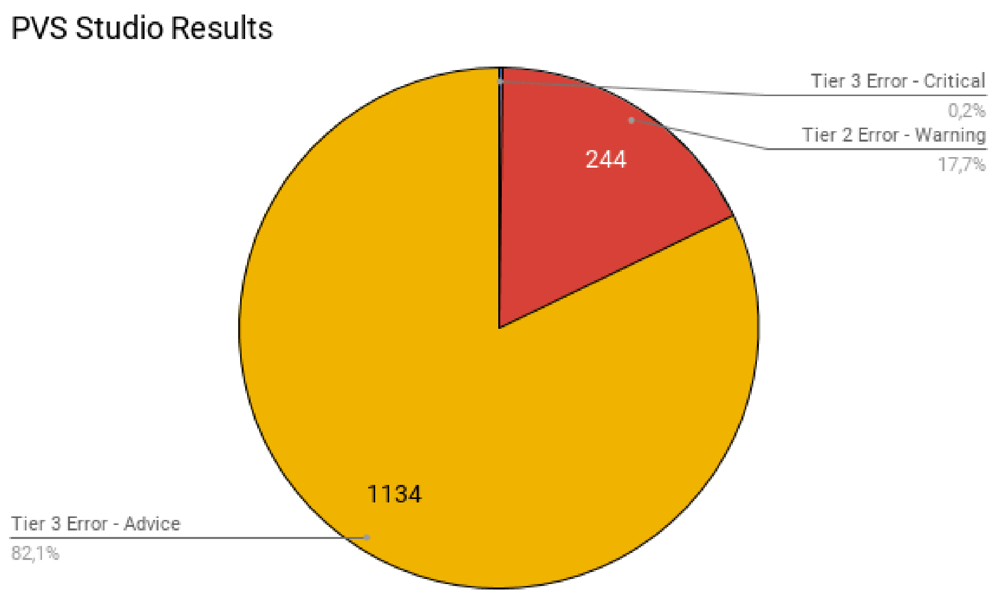


Figure 2: Number of errors found by PVS Studio.

Microsoft Visual C compiler (MSVC) which in its newest version (MSVC 2019) offers a plethora of diagnostics which are unavailable in the 2003 version of the product. Furthermore, kernel analysis on e.g. the Linux kernel does suffer fewer of the drawbacks, as the source code is completely available and modern enough to be compilable with current toolchains. As that is not easily applied to our use case, we decided on writing manual helpers which try to extract comment validation from the source and match them with the PVS results. As for the mapping of CVEs to the source code we also employed a helper to classify the defect origin containing component to ease the workload for finding an exploit present in the WRK source code.

4.1 Specific Solution

During analysis of the results we decided on finding variations of commonly used comments to get an estimate for the amount of PVS results which can be explained by comments in the source code. Ultimately we settled on the word list in Annex A, which definitely does not include all comments which explain missing validations. Nevertheless the small word list generated 150 different matches in the .c files, which often coincided with PVS results.

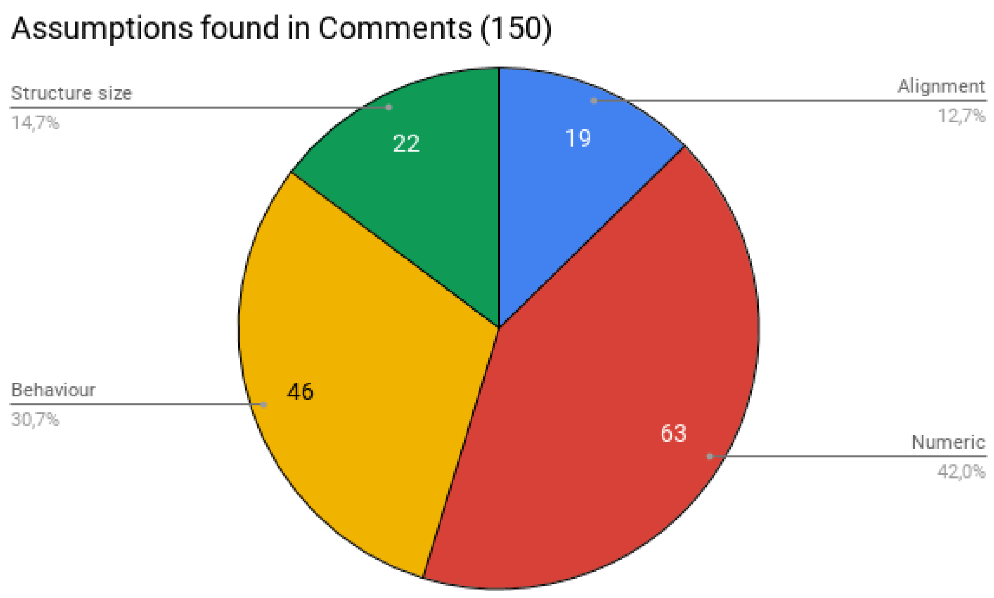


Figure 3: Kinds of assumptions expressed in comments.

5 Implementation

The WRK analysis tool checks all .c files and checks line by line whether they contain a common comment annotation substring, i.e. an expression which occurs in different places as a comment explaining the missing validation. This approach obviously is far from ideal, as it currently is suffering from a bad hit rate on varying expressions due to missing a stopword filter, stemmer and the like. The only normalization done is transforming all text to the lower character variants to elide capitalization errors. As this was not the focus of this project the drawbacks are acceptable for us. The second analysis tool checks the tables downloaded from the CVE Details website[2] for component names such as 'GDI', '.NET' etc. to reduce the amount of exploits to be checked before we find one where all components are present in the WRK source code. Of the 500 exploit descriptions checked this way, 306 contained one term listed in Annex B.

In the resulting set of exploits we discovered CVE-2005-2827, a local privilege escalation exploit which resides inside the thread termination routine. The entire infection chain resides inside of source code available in the WRK.

[2]`https://www.cvedetails.com/vulnerability-list/vendor_id-26/product_id-739/Microsoft-Windows-Xp.html`.

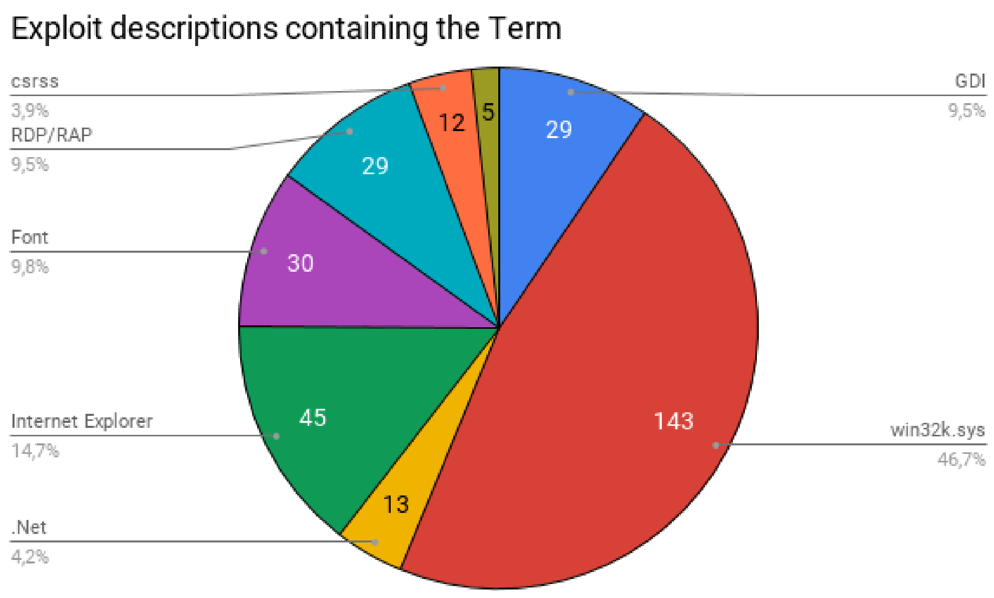


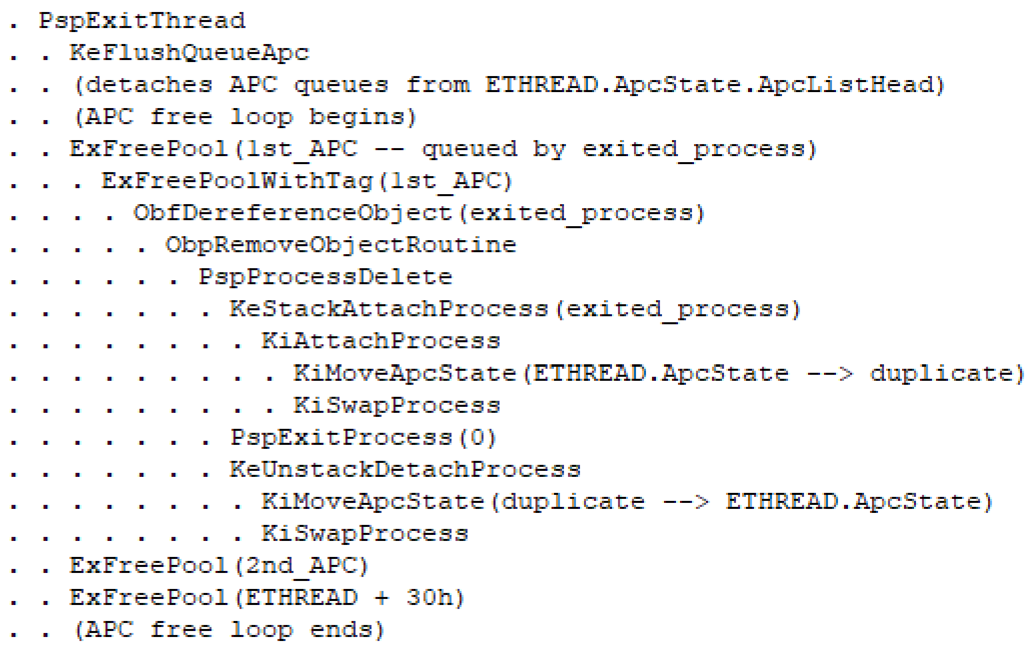
Figure 4: Sources of exploits found by the analysis.

