

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Vít Kabele

Syscall emulation support in HelenOS

Department of Dependable and Distributed Systems

Supervisor of the master thesis: Mgr. Vojtěch Horký, Ph.D.

Study programme: Computer Science

Study branch: ISS

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In *Prague* date *18.6.2021* *Kabek*

Author's signature

I would like to thank Mgr. Vojtěch Horký, Ph.D. for his professional yet friendly supervision of this thesis, my entire family for their endless support during my studies, and Michaela Vystrčilová for her (not only emotional) support and proof-reading. Last but not least, this work would not be possible without all the inspiring people, both teachers and students, from the Faculty of Mathematics and Physics.

Title: Syscall emulation support in HelenOS

Author: Vít Kabele

Department: Department of Dependable and Distributed Systems

Supervisor: Mgr. Vojtěch Horký, Ph.D., Department of Dependable and Distributed Systems

Abstract: There are two main options for running a program intended for one operating system on a different one. We can modify the program to use the API of the new OS. Alternatively, we can provide a compatibility layer in the new OS, transparent to the program, without changing the application. HelenOS ecosystem already allows the first. This thesis focuses on supporting the latter. This thesis describes the kernel binary interface and analyses existing solutions on Linux, Windows, and specialised systems. Then we describe our prototype that transparently traps syscalls and emulates them. The emulation is implemented fully in userspace (except for a small kernel trampoline), and its code is executed in the context of the original application. The result allows running some of the unmodified Linux programs (focusing on GCC toolchain) on the x86_64 instance of the HelenOS system.

Keywords: HelenOS POSIX Syscall emulation Linux

Contents

| | |
|---|-----------|
| Introduction | 3 |
| 1 Context | 4 |
| 1.1 Software emulation | 4 |
| 1.2 Privilege levels | 5 |
| 1.3 Syscalls | 6 |
| 1.4 Compilation process | 6 |
| 1.4.1 Dynamic linking | 7 |
| 1.5 ELF | 7 |
| 1.5.1 Execution | 8 |
| 2 Analysis | 9 |
| 2.1 The Linux userspace | 9 |
| 2.1.1 Program loading | 9 |
| 2.1.2 Syscalls | 10 |
| 2.2 Introduction to HelenOS | 13 |
| 2.2.1 Threads and tasks | 13 |
| 2.2.2 Program loading | 14 |
| 2.2.3 Syscall architecture | 14 |
| 2.3 Windows Subsystem for Linux | 15 |
| 2.3.1 Pico process | 16 |
| 2.4 Syscall User Dispatch | 16 |
| 2.5 Wine | 18 |
| 2.6 libposix | 20 |
| 2.6.1 liblinux | 20 |
| 2.7 Other implementations | 21 |
| 3 Design and implementation | 23 |
| 3.1 Program lifecycle | 23 |
| 3.2 Architecture | 24 |
| 3.3 The Masquerade API | 25 |

| | | |
|-------|---------------------------------------|-----------|
| 3.3.1 | Segment registers | 26 |
| 3.4 | Linux emulator | 27 |
| 3.4.1 | Memory layout | 27 |
| 3.4.2 | Program loader | 28 |
| 3.4.3 | Syscall emulation | 28 |
| 3.5 | Considerations | 31 |
| 3.6 | Evaluation | 32 |
| | Conclusion | 34 |
| | Bibliography | 35 |
| | A Running Linux app in HelenOS | 36 |

Introduction

Motivation

A rule of thumb states that the popularity of a platform is determined by the amount and quality of available software. This is bad especially for hobby platforms, such as HelenOS. HelenOS claims to be a general purpose operating system able to run on servers, desktops and embedded devices.

The HelenOS API is designed from scratch and therefore incompatible with any other existing operating system. HelenOS now only supports two types of apps: the native programs built against HelenOS libraries, and programs modified to conform to them. As a consequence, the amount of available applications is extremely limited. This blocks a wider adoption of the HelenOS system.

Bringing apps from different platforms, for example Linux, would alleviate the problem. Ideally, we would like to have a binary compatibility with the Linux kernel, but without abandoning the carefully crafted native APIs. Binary compatibility would allow HelenOS users to run unmodified Linux binaries along with native ones. To achieve binary compatibility, we need to build an emulation layer.

Goals

In this thesis we take a look at the kernel/userspace interface focusing on HelenOS and Linux.

Our goals are to:

- Analyse existing solutions for implementing binary compatibility on both established (Linux, Windows) and lesser known (L4Linux, Mach) platforms. Based on the analysis we propose the best model for HelenOS.
- Create a prototype implementation of the proposed solution for the x86_64 architecture and Linux applications.
- Run unmodified Linux programs in our instance of HelenOS.

Chapter 1

Context

In this chapter we introduce some of the basic concepts of program execution on software platforms.

We describe the software emulation use-case, we take a look at the CPU privilege model and the most common way of calling functions across the privilege boundary. We also present the compilation process leading from the source code to the executable and the popular ELF file format.

1.1 Software emulation

Several factors determine the acceptance of operating systems, some of them being the amount, kind and quality of available software.

Windows is famous for its support of video games and office suites; Linux is popular among programmers, while content creators widely use macOS. Similar characteristics apply to most established platforms.

This fact blocks the popularisation of new systems because developers miss the motivation to port their software due to the lack of users, and users miss the motivation to use the system if there is no usable software. Windows Phone is one of the systems that ended because its developers were unable to overcome this obstacle¹.