5.1 Minor Detail

The vulnerability origin is located inside PsPExitThread's APC freeing loop as well as in the behaviour of KiMoveApcState, which is in the call stack downstream from PsPExitThread. When a thread is in the process of exiting, PsPExitThread will detach the thread's APC queues from the first two elements in the EThread.ApcState.ApcListHead list, so that each queue is now a circular doubly-linked list. Here the defect occurs, as it is intended to free the pointer supplied by the kernel. If the exiting thread is the last thread associated with a process, PsPExitProcess may be called during the apc cleanup routines, which in turn reconnects the previously disconnected first list elements. The callstack is displayed in Figure 5 for brevity, as the actual code spans several hundred lines.

6 Evaluation

PVS would not have been able to find this vulnerability due to not working well with inline assembler in C source code. Furthermore, it is an intricate exploit which requires knowledge of control flow and entries to Windows data structures which PVS does not have. A kernel based static analyzer would not have caught this exploit either due to the same reasons. Nevertheless the choice of PVS as analyzer was definitely misguided.



```
. PspExitThread
. . KeFlushQueueApc
. . (detaches APC queues from ETHREAD.ApcState.ApcListHead)
. . (APC free loop begins)
. . ExFreePool(1st_APC -- queued by exited_process)
. . . ExFreePoolWithTag(1st_APC)
. . . . ObfDereferenceObject(exited_process)
. . . . . ObpRemoveObjectRoutine
. . . . . . PspProcessDelete
. . . . . . . KeStackAttachProcess(exited_process)
. . . . . . . . KiAttachProcess
. . . . . . . . . KiMoveApcState(ETHREAD.ApcState --> duplicate)
. . . . . . . . . KiSwapProcess
. . . . . . . PspExitProcess(0)
. . . . . . . KeUnstackDetachProcess
. . . . . . . . KiMoveApcState(duplicate --> ETHREAD.ApcState)
. . . . . . . . KiSwapProcess
. . ExFreePool(2nd_APC)
. . ExFreePool(ETHREAD + 30h)
. . (APC free loop ends)
```

Figure 5: Call stack of the CVE-2005-2827 vulnerability.

6.1 Threats to Validity

The choice of PVS was a blunder, a kernel focused analyzer would most probably provide better insights. Also, as we did not have the entire kernel source code available our analysis was obviously incomplete. A more thorough approach could be taken by trying to obtain PreFAST and use this to facilitate better understanding of the windows kernel as it was used during that time at Microsoft for the windows kernel. Ultimately, better analysis tools could produce better results but as far as the analysis with PVS went it was sufficient for some insights. Furthermore, due to the recent leak of Windows source code spanning from 3.1 to Windows 7 CE one could probably apply more modern techniques to the newer source code. This might result in clearer insights as well as a more uncurated view on the source code, as the WRK source was explicitly polished before the release.

7 Related Work

Fuzzing is an integral part in kernel hardening, as projects such as kAFL or Google's Syzcaller have demonstrated. Fuzzing is an automated analysis technique where the system under test (SUT) is provided with unexpected or invalid data input. If the SUT responds in an unintended manner a bug may have been found. The effectiveness of this technique has been proven time and time again by revelations such as Shellshock or more recently in 2018 when the existence of a reduced instruction set (RISC) was discovered which allowed circumventing all processor privilege checks. Research into kernel hardening and security has lead to scores of new security features for the kernel, such as Kernel Address Protection, PatchGuard, Boot Configuration Data or most recently, Kernel Data Protection by means of virtualization based security. These features all work to prevent unauthorized kernel data modification, but as complexity rises new attack vectors still continue to appear regularly. The static driver verifier (SDV) deserves a special mention here as well. It is the must-use tool for Windows kernel driver development, offering intricate knowledge of the Windows driver kit (WDK) functions and their use and even finding dozens of bugs in the official WDK samples.

8 Conclusion

Using a user space analyzer for kernel space code comes with several drawbacks which seriously limit the ability to glean useful insights into the SUT. All memory management based warnings are useless, as malloc/free are not present but rather kernel specific allocation routines. Furthermore due to not being able to trace callsites argument validation bugs was a big false positive source as well. In addition to this, we also have an even more limited analysis as all functions containing inline assembler are not processed either. In conclusion, while the analysis did provide some results, they were often annotated by comments in the source code. In addition to the reduced result set due to earlier mentioned shortcomings the results were not as meaningful as hoped. This does not reflect on other analyzers, as a kernel focused analyzer will probably provide better results. A modern compiler will also emit more warnings and suggestions than the MSVC 2003 compiler, which would be valuable on its own.

9 Annex A – Comments analyzed

must be 0
must be <
must be smaller than
must be >
must be greater than
must be zero
must be non-zero
must be aligned
must fit within
at least as large
is expected to
is required to
must be sizeof

10 Annex B – Exploit description terms

gdi
win32k.sys
.net
internet explorer
bluetooth
font
rdp
rap
csrss
directx

11 Annex C – Abbreviations

PVS - PVS-Studio, a static analysis tool
WRK - The Windows Research Kernel
WDK - Windows Driver Kit
RISC - Reduced instruction set

Performing Deep Packet Inspection in User Space
Implement a nf_queue backend to filter packet payloads in user space

Leonard Seibold

Hasso Plattner Institute for Digital Engineering
`leonard.seibold@student.hpi.de`

While there are already many tools for rule based filtering of packets available, most prominently IPTABLES, those tools are usually limited to packet header information and do not allow for a more in-depth analysis of the application layer payload. This may suffice in most use-cases, deep packet inspection however can allow for much more advanced intrusion detection and traffic shaping. The author has investigated ways to implement deep packet inspection on Linux, how the process can be moved into user space and demonstrated a way how user space packet filters can be orchestrated using UNIX pipes.

1 Introduction

This reports represents the development process, results and changes in the original idea of my project "Deep Packet Inspection" I worked on for the Operating Systems II lecture at the Hasso Plattner Institute. The goal of the project originally was to perform "Deep Packet Inspection" on a Linux based operating system, however, new ideas started to emerge while learning more about the architectures and APIs used in the Linux kernel.

Deep packet inspection is generally understood to be the process of analyzing packet payload data and taking actions according to that data if wanted. Actions may include dropping the packet, traffic shaping, network address translation or even manipulating the payload before the packet gets passed on. Typical firewalls, for example the well-known iptables, usually only filter by information found in the internet and transport layer headers of packets, such as source and target IP addresses, TCP/UDP ports and so on. This suffices in most situations, as it allows for network address translations and traffic shaping operations, securing servers by allowing only specific source IP addresses and so on. There are however many use cases for which a firewall that is capable of deep packet inspection may provide significant benefits. For example by analyzing the application layer protocol, much more advanced intrusion detections and DDoS protections are possible. Attackers often abuse bugs in serverside applications by sending data the protocol-implementation cannot handle properly and thereby causing a much higher workload than just sending packets that get dropped instantly. Other applications of the technique such as load balancing are conceivable as well. The scope of this project is however not to find and implement these applications but merely to explore ways how deep packet inspection can be implemented on a Linux based operating system.

2 Development Process

2.1 Creating a Loadable Kernel Module

The fist step was to understand how package filtering is typically done in the Linux kernel. After a bit of research, it quickly became apparent that a kernel module would be needed to perform deep packet inspection. As I did not have any experience with Linux development, my first goal was to create a loadable kernel module (LKM). As the build process is not trivial, I decided set up the project repository with CMake. This ensured builds are reproducible, source files could be added easily without manually updating a Makefile and it would be easy to later add new modules to the project, for example a user space library to interact with the kernel module.

2.2 The Netfilter API

After getting a simple kernel module to load, the next step was to understand how I could hook into the packet traversal in the Linux kernel. Since version 2.4, Linux includes the so called Netfilter API. [6] This API allows Kernel developers to register synchronous hooks at various stages of the packet traversal (e.g. at pre routing, forwarding, input etc.). [4] Those stages happen to match IPTABLES' filter chains, as the IPTABLES backend that actually filters packets is in fact implemented using the same Netfilter API. Those hooks then receive a pointer to a sk_buff (the data structure that represents packets in the Linux kernel) and can decide whether to accept, drop or queue the packet (see Listing 1 for a simple example).

Listing 1: Registering a hook to the Netfilter Framework

```c
static struct nf_hook_ops *nfho = NULL;

unsigned int hook_func(void *priv
                        struct sk_buff *skb
                        const struct nf_hook_state *state)
{
    // analyze packet data and cast a verdict
    return NF_ACCEPT;
}

static int __init LKM_init(void)
{
    nfho = (struct nf_hook_ops *)
                kcalloc(1, sizeof(struct nf_hook_ops), GFP_KERNEL);
    nfho->hook = (nf_hookfn*) hook_func;
    nfho->hooknum = NF_INET_PRE_ROUTING;
    nfho->pf = PF_INET;
    nfho->priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, nfho);
}

module_init(LKM_init);
```

This approach worked fairly well. The hook function can access the full packet payload, interpret it and take actions accordingly. It can even manipulate packet data and therefore even advanced operations such as traffic shaping and network address translation are possible.

2.3 Moving the Filter to User Space

It is however quite inconvenient to write a Kernel module for every filter we want to apply. At this point, I decided to try and move the actual filter process to user space. As the Linux kernel does not have direct upcalling capabilities (there is no way to just call a function in user space from kernel space without some sort of inter process communication), I therefore needed to explore different ways of how to communicate with a user space process. [3]

My first idea was to use a pseudo filesystem. Pseudo filesystems are commonly used to expose kernel space information to user space and to provide an interface for configuration of Kernel (module) features at runtime. This is done via files. These do however not represent information written to physical disk, but rather constitute inter process communication with the kernel. Many examples for this technique can be found in the Linux kernel, for example the process pseudo filesystem that exposes information about running processes, or the devices pseudo filesystem that contains raw interfaces to physical (and virtual) devices of all kind. After getting a simple sysfs based interface to work, I quickly discarded this first approach, as the read and write operations are not stream based. The API simply provides a char buffer that can be read from or written to. While there are ways to implement stream based operations (e.g. via a character device driver), other inter process communication techniques seemed much more appropriate for the stream oriented nature of packet traversal.

2.4 IPC via Sockets

My next idea was to use socket based inter process communication to transmit packet data and receive verdicts from a user space process. I decided to go for a UNIX domain socket. The specification allows us to use the filesystem as address space. This means, a file will be created when launching the socket that represents it to user space. One advantage if this is that I did not need to implement any authentication or access control myself, I could just set the file access modes appropriately, for example to only allow user space processes running under the root user to access the socket.

As the socket API is not well documented and has some important differences to the user space variant, it took me a while to get basic communication to work. I was then able to send packets to a user space client. I then noticed a problem with the approach: to receive the verdict from user space, the thread receiving the data needs to wait. Normally this is done by scheduling the thread until a client writes to the connection. This does however not work in the hook function's context as the kernel is atomic at that time. Trying to wait anyway causes a kernel panic. While it would technically be possible to use a busywait loop that continuously checks whether new data has been written to the connection, this is not advisable as blocking an atomic kernel thread causes further problems. These kernel threads wont be scheduled automatically so a busywait can ultimately cause the system to become completely unresponsive. This was a big problem, as there is no easy workaround.

After many hours of research, I learned about another feature of the Netfilter framework. Netfilter hooks not only can decide whether to drop or accept a packet, they can also make the framework queue it. This means, that the packet will not continue in packet traversal, but its memory resources will also not be freed. It can then later be reinjected into packet traversal using another API function. While trying to learn more about this part of the API, I noticed that it is already used by another in-source kernel module called the nfnetlink subsystem. [7] This subsystem implements essentially exactly what I was trying to achieve: it provides an IPC interface using a netlink socket (which works fairly similar to UNIX domain sockets) for user space packet filtering. Due to the basically non-present documentation, I was however not able to get it to work as promised. As this is also a learning project and the socket communication part was already implemented, I therefore decided to go ahead with my own implementation.

2.5 Implementing a Queue Handler

The Netfilter API allows us to register queue handlers similar to the registration of packet hooks. This queue handler gets called whenever a hook commands the framework to queue a packet. The queue handler is however allowed to schedule at any time. This allowed me to keep track of meta information needed to reinject the packet that was sent to user space in a thread safe manner using mutex locks. After many debug sessions (I ran in many smaller multithreading related problems such as lost wakeups due to unsafe locking, etc.) I eventually got this approach to work and was able to to filter packets in user space. I also wrote a small user space library to easily pull packets from the kernel without the need to manually handle the socket connection.

2.6 Orchestrating User Space Packet Filter with Pipes

While presenting my progress in the BS2 Sessions another idea arose. It would be nice to be able to orchestrate multiple small filter programs together instead of one single complex filter program. A UNIX centric approach that uses already well known concepts seemed like a sensible solution: combining multiple filter programs with FIFO pipes. I therefore implemented a simple user space program that simply writes the packets it receives from the kernel module via the socket connection to a FIFO pipe and reads the packets and verdicts to push back to the kernel module from another FIFO pipe. I could then implement small filter programs that read packet and verdict information via the standard input and write packet data and their verdict to standard output instead of using the socket connection directly. This allowed me to simply chain them together via shell pipe operators. I also extended the user space library with functions to wrap the read and write process and provide data structures for the packet data and verdict. A simple packet filter that uses this framework can now be implemented as simple as seen in Listing 2.

Listing 2: Filtering packets in user space using the pipe based approach

```c
#include "dpi.h"
#include <stdlib.h>
#include <unistd.h>

int main()
{
    p_buff packet;
    unsigned char data[MAX_BUF_SIZE];
    unsigned int verdict;

    while (1) {
        read_packet(STDIN_FILENO, &packet, data, &verdict);
        /*
          analyze packet.data, eventually set the verdict,
          e.g. to DPI_DROP
        */
        write_packet(STDOUT_FILENO, &packet, verdict);
    }

    return 0;
}
```

The pipe handler also pushes back the packet data to the kernel module, so it is also possible to manipulate the packet payload. This means a user space packet filter has essentially the same capabilities as a Netfilter hook based packet filter in kernel space and can even implement advanced things like network address translation.

This concludes the development process of this project.

3 Analysis

3.1 Limits

While this approach for user space filter programs is quite powerful regarding its packet filtering and manipulation capabilities, there is a big downside to it: performance. Sending packets to user space via sockets comes with a lot of overhead. Multiple threads are involved, those threads regularly need to wait for locks to ensure thread safety. This introduces many context switches that need to happen before a packet continues traversal. The packet data also gets copied around in memory quite a bit as kernel memory is not directly shared with the receiving side of the socket connection (this would obviously constitute a big security risk and violate the kernel space isolation). On the client side it does not look better: all read and write operations regarding the socket are implemented with system calls which means that receiving a single packet with meta information in user space needs multiple software interrupts. Adding the pipe based orchestration technique makes this even worse as this is the same for read and write operations on pipes.

This analysis makes it obvious that the process comes with a probably quite significant performance impact, even though I was not yet able to properly measure this impact due to time constraints for this project.

3.2 Related Work and Alternatives

The most important alternative to mention is of course the existing nfnetlink subsystem and the corresponding user space library (called LIBNFNETLINK_QUEUE) of the Netfilter project. The approach is extremely similar to mine, the biggest difference essentially being the use of a different socket type. The nfnetlink subsystem also implements more sophisticated things such as per network namespace queues.

Another approach that deserves to be mentioned are extended Berkeley Packet Filters, or short eBPF. [1, 2] Newer Linux kernels (from 3.15) include a in-kernel virtual machine. This machine can execute eBPF programs (compiled to a bytecode format) at various trace points in the kernel. This allows for packet filters that are safe (a crash does not affect the kernel as the programs runs in a virtual machine) and much faster [5] than my approach (the filter program gets executed directly in kernel space without the need to communicate with a user space process). While eBPF has many advantages, especially regarding the performance impact, it also has its own limitations. For example to be able to guarantee that a program terminates, the eBPF virtual machine does not allow instructions to jump back in the execution, so while writing eBPF programs, only loops that can be unrolled at compilation time can be used. [5] eBPF programs also have a size limit. [5] While it possible to work around this by attaching multiple eBPF programs that get executed in succession, these limitations in combination with the fact that user space libraries cannot be used make it significantly harder to implement packet filters, especially those that parse application layer protocols.

3.3 Next Steps

The first thing needed is to make the implementation more stable. At its current state the repository is highly unstable. There are several memory leaks present in the kernel module. Also, it does not properly clean up, e.g. the file the socket gets bound to will not be deleted automatically after unloading the module which prevents it from loading again if not deleted manually. There are also other implementation details that should be improved.

To be able to assess the usability of this approach better, a more in-depth performance analysis including performance profiling data is needed as this is the biggest limiting factor. More research is also needed to evaluate whether the process can be sped up by more parallelization in kernel and/or user space (this could possible make sense in the case of multiple network interfaces).

Furthermore, it would be interesting to investigate whether alternative techniques such as eBPF could be integrated into the process to make use of their respective advantages.

4 Conclusion

While working on this project, I learned a lot about Linux development: how loadable kernel modules work, what the Netfilter API is and how packets traverse the kernel, how thread safety becomes quite important in kernel space and how to use different approaches to inter process communication. I was able to greatly improve my understanding of operating system concepts by actually working with them directly.

While I reached my original project goal faster than I thought in the beginning of the semester, this also allowed me to introduce new goals after learning more about what the challenges are and how the different APIs work. I was therefore able to achieve more than I thought possible with the amount of experience and time I had.

The project greatly motivated me to deepen my knowledge and understanding of operating systems and especially the Linux kernel even further. As already explained in subsection 3.3 there are many ways to continue this project.

References

[1] J. Corbet. *A JIT for packet filters.* `https://lwn.net/Articles/437981/`. Online; accessed 30-September-2020.

[2] T. Graf. *Why is the kernel community replacing iptables with BPF?* `https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/`. Online; accessed 30-September-2020.

[3] *Kernel Space, User Space Interfaces.* `https://wiki.tldp.org/kernel_user_space_howto/`. Online; accessed 30-September-2020.

[4] I. Pronchev. *Packet Capturing Using the Linux Netfilter Framework*. Technische Universität München. Online; accessed 30-September-2020. 2006.

[5] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. "Performance Implications of Packet Filtering with Linux eBPF". In: *2018 30th International Teletraffic Congress (ITC 30)*. Volume 01. `https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ITC30-Packet-Filtering-eBPF-XDP.pdf`. 2018, pages 209–217.

[6] *The Netfilter Project*. `https://www.netfilter.org/`. Online; accessed 30-September-2020.

[7] *The netfilter.org "libnfnetlink" project*. `https://www.netfilter.org/projects/libnfnetlink/`. Online; accessed 30-September-2020.

LM-EV3-NS-OS-UART-BS-EP
Extending NinjaStorms by UART drivers

Marc Fabian Lindner

Hasso Plattner Institute for Digital Engineering
`marlindner@uni-potsdam.de`

This project report shows the development of a new feature for the NinjaStorms Operating System, which enables one to read data from the Lego Mindstorms EV3 UART Sensors. We create a function to print out colors that were measured with the EV3 color sensor, which can be used as a blueprint for other sensors. To do that, we will give a summary on the protocol, that the EV3 sensors use. We also discuss different problems and other complications that arose during the development process and how these problems were solved. For that matter, we will compare our own developed code with the original Lego Source Code for the EV3 and take a look at the different advantages and disadvantages of both implementations. We end our report with an outlook on possible further advancements and improvements of this feature.