Software emulation bridges the gap between different platforms. The most popular emulation layers are the Windows Subsystem for Linux (WSL) and Wine. Microsoft provides the WSL to popularize Windows among software developers used to Linux/UNIX systems. On the other hand, Wine is for people who are dependent on native Windows programs but want to leave the Windows world.

¹<https://www.zdnet.com/article/windows-10-mobile-microsoft-just-put-the-final-nail-in-the-coffin/>

Windows users can run their office suites under Linux and the Linux users can play Windows video games.

1.2 Privilege levels

A typical computer system comprises two parts: privileged kernel space and the non-privileged userspace. This separation is only possible thanks to CPU hardware support.

The early CPUs used to have only one mode in which all running code could do anything. This model had apparent security problems as a faulty user application caused the whole system to crash in some cases. An example of such a system is the MS-DOS or the original MacOS.

Today, the kernel-mode is used to execute only the code that controls the machine. Whatever can work (reasonably fast) in the userspace mode is executed in the userspace mode.

Different operating systems have different opinions on what “necessary to run in the kernel-mode” means. The Linux or FreeBSD have so-called monolithic kernels. Monolithic kernels implement many functionalities. On the other side of the spectrum are microkernels. A microkernel contains only a minimum of functions and shifts as much functionality as possible to the userspace. There is a tradeoff between speed and complexity. The microkernels are typically quite simple (in terms of operating systems design), because more of the complexity is shifted to the userspace programs. The monolithic kernels are more complex, they easily consist of millions of lines of source code, but they speed up the system by reducing the number of context switches. The microkernels are indeed a bit slower, but Liedtke [1] shown that careful implementation of the IPC can considerably reduce the slowdown to acceptable values. Examples of microkernels are the L4 or the HelenOS Spartan kernel. Some kernels try to combine the best of both worlds. They are called hybrid kernels. The Windows NT kernel and the modern macOS Darwin kernel are hybrid kernels.

An entity running in the userspace is called a process (or task on HelenOS). It has the privileges to do anything that does not interfere with the outer world. To do an I/O operation (like reading or writing the file), the process must ask for kernel assistance.

The kernel decides what the process can do based on its privilege model and then multiplexes the access to the underlying resources for all processes.

1.3 Syscalls

From the implementation's point of view, a process cannot use a traditional function call to request a kernel functionality. The function call does not perform privilege mode switching, which is necessary.

Recent CPUs have dedicated instructions for mode switching. Older Intel models achieved it by standard `int` instruction. The specialized instructions on recent CPUs supersede this approach mainly for performance reasons.

Each syscall provides a kernel functionality to the userspace. The set of available syscalls varies between different operating systems. An application must only use the syscalls that are available on its destination platform. In other words: an application that uses the Linux syscalls will not run on Windows and vice versa. The available syscalls and their definitions are part of the so-called kernel ABI (Application Binary Interface).

1.4 Compilation process

In this section, we describe the compilation process that leads from a program's source code to an executable binary.

The compilation process has numerous logical steps. First, the compiler transforms the source code into an object file. A filename ending with the `*.o` suffix usually identifies the object file in the ELF format (section 1.5). The object file contains machine code and symbols. The symbols describe a mapping from the machine code to the source code. A symbol is usually a function name or a name of a global variable.

The compiler creates two tables in the resulting object file. One table contains unknown symbols that are not resolved during compilation, and the second contains exported symbols. Exported symbols are those which are defined in the object file and are not marked static.

Once all the source files are compiled, they are linked together, and the unknown symbols are resolved using the exported symbols from other object files.

In this phase the program is linked with static libraries. Static libraries are object files compiled independently of our program. They become an integral part of the produced binary. Although this is not a technical requirement, they are packed in archives with the `*.a` file extension.

An advantage of statically linked libraries is the possibility of performing link-time optimizations. Another advantage is that the binary is self-contained.

1.4.1 Dynamic linking

Static libraries also have disadvantages. If multiple programs require the same library, the storage contains multiple copies of the same code. Saving storage space might seem irrelevant with today's hard-disk prices, but we should remember that copying the same code also increases the size of container images, leading to their slower deployment.

Another issue relates to security. All statically linked programs must be updated individually when a bug appears in one of the used libraries.

The developer has the choice of avoiding this by using dynamically linked libraries. When a program depends on a dynamically linked library, the linker writes its path in one of the ELF sections. The operating system then loads the library into the program address space every time it executes the process. Shared libraries are ELF files with a `.so` extension (shared object).

A dynamic linker does not have an opportunity to modify the loaded code because it must run reasonably fast and avoid expensive operations.

The compiler creates a table of unknown symbols called the Global Offset Table which is empty before dynamic linking. When a library is linked to the process's address space, the dynamic linker fills the entries in the GOT with the real symbol addresses. This process is called relocation.

The linker creates the libraries as position-independent code to avoid multiple shared libraries linked to the same addresses.

1.5 ELF

The ELF acronym stands for the Executable and Linkable Format (sometimes also Extensible Linking Format). It is a file format used to store executable files (both executables and libraries) on permanent storage. It supersedes the older `a.out` format. Many different operating systems, including Linux, FreeBSD or HelenOS, use it.

The ELF file consists of segments further divided into sections. The segments are essential for the program's execution, while the sections are of greater importance when linking.

Every ELF file begins with an ELF file header. It contains a magic number, the program's entry point, identification of the host operating system and the target hardware architecture. It also points to the program header table, which describes the segments, and the section header table describing the sections.

1.5.1 Execution

The program loader loads the file into the system memory segment after segment. Each segment contains a memory destination and the offset from the file's beginning. Each segment can also be Readable, Writeable and Executable. The permissions are reflected by the program loader