1 Introduction

1.1 Motivation

The Lego Mindstorms EV3[1] (short: EV3) is a programmable robot out of Lego parts, which aims to introduce kids and teenagers to programming. On the EV3 itself installed is a custom made Linux distribution. However the EV3 also allows to launch a custom made operating system from a MicroSD Card. One of these custom made operating systems is NinjaStorms.[2] It is a very minimal operating system and only supports some basic features. For example it can already read out the state of the EV3 touch sensor or the EV3 ultrasonic sensor. Since most of the sensors, especially the newer ones, are more complex and support more features, they now use UART (Universal Asynchronous Receiver Transmitter) devices to communicate with the EV3. Currently there is no feature implemented in NinjaStorms, that would allow one to communicate with these UART sensors, which makes it impossible to read out data from these sensors. The main goal of this project is to implement this functionality and lay out the groundwork to read out data from all UART sensors.

[1] `https://www.lego.com/en-us/product/lego-mindstorms-ev3-31313`.
[2] `https://github.com/ninjastorms/ninjastorms`.

1.2 The development plan

To achieve this goal we created the following project plan:

1. **Control the already existing sensor code:** In this step we check if the already implemented code for the touch sensor and the ultrasonic sensors are working correctly. We do this to ensure, that we won't run into errors and problems later on, that arise from the assumption, that the already implemented code works correctly.

   This step also has the added purpose of learning and understanding the source code of NinjaStorms. We especially look out for parts of the code that can be reused for our own implementation of the UART sensors.

2. **Learn UART communication:** This step focuses on gathering information on UART devices and how they communicate with one another. We also need to research how we can send and receive data on a software level, since we do not know how much access to the hardware of the EV3 we will have. Therefore it is especially important to find information targeted specifically at the chipset that is used in the EV3.

   Since this step shares some common goals with the next and previous step, we can work on this step in parallel to the other two.

3. **Read the documentation for the EV3 and the UART sensors:** After we informed ourself how the UART communication works in general, we will take a look at the official documentation for the EV3. For us the Hardware Development Kit and the Firmware Development Kit are of great interest.[3] We will use these two documentations to learn, how the EV3 communicates with it's sensors and which top-level protocol or format it uses for that.

   All this is mainly preparation for the next step of our plan.

4. **Look at already existing implementations:** This is the most important part of our entire project work. Here we will take a look at previous works, that already implemented the UART protocol for the EV3. We have the choice between a lot of custom made operating systems for the EV3 that support communication via UART. To achieve the best results, we decided to mainly look at just one specific code base: the official operating system for the EV3, which was developed by the Lego Company.[4]

   We will use this code together with our accumulated knowledge from the previous steps as a base for writing our own implementation.

5. **Implement code to communicate with UART devices:** After we finished our research on the topic, we can start our own implementation of the UART

[3]`https://education.lego.com/en-us/support/mindstorms-ev3/developer-kits`.
[4]`https://github.com/mindboards/ev3sources`.

communication in NinjaStorms. Our goal is to write code that is independent from a specific sensor and can therefore work with every UART device, not just the EV3 sensors. If possible we also want our code to be not specific to the EV3 and also work in different situations (for example on a different hardware), most notably the QEMU[5] emulator.

6. **Add functions to communicate with EV3 sensors and implement one new sensor:** In the final part of our project work, we will make use of our previously created code and functionality to implement top-level functions for communicating with the EV3 sensors. We conclude our project with writing one function to read out actual data from one EV3 sensor, which can afterwards be used as a blueprint to write new functions for new EV3 sensors or functions with different behavior.

We need to point out, that many of the steps have overlapping parts. Therefore we often worked in multiple steps at once and sometimes even went back to old steps, but with some new knowledge or to gather new information. Even though it was not possible to follow the outlined order of steps strictly, the plan still provides a good overlook at the approach we took and the order in which we completed tasks during our project work.

Therefore we have to note, that the plan proved to be a very successful approach to achieve the final goal of this project.

2 Research Summary

In this chapter we will summarize the most import research findings for the future project work. This will mainly cover steps one to three of our research plan, but will also include some knowledge from step four.

2.1 UART

UART stands for "Universal Asynchronous Receiver Transmitter". It refers to different hardware devices, that can be used for a serial communication. Two UART devices can communicate with one another via two shared lines, commonly called the TX (Transmitter Line) and RX (Receiver Line). The UART device can accept one byte of data and it sends all bits sequentially, which will than be received by the opposite UART device, from which the full byte can be retrieved afterwards.

Since we use two different lines, we have a bi-directional communication. It is however to note, that UART devices don't use a clock line, like many other devices to transmit and receive data. Therefore we have to manually set the used bitrate (the number of bits transmitted per second) for every application and the bits need

[5] `https://www.qemu.org/`.

to be send asynchronously. To ensure a correct transmission, the bitrate needs to be identical on both ends.

We can access and modify different settings, as well as send and receive data on a software level via the use of multiple registers, which are mapped to specific hardware addresses.

2.2 Protocol used for sensor communication

The EV3 implements its own protocol on top of UART to communicate with the different EV3 sensors, here called the sensor protocol. The protocol itself is very modular, which makes it possible to easily use new sensors, without modifying or extending the original EV3 firmware.

Every sensor has different modes. The data it will measure and send is determined by the mode the sensor is currently in. A sensor can only be in one mode at a time and the amount of modes a sensor has is different for each sensor.

A visualization of the protocol can be seen in Figure 1. The protocol is described in further detail below. It can be roughly separated into two different parts.

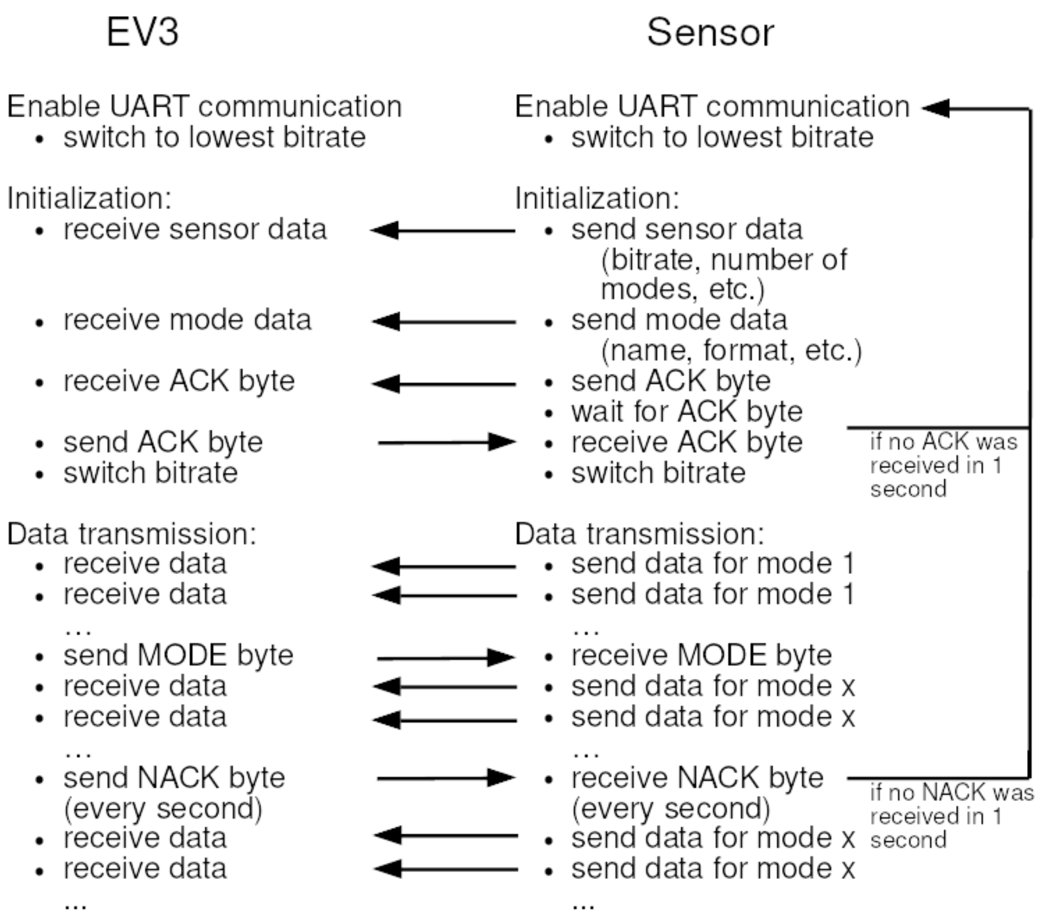


Figure 1: A summary of the sensor protocol

First comes the initialization sequence. In this part the sensor just sends a bunch of data to the EV3, mainly describing its different modes. It sends for example the names of every mode, as well as a range of values it can send while in this mode and in how many bits this data is transferred. But the sensor also sends some generic information about itself, for example the bitrate which it will use to send the measured data. At the end of the initialization sequence the sensor expects a so called ACK Byte (Acknowledgement Byte) from the EV3, to confirm the integrity of the transmitted data. This also marks the initialization sequence as complete. If the sensors does not read an ACK Byte, it will simply retransmit the data after a short amount of time. After the initialization sequence is completed, the sensor will change the bitrate in which it will transmit data from now on and it will start sending data for the default (first) mode. As an example, the initialization sequence of the EV3 color sensor can be found in Appendix 1.

With that the second part of the protocol begins, where the sensor will continuously send the data it measures. While switching modes, the sensor changed the bitrate it communicates in. Therefore we also need to adjust the bitrate of the EV3, in order to receive this data correctly. The data is send in packets. Each packet begins with a start byte, which contains the mode the sensor is currently in and also the amount of bytes of actual data, which is equal to the length of the packet minus 2. After that the data is send. A packet ends with a checksum byte, which is a simple XOR Operation over all previous bytes from the packet. The start and checksum both do not contain new information and are just a help to determine the start of a packet or check for the integrity of the transmitted data. We are able to change the current mode of the sensor by sending a specific MODE packet. The sensor will change modes, as soon as it receives the MODE packet. In order to confirm, that all the data was transmitted correctly, the sensor expects a NACK Byte (Not Acknowledgement Byte) to be send at least once per second. If it does not receive such a NACK Byte it will reset itself and we have to go through the initialization sequence again.

3 Implementation Phase

After we have gathered enough information from our research, we are ready to start implementing our own solution. Since a very primitive UART write function already exists in the NinjaStorms codebase to communicate with a serial console, we will start there and try to extend it, to allow also reading from a serial console. When this turns out to be successful, we will then start to implement our first layer of code, which handles the raw UART communication with different devices. In the end we will write our second layer of code, which will implement the communication protocol used by the EV3 sensors and write a small example function to communicate with an EV3 color sensor.

3.1 `getchar` function

We start by writing a simple `getchar` function, which will read one character of data from an UART device and output it. It is intended to be used with the serial console. We can base our function on the already implemented function `putchar`, which takes one character as an input and sends it to an UART device.

We can access the last read byte, by simply dereferencing a specific hardware address, which points to the Receive Buffer Register (RBR). This will also clear the register for the next byte. The hardware address can be found by reading the documentation of the chipset, that is used in the EV3. Because the built in UART device used for sending and receiving data has a FIFO built into it, we don't have problems with characters arriving in the wrong order or characters dropping out for now. This makes our code significantly easier.

The only thing we have to look out for, is to only read out the register, when actual new data is available. If we try to read the contents of the register, while there is no new data available, the read operation will return an undefined result. Therefore we have to wait until new data is available before we start the read operation. The UART device offers us two possibilities to do this.

The first is to make use of the built in interrupt system. By reading out the state of the Interrupt Identification Register (IIR), we can wait for an interrupt that signals new incoming data. This solution works in most cases, but we run the risk, of not receiving the interrupt, because it was already handled by another function. Since interrupts are only send once and the state of the IIR resets when we read it, too, we could potentially wait indefinitely.

The alternative solution is to make use of the Line Status Register (LSR). This special register holds various information on the state of the UART device. This also includes one bit which indicates wether new data is available to read. The LSR has the benefit, that it does not reset when we read it and it therefore always represents the current state of the device. This solution is also very easy to implement and keeps our code short and easy to understand.

Using the second approach, our code is only a few lines long and works very well with a serial console, as long as we can read and process the data fast enough. If we fail to do this, the internal FIFO of the UART device will overflow and new incoming data will overwrite old data. This can lead to weird and unpredictable behavior. This problem was not fixed, because we would need a working interrupt system and an ability to request memory at runtime to store newly read data, both of which were not implemented in NinjaStorms at the start of the project work.

3.2 GPIO pins and UART register setup

After implementing this first prototype, we immediately notice some big problems: Our code only works with a serial console and only on port 1. We assumed, that the sensors needs some extra initialization to start a UART communication, so we started looking for initialization functions in the original EV3 code. We found a few functions that were involved during the initialization phase, which handled the setup of some other UART register and that also enabled some General Purpose IO (GPIO) pins. So we extended our own implementation and created an initialization function.

But this still was unsuccessful and we were unable to read any data from the UART sensors. This problem proved to be a great obstacle and took many days of research across different sources and some help to overcome. We found the solution in the hardware documentation of the chip that was built into the EV3. This chip had some extra registers, one of which was the Power Management register (PWMG), which had to be set to a special value, in order to send power to the sensor. This was not needed if a serial console was attached, since it was already powered by the other end point of the connection, most likely a computer.

There is one other point of interest with this problem however: The PWMG register was never used in the original EV3 source code. We also could not find any other code, that would fulfill the same purpose. It is unknown wether the original EV3 source code works as expected and if so, how it manages to do so. Since building the EV3 source code proved to be a non trivial task, this was not further looked into.

We also need to note, that a closer look at the original EV3 source code revealed, that the UART communication for the sensor ports 3 and 4 works entirely different than the one for the first to ports and therefore needs it's own separate implementation. This was also confirmed by the EV3 hardware development kit. In order to keep this project manageable, we decided to focus our future work only on the ports 1 and 2.

3.3 UART implementation summary

The biggest decision we had to make for our implementation, was how we wanted to access the different UART registers. We had multiple options, all with there own advantages and disadvantages. We decided, it would be best, to keep our code the most readable and accessible for future work. Therefore we concluded, that the use of C-Structs would be the most optimal and give the most natural result.

We started by creating a new Struct datatype `uart_registers`, which lists all the names of the available registers in the order they appear in. This approach works well, since all the registers of one UART device are grouped together with no memory in between and all registers have the same byte length. In the case of the EV3 it was four bytes for each register. For every element of the struct we need to choose a datatype which corresponds to exactly four bytes of length. In most cases, the normal `int` datatype fulfills this requirement. We chose to use `unsigned int`, to make accessing single bits of a register a bit less troublesome. Since some registers use the same hardware addresses, but are restricted to read or write only access, we also made use of unnamed unions in our struct.

As a convenience, we also created an array which contains pointers to `uart_registers`, but instead of creating these structs ourself and writing the addresses into the array, we simply write the addresses of the first UART register to the array. This makes it possible to access all UART registers of all the UART devices (currently only

the devices for port 1 and 2) via a single array. The port can be selected via the numbers 0 to 3, or via the already existing `sensor_port` enum, and the register can be selected by accessing single values of the struct. If we, as an example, want to access the Line Status Register (LSR) of port 2, we can do this very easily with the following snippet:
`uart_ports[SENSOR_PORT_2]->lsr`

This approach has also some disadvantages. The biggest one, is that the whole code is extremely vulnerable to changes in the `uart_registers` struct and the corresponding data types. One small change could easily make the entire code fail and lead to undefined and potentially dangerous behavior. This problem however should not appear in practice, since modifications on the struct are not expected to happen and are also not needed, since the struct already contains all available registers.

The other disadvantage being that we cannot protect the registers from unintended use, meaning it is possible to write to a read-only register or read from a write-only register. Since a read and a write-only register share the same address space, this leads to the access of a wrong registers, which in turn can lead to undefined behaviors and errors. This could be solved to some degree by the use of wrapper functions for every register, but this would make the code unnecessarily complex for just such a simple task. Therefore at the moment caution is advised, when working with the UART registers directly.

We also implemented some small and simple functions to work with the UART device, which make use of the UART registers. The functions we implemented for example enable us to read and write data via a UART port or change the bitrate, that the UART device works with. These functions are very simple and only a few lines long, wherefore we will not discuss these functions in more detail here.

3.4 EV3 sensor example code

On top of the functions for the UART communication we made some functions to communicate with the EV3 sensors specifically. These functions mainly consist of reading data until some specific sequence of data was read and afterwards sending some specific data. These functions implement different aspects of the EV3 sensor protocol. With the help of these functions it is really easy to write code for different applications, which want to make use of the EV3 sensors.

We will now take a look at an example application, which uses the EV3 color sensor to detect colors and prints out the numbers associated with the detected colors on the console. A simple version of this demo function looks like this:

Listing 1: EV3 color sensor example function

```
void demo_ev3_color(void) {
  uartsensor_setup_color(SENSOR_PORT_2);
  uartsensor_change_mode(SENSOR_PORT_2, COL_COLOR);
  while(1) {
    unsigned char d = uartsensor_read_data(SENSOR_PORT_2);
    printf("%i\n", d);
    uartsensor_send_nack(SENSOR_PORT_2);
  }
}
```

Using this function as a blueprint, we can easily create new functions for other UART sensors. The primary thing we need to change for that, is the function `uartsensor_setup_color`. Therefore we will also take a look at this function:

Listing 2: The `uartsensor_setup_color` function

```
int uartsensor_setup_color(sensor_port_id port) {
    uartsensor_setup(port);
    uartsensor_wait_init(port, EV3_COLOR_ID);
    uartsensor_dump_bytes(port, EV3_COLOR_DUMP);
    if (!uartsensor_respond_ack(port)) {
        return 0; // Some Failure occurred
    }
    uartsensor_set_middle_bitrate(port);
    uartsensor_wait_data(port, UARTSENSOR_MODE_DEFAULT);
    return 1;
}
```

At the end of this function, the initialization part of the EV3 sensor protocol is done and we are able to read out data from the sensor by using the `uartsensor_read_data` function.

In most cases it should be enough to change only a few values, to adapt this function to work with a new sensor. One of these being the `EV3_COLOR_ID` value, which holds the ID of the sensor. This ID is sensor specific and is sent by the sensor itself and can be found out by hexdumping the entire initialization protocol of the sensor. Another value is the `EV3_COLOR_DUMP`, which gives the length of the initialization protocol of a sensor. Lastly one may need to change the bitrate a sensor uses. This information, again, is sent by the sensor itself during the initialization.

The last thing to note is the `uartsensor_send_nack` function. Since the EV3 sensor protocol expects a NACK Byte to be send every few milliseconds, it is necessary, that the user calls this functions often enough. If the function is not called often enough, the EV3 sensor will reset and a new call to the `uartsensor_setup_color` function is necessary.

4 Evaluation

Compared to the original Lego Source Code, our implementation is quite small. When tested with the EV3 color sensor it worked very well and errors were only encountered very rarely.

Reading data from the EV3 sensors is also very fast and mainly limited by the bitrate the sensor uses and not our implementation. Going through the initialization sequence however is comparatively very slow, with around one to two seconds.

One reason for this is, that during the initialization sequence the sensor uses the lowest available bitrate. The main issue however is that when the sensor is plugged into the EV3, there is a big change of errors occurring during the start of the communication. When this happens we are unable to detect that a initialization sequence has just started and need to wait for the next iteration, which consumes a lot of time. This is really hard to avoid and requires multiple different features, that are currently not implemented in NinjaStorms, to create a fully working solution to this problem.

The main drawback of our implementation is the fact, that it is mostly hardcoded. This is quite a big problem, since the sensor communication protocol is designed to transmit all relevant data during the initialization sequence, so that even new sensors can be used without modifying or extending the already existing code. Our implementation completely ignores this modularity of the protocol and therefore every new sensor needs to be initialized manually by the programmer. This has the benefit that the initialization protocol does not need to be parsed and evaluated, which makes our code faster. The main reason we choose to implement a non modular approach, was to keep our code small and simple. Implementing a modular approach is possible, but would drastically increase the code size and complexity.

One big problem with this approach however are communication errors. Communication errors during the initialization sequence are no real problem and are solved by just waiting for the next initialization sequence. A big problem however are communication errors that occur during the transmission of data. Our code has no recovery option implemented, so communication errors are hard to check for and hard to avoid. This makes our code very unstable, if these errors happen regularly. In practice, we never encountered these errors and the UART protocol itself already prevents simple communication errors with the help of a parity bit, so the significance of this problem is not as high.

One other main downside with our implementation is the non existing handling of NACK bytes. The sensor expects a NACK byte to be sent by the EV3 at least every second. This informs the sensor, that it is still connected to the EV3 and that the data has been received correctly. Optimally the NACK byte would be transmitted fully automatic by our implementation, completely independent of the top level application. Since we need to time the transmission of the NACK byte, we would need some kind of timer based interrupt system. A timer is already implemented in NinjaStorms, but it is used for the scheduler. Therefore changing the timer would mess with the scheduler of NinjaStorms, which would lead to unexpected behavior. A working interrupt system, which makes use of the timer, first needs to be implemented and the scheduler needs to be changed, to use this new interrupt system. Another alternative would be to just send a NACK byte as often as possible in the background. This only requires a basic process scheduler and not a fully functional interrupt system. Because of other projects on the EV3 that happened at the same time, it was very likely that the scheduler would see a overhaul, which would mean that our code would need to be rewritten to fit this new scheduler, once all projects are merged together. Therefore we did not choose to implement this approach for the time being. At the current state, the programmer has to manually send the NACK byte at least every second. Depending on the application this can be a very easy or a very hard task. Since all projects have now finished, this would be the problem most worthy reinvestigating.

A working timer based interrupt system would also benefit our code in other places, for example when detecting the start of a initialization sequence.

5 Conclusion

We could reach our goal of implementing the UART sensor protocol in NinjaStorms. With this new functionality many new sensor types can be used with the EV3. The actual effort to implement a new UART Sensor is comparatively small and easy, requiring only one manual parsing of the initialization sequence of a sensor. This even enables an easy implementation of many custom made sensors, that follow the same protocol.

A lot of the base functionality was implemented and only very few features needed to be cut, mostly because of technical limitations. Excluding some special details of the EV3, most of the code could also be used for some other hardware, to enable a basic communication with the EV3 sensors or other UART devices.

During our research phase we had only a very limited amount of resources available. Additionally the documentation of the EV3 was very short and contained not much useful information. Therefore we had to rely on the original EV3 source code as our main source of information. This proved to be challenging at some points, because the original source code is very big (with over 4000 lines of code) and can also be very complicated. Despite all of this, we were able to write our own implementation in considerably less space, and it still works really well.

Considering all of this, we can only see the project as a great success.

6 Annex – Initialization sequence of the EV3 color sensor

This is the initialization sequence from the EV3 color sensor. Packages are split up in multiple lines, to improve readability. After every line, a human readable interpretation of the package is provided.

```
// TYPE: Device type
40 1d a2
// MODE: Number of modes
49 05 02 b1 | 6 modes, 3 visible |
// SPEED: Max communication speed
52 00 e1 00 00 4c | bitrate of 57600 |

// Mode 5:
// NAME: Name of mode
9d 00 43 4f 4c 2d 43 41 4c 00 41 | COL-CAL |
// RAW: Raw value span
9d 01 00 00 00 00 00 ff 7f 47 a4 | 0.0 - 65535.0 |
// SI: SI unit value span
9d 03 00 00 00 00 00 ff 7f 47 a6 | 0.0 - 65535.0 |
// FORMAT: Format data
95 80 04 01 05 00 ea | 4 * 16Bit values, 5 decimal places |

// Mode 4:
9c 00 52 47 42 2d 52 41 57 00 5d | RGB-RAW |
9c 01 00 00 00 00 00 0c 7f 44 55 | 0.0 - 1023.0 |
9c 03 00 00 00 00 00 0c 7f 44 57 | 0.0 - 1023.0 |
94 80 03 01 04 00 ed | 3 * 16Bit values, 4 decimal places |

// Mode 3:
9b 00 52 45 46 2d 52 41 57 00 5c | REF-RAW |
9b 01 00 00 00 00 00 0c 7f 44 52 | 0.0 - 1023.0 |
9b 03 00 00 00 00 00 0c 7f 44 50 | 0.0 - 1023.0 |
93 80 02 01 04 00 eb | 2 * 16Bit values, 4 decimal places |

// Mode 2:
a2 00 43 4f 4c 2d 43 4f 4c 4f 52 00 00 00 00 00 00 00 6d | COL-COLOR |
9a 01 00 00 00 00 00 00 00 41 25 | 0.0 - 8.0 |
9a 03 00 00 00 00 00 00 00 41 27 | 0.0 - 8.0 |
// SYMBOL: SI symbol
9a 04 63 6f 6c 00 00 00 00 00 01 | col |
92 80 01 00 02 00 ee | 1 * 8Bit values, 2 decimal places |

// Mode 1:
a1 00 43 4f 4c 2d 41 4d 42 49 45 4e 54 00 00 00 00 00 6b | COL-AMBIENT |
99 01 00 00 00 00 00 00 c8 42 ed | 0.0 - 100.0 |
99 03 00 00 00 00 00 00 c8 42 ef | 0.0 - 100.0 |
99 04 70 63 74 00 00 00 00 00 05 | pct |
91 80 01 00 03 00 ec | 1 * 8Bit values, 3 decimal places |

// Mode 0:
```

```
a0 00 43 4f 4c 2d 52 45 46 4c 45 43 54 00 00 00 00 00 7d | COL-REFLECT |
98 01 00 00 00 00 00 00 c8 42 ec | 0.0 - 100.0 |
98 03 00 00 00 00 00 00 c8 42 ee | 0.0 - 100.0 |
98 04 70 63 74 00 00 00 00 00 04 | pct |
90 80 01 00 03 00 ed | 1 * 8Bit values, 3 decimal places |

// ACK
04
// SYNC
00

// Repeat
```

Quantum Computing from a Software Developers Perspective

Selina Raschack

Hasso Plattner Institute for Digital Engineering
selina.raschack@student.hpi.de

As the limits of Moore's Law lay within the physical limitations in how small transistors can be created, new ways of computing are needed. Quantum computing is one of them and it is important for software developers to familiarize themselves with its key concepts. Developers that wish to integrate quantum computing into their portfolio need to find ways to come up with algorithms that incorporate the benefits from quantum mechanics. Well-established principles in software design and implementation should be preserved to keep the code maintainable and flexible for further changes. Given a simple puzzle game to implement, in this paper a hybrid approach is taken to analyze if the SOLID design principles and Test Driven Development can be applied on quantum programs.

1 Introduction

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain."

Gordon E. Moore [5]

In 1965, Gordon E. Moore, co-founder and chairman emeritus of Intel Corporation, published his observations on the average growth of processing power for computers. Back then, a CPU contained one core whereas today chips usually contain multiple cores as the underlying components can be designed and produced in more compact ways. The boundaries of Moore's Law lay within the physical limitations in how small transistors can be created. [4] Therefore new ways of utilizing given processing power more efficiently are researched constantly. One way is to refine the ways in data procession itself to handle big amounts of data efficiently. Another approach is to design and implement specialized computing units, quantum computers being one of them. A couple of problems that are hard to compute for classical[1] computers are easy or at least feasible problems for quantum comput-

[1]The term classical refers to computer systems working with bits that represent exactly one out of two possible states {0, 1}.

ers. To benefit from these strengths it is important for classical software developers to enhance their tool box with concepts of quantum computing.

In this paper, quantum computing will be approached from the perspective of a classical software developer with no or very limited knowledge on quantum computing. An exploratory way would be starting with a couple of basic concepts in quantum computing and then implement a small demo application to illustrate how they work. In this paper, a more conservative approach is taken. Instead of looking for problems themselves that are suitable to be solved with quantum computing an already given classical solution will be analyzed to identify the parts that can be encapsulated into a quantum computing module. Therefore, in the subsection 2.1 a simple puzzle game and some related concepts in game design in general are introduced. Key concepts in quantum computing will be introduced in subsection 2.2 to be able to understand the differences between classical and quantum computing. In section 3 two types of problems are highlighted: How to come up with an algorithm that incorporates the benefits from quantum mechanics? How to take well-established principles in software design to keep the code maintainable and flexible for further changes? Both of them are addressed in section 4 by providing a hybrid setup between classical and quantum computing to implement parts of the given puzzle game in section 5. The implications of this approach and possible takeaways for classical software developers or project teams who wish to include quantum computing into their portfolio will be evaluated in section 6. A couple of additional resources will be provided in section 7 after which the paper concludes.

2 Context

There is already a very active research community working on solutions to provide quantum proof cryptography algorithms. To raise awareness for other disciplines and to get businesses interested into features and advantages of quantum computing the IBM researcher Dr. James Wootton has experimented with quantum based approaches for the computer games industry. In his article "Creating infinite worlds with quantum computing" he provides a proof of concept in how to randomly generate a terrain and to ensure features that prevent the player from being trapped.

2.1 Motivation

In this paper a simple puzzle game is considered. The basic setup consists of a rectangular shaped board with colored tiles on it. Tiles are set up randomly and the core goal for each move of the player is to line up three or more tiles with the same color in a column or a row (*match*). The player can choose freely which tile they want to move and then change the tile's position on the board with one of its neighbored tiles. The move is valid in case one of the two moved tiles is part of a match. Otherwise the move is invalid and has to be taken back. Mathematically, the game board can be represented by a $n \times m$ matrix and the colored tiles by natural numbers. Cells with the same natural number in the matrix imply that the tiles on the board at the respective position have the same color:

$$M = \begin{bmatrix} 0 & 1 & 3 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \\ 3 & 0 & 2 \end{bmatrix}$$

In the given example moving the tiles on position m_{31} and m_{32} will result in the first three cells of the second row containing color 1. Therefore, this would be a valid move. On the other hand, moving the tiles on position m_{11} and m_{12} would be an invalid move.

From the perspective of good game design the following two aspects are important to keep in mind [8]:

Create excitement There are different ways to create excitement for players. Some of them qualify better for certain type of games than others. The constrains opposed are that it gives the player a unique entertaining experience. This does not automatically imply that the content has to be randomly created. If randomness is chosen on the other hand, the content has to be *random enough* without becoming *too random*.

Be solvable By creating parts of a game randomly a great variety of challenges for players can be created on runtime. The constrains opposed on randomly created solutions in game design are that the levels need to be solvable and created in a way that there are no features that trap the player. These conditions have to be satisfied in a feasible amount of time or computer memory to keep the player excited.

In the given puzzle game *random enough* requires that the applied randomness has an effect on how to solve the puzzle. In the given example it would not be enough to randomly assign a different color to a natural number each time the board is set up. Since the underlying pattern would stay the same and moving the tiles on position m_{31} and m_{32} would always be a valid solution. If we further assume that not all tiles are movable, always assigning them to a different color each time the board is set up would be *too random*. The player has no indication to distinguish between movable and solid tiles. Therefore, always assigning the same color to solid tiles keeps players excited as they know which tiles can be used when searching for a valid move. To *be solvable* implies that there is at least one valid move for the player to take each time the board is set up.

There are a couple of edge cases when finding a suitable setup for the board and a classical computing approach can be time consuming:

- not enough different colors are used
- too many different colors are used

In the first case it will become difficult to set up the board in such a way that it does not already contain a solution. When setting up the board randomly this step may has to be done multiple times before the board is set up in a way that it requires the players to make a valid move. In the second case it will become difficult to set up the board in such a way that is is solvable. When setting up the board randomly this step may has to be done multiple times before the board is set up in a way that the player can make a valid move at all.

2.2 Background

In 1948 Claude E. Shannon published his paper "A Mathematical Theory of Communication" leading into the fields of Information Theory. In a general communication system an *information source* produces a message to be communicated. But it is not the message itself that is transmitted to the *destination* over the underlying *channel* but a suitable signal. No matter what the chosen medium for the channel is, it comes with a source for noise. In their paper "Information Theory and the Digital Age" Aftab et. al. summarized a couple of Shannon's key concepts. The following three concepts illustrate the difference in discrete and continuous communication [1]:

Channel capacity Every communication channel has a speed limit within which it is possible to transmit information with zero error.

Digital Representation By distinguishing between a message and its representation as a signal that is transmitted over the channel, text message, same as images, sound or video messages can be represent digitally.

Entropy is the information content of a message. It measures the amount of uncertainty involved that the received message is the one originally transmitted.

In classical computing a discrete communication is utilized whereas in quantum computing it is a continuous communication. From a developers perspective it therefore is important to understand that quantum computing relies on probabilities rather than deterministic states.

A very in-depth introduction to quantum computing and the mechanics and differences in classical deterministic systems, classical probabilistic systems and quantum systems was published by Noson S. Yanofsky [9]:

System A system contains a finite set of states and a finite set of transitions to change from one state into another. As an example graphs can be used to visualize a system with states. The states are represented by the nodes of the graph and transitions by its edges.

Classical deterministic system A classical deterministic system is a system with its state and transitions being deterministic.

Classical probabilistic system A classical probabilistic system is a system with its state and transitions not being deterministic. The probabilities of states and transitions are given as real numbers between 0 and 1.

Quantum system A quantum system is a system with its state and transitions not being deterministic. Furthermore, the probabilities of states and transitions are given as complex numbers c such that $|c|^2$ is a real number between 0 and 1.

After giving an overview of the underlying mathematical concepts and how problems are solved within these systems, he introduces classical and quantum generalizations of bits to qubits and logical gates to quantum gates [9]:

Bits In classical computing bits are the atomic parts each information can be broken down to. A bit represents one of the two disjoint states $|0\rangle$ (*zero*) and $|1\rangle$ (*one*). Mathematically, a bit can be represented by two 2×1 matrices:

$$|0\rangle = \begin{array}{c} 0 \\ 1 \end{array}\begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{array}{c} 0 \\ 1 \end{array}\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The first matrix contains all possible values, *zero* and *one*. The second matrix contains two discrete values, *zero* and *one*, to tell both states apart from each other.

By using this type of notation the whole concept of bits can be easily enhanced to qubits. Instead of having two concrete values the second matrix contains two complex elements c_0 and c_1. Therefore, the classical bit can be seen as a special type of qubit:

Qubits In quantum computing the concept of bits is extended to the so called *quantum bits* or *qubits*. A *qubit* can be any state that can be represented as:

$$\begin{array}{c} 0 \\ 1 \end{array}\begin{bmatrix} c_0 \\ c_1 \end{bmatrix}, \text{ where } |c_0|^2 + |c_1|^2 = 1 \text{ and } c_0, c_1 \in \mathbb{C}$$

Note that from the previous introduced definition of a *quantum system* the probabilities of states and transitions are complex numbers that should comply to the constrain that their square product leads to real number between 0 and 1.

Classical Gates In classical computing logical gates like *or*, *and*, or *not* are used to manipulate bits. Mathematically, they can be represent as a $2^m \times 2^n$ matrix with n being the number of input bits and m being the number of output bits of the logical gate. To give an example:

$$NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

NOT of $|0\rangle$ equals $|1\rangle$ and vise versa.

Quantum Gates A quantum gate is any unitary matrix that manipulates qubits. It turned out that one of the most important matrices in quantum computing is the *Hadamard matrix*:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

as it is also its own inverse. In this paper the rotation operator R_y-Gate is used:

$$R_y = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

It is a single-qubit rotation around the y-axis where θ is a radian.

In the lecture "Quantum Computing – How does this work?" The IBM researcher Dr. Johannes Greiner discusses a couple of practical implications when working on quantum computers [2]:

Superposition With the given representation of a qubit it is possible to create a quantum state that is a combination of $|0\rangle$ and $|1\rangle$:

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = c_1|0\rangle + c_2|1\rangle$$

If c_1 and c_2 are non-zero, the qubit's state contains both $|0\rangle$ and $|1\rangle$ at the same time which is called superposition.

Entanglement With two qubits a quantum state can be represented as a combination of:

$$c_1|00\rangle + c_2|01\rangle + c_3|10\rangle + c_4|11\rangle$$

where $|c_1|^2 + |c_2|^2 + |c_3|^2 + |c_4|^2 = 1$, $c_1, c_2, c_3, c_4 \in \mathbb{C}$ and where $|01\rangle$ means the first qubit is in state $|0\rangle$ and the second qubit is in state $|1\rangle$. If two or more of the complex factors are non-zero it is not possible to separate the qubits. This phenomenon is called entanglement.

Measurement A qubit's state cannot be determined exactly but it can be measured. A measurement will force a qubit's state $c_1|0\rangle + c_2|1\rangle$ to be either $|0\rangle$ or $|1\rangle$ with $|c_1|^2$ probability to be $|0\rangle$ and $|c_2|^2$ probability to be $|1\rangle$. It is important to highlight that the result of a measurement is only valid for the given measurement. Therefore, it is mandatory to measure more than once.

Interference To increase the probability of getting the right answer during measurements the concept of interference is used. As discussed earlier, in quantum computing signals are transmitted continuously and therefore noise handling is an important factor to keep in mind.

3 Problem

In classical computing a variety of well researched and established tools and techniques exists already. To give an example, in a classical business application a three layered architecture consisting of a data layer, business logic layer, and presentation layer is canonical. From the perspective of a software developer it does not matter whether the task is to build a classical business application, to build an application that utilizes machine learning algorithms to work on big data or to build an application that utilizes quantum computers to efficiently work on factorization problems. The concrete algorithm might be new or the way how to access the computing hardware is, but the well established base architecture should not be undermined. Therefore, from a developers perspective it is desirable to add the quantum computing concepts without compromising tools and techniques that are invariant to the type of computing.

In this paper the following two principles will be evaluated:

SOLID principles The SOLID principles are five design principles, mainly in object-oriented programming, to ensure software remains verifiable, maintainable and flexible.[3, 7]

- A class [a function] should only have a single responsibility
- A software entity should be open for extension but closed for modification
- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that programming
- many client-specific interfaces are more effective than a generic multipurpose one
- dependency should rely on abstraction

Testing Test Driven Development covers a programming principle to ensure code is verifiable and correct.

To make full use of the benefits of quantum computing software developers have to consider the previous discussed fundamentals in information theory and quantum mechanics. Therefore, choosing a suitable quantum algorithm or coming up with one for themselves will be necessary.

4 Solution

To abstract some of the implementation details and make quantum computing more accessible to different type of projects various tools and frameworks are published and refined constantly. IBM's "Quantum Experience"[2] and the Python library "Qiskit"[3] being some of them and both tools will be used the following implementation part. This supports a hybrid setup in which quantum computing modules can be integrated along with classical computing modules. With such a hybrid approach it is also easier to preserve suitable well-established principles in classical software design.

To identify the parts that should be solved with quantum computing a given problem will be broken down into multiple steps. In the given puzzle game setting up the board with classical computing could be broken down as follows:

[2]https://quantum-computing.ibm.com/docs/.

[3]https://qiskit.org/textbook/preface.html.

- Randomly assign a color to each tile on the board
- Check, if the board is solved
- Move a tile

With step one the board is randomly set up. With the second step the random setup can be checked, if it already is solved. To check if the board is solvable, step three can be used to move two tiles and then step two can check if the board would be solved.

In a hybrid scenario those steps have to be identified that are suitable to approach with quantum computing. Therefore, setting up the board will be broken down into slightly different steps:

- Randomly assign a color to a predefined set of tiles on the board
- Find a solution that most likely solves the board
- Assign a color to those tiles that have been left out in the first step based on the found solution in step two

In the first step classical computing will be used to create a board setup where a couple of tiles already have a color assigned. This setup will be used as a seed in step two to determine a solution that most likely solves the board. This step will be solved with quantum computing. In the final step the remaining tiles will get a color assigned with a classical computing approach. Due to the computed solution from step two for each tile the color that most likely creates an instant solution is known. Therefore, this color should not be assigned. On the other hand, a color that will most likely create a solution on one of the neighboring positions can be desirable as it makes the board to be solvable.

5 Implementation

In this chapter the quantum computing part of finding a solution that most likely solves a given board in the puzzle game is briefly discussed. In this example the algorithm from [8] is reused.[4] This is not a perfect solution for the puzzle game but gives a first idea in how setting up a board in such a way that tiles with the same color are clustered. The goal is to keep the algorithm itself encapsulated and testable. The following Listing 1 contains the steps within a basic quantum module to solve a given board[5] Figure 1 shows how the *solve board* circuit looks like.

[4]In the original paper one of the core ideas is to randomly create a terrain by using a heightmap. Therefore, for each cell on a grid a value is calculated in a way that neighbored cells only contain values that are relatively close to each other.[8] In the original paper the idea behind this function and how it works is discussed in more detail. For this paper it will only be applied.

[5]It should be clearly stated that most of these steps are done in the *quantum_tartan* function in [8]. The code is slightly rearranged to make it more suitable for testing.

Listing 1: Basic solve board quantum module in qiskit

```
state = pre_quantum_steps(board)
circuit, reqister = init_circuit(state, amount_of_registers)
circuit = solve_board_circuit(circuit, reqister)
backend = get_backend()
job = run(backend, circuit, amount_of_shots)
solution = post_quantum_steps(job)
```

```
n = 10
theta = 4
circuit.ry(2*pi*theta,reqister)
c = ClassicalRegister(n)
circuit.add_register(c)
circuit.measure(reqister,c)
circuit.draw()
```

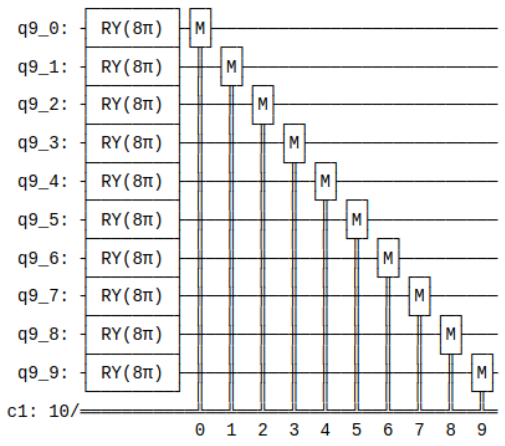


Figure 1: Quantum circuit with ten registers representing the solve board algorithm

6 Evaluation

After breaking down this task into multiple steps it can be seen that for quantum computing modules different software principles can be applied as well. For a TDD approach building the concrete circuit needs to be tested as well. By using the command pattern[6] building the concrete quantum circuit is encapsulated. To actually test the circuit the TDD approach could not fully be applied. The circuit can be tested on a functional level and with the modular design of the rest of the quantum module a small test environment can be set up. The outcome can than be evaluated and the circuit can be easily enhanced or replaced if needed. By designing these type of functional tests developers have to keep in mind that in quantum computing states are measured in probabilities. The qiskit library also offers a package to represent a circuit as a directed acyclic graph. This may be

[6]The command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.[6] In this example it is used to encapsulate the "solve board" behavior.

used to then design test cases around the graph representation. Apart from that the principle is invariant from the applied way of computing.

The general idea of breaking down complex steps into a set of simple steps is invariant to the applied computing. By applying further iterations of breaking down steps, a point can be reached when each step covers a single responsibility. This applies to quantum computing as well. Keeping the steps classical that are not suitable for quantum computing, preserves the full support of the already established principles.

On the other hand, choosing a suitable quantum algorithm or coming up with one is not easily applied. Knowing and understanding concepts from the quantum mechanics background is a first essential step. Applying the knowledge into problem solving is a different one. In the given paper this has not been achieved. Instead, it has been shown a way on how to utilize solutions already created by other developers that are more experienced in quantum computing. Especially in project teams this leads to the question whether each software developer has to have a deeper understanding of quantum computing or whether it is sufficient enough to bring together a diverse team. One of the developers can have a deep knowledge in quantum computing whereas the other one is specialized on other aspects of software design. Both of them only need a basic understanding of each others expertise so they can understand each other's needs when it comes to actually designing a software component.

Finally, it should be acknowledged that the mathematical background that will be part of the education for next generation software developers will adapt in case quantum computers become more accessible. Coming up with quantum algorithms will become more natural for developers with a deeper understanding of these underlying mathematical concepts.

7 Related Works

The referenced paper by Yanofsky ([9]) was written as an excerpt for the book "Quantum Computing for Computer Scientists" by Yanofsky and Dr. Mirco Mannucci. The reader that wishes to approach quantum computing fundamentals in a structured way, including different theoretical topics, may take their next steps by reading this book. The book starts with complex numbers, discusses variety of quantum algorithms or protocols relevant in cryptography. But also topics in theoretical computer sciences, information theory, or quantum hardware are covered.

To approach quantum computing through practically applying it, the IBM Quantum Experience and the Qiskit textbook can be consulted as next steps. The textbook gives a couple of short challenges to solidify basic quantum computing concepts. On a more advanced level software developers can participate in the ICPC Quantum Computing Challenge powered by IBM Quantum. [7] In these chal-

[7]`https://challenges.quantum-computing.ibm.com/icpc`.

lenges quantum computing is applied to solve problems in the fields of physics, finance, chemistry and more. To have a deeper look into current topics of game development with quantum computing Dr. Wotton's blog[8] might be a good start. It also should be mentioned that there is python support for the open source game engine Godot. [9]

There are quantum computing libraries and simulators for other programming languages than Python as well. Oracle gives a brief introduction into quantum computing with Java and Strange. [10] Developers specialized in developing Microsoft applications can try out Microsoft Q#. [11]

8 Conclusion

In this paper quantum computing was approached by looking into the development of a small puzzle game. The two major questions raised were "How to come up with an algorithm that incorporates the benefits from quantum mechanics?" and "How to take well-established principles in software design to keep the code maintainable and flexible for further changes?" The general concept throughout the whole paper is to start at point well known. The given problem is one that has been implemented with classical computing multiple times already. When setting the required theoretical background, concepts in quantum computing have been presented as generalizations of the equivalent concepts in classical computing. When implementing a small quantum module an algorithm was reused that has been designed by a more experienced quantum software developer and researcher. By taking this approach well established tools and techniques in that are invariant to the underlying computing concept could be naturally applied on the quantum module as well. To apply quantum computing when working on a solution for a given problem, the perspective has to include an understanding on how quantum computers work. To solve the board setup for the given puzzle game the steps of the algorithm have been rephrased and during implementation it has been shown that the understanding is required when it comes to design test cases for pure quantum functions. To come up with own unique solutions that fully benefit from the advantages of quantum computing software developers have to deepen their theoretical background in this area as well. The mathematical foundation and understanding the implications that come with concepts like probability or interference is essential. By starting with single steps and keeping modules small well-established principles in software design and development can be preserved and help to keep quantum modules and the applications around them maintainable and open for future changes.

[8]https://decodoku.medium.com/.

[9]godot...https://godotengine.org/ https://github.com/touilleMan/godot-python.

[10]https://developer.oracle.com/java/quantum-computing.html.

[11]https://docs.microsoft.com/en-us/azure/quantum/?view=qsharp-preview.

References

[1] Aftab, Cheung, Kim, Thakkar, and Yeddanapudi. *Information Theory and the digital Revolution*. 6.933 Project History, Massachusetts Institute of Technology, 2001.

[2] HPI and IBM. *On the Road to Quantum Computing*. Online; accessed 30-October-2020. Oct. 2020. URL: `https://open.hpi.de/courses/ibmpower2020`.

[3] R. c. Martin. *Design Principles and Design Patterns*. 2000. URL: `www.objectmentor.com`.

[4] MemeBridge. *Moore's Law or how overall processing power for computers will double every two years*. Mar. 2021. URL: `http://www.mooreslaw.org`.

[5] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pages 33–35. DOI: `10.1109/N-SSC.2006.4785860`.

[6] Wikipedia. *Command pattern*. Online; accessed 30-October-2020. Feb. 24, 2021. URL: `https://en.wikipedia.org/wiki/Command_pattern`.

[7] Wikipedia. *SOLID*. Online; accessed 30-October-2020. Feb. 24, 2021. URL: `https://en.wikipedia.org/wiki/SOLID`.

[8] D. J. Wootton. *Creating infinite worlds with quantum computing*. Online; accessed 30-October-2020. Apr. 2019. URL: `https://medium.com/qiskit/creating-infinite-worlds-with-quantum-computing-5e998e6d21c2`.

[9] N. S. Yanofsky. *An Introduction to Quantum Computing*. 2007. arXiv: `0708.0261 [quant-ph]`.

ISBN 978-3-86956-524-8
ISSN 1613-5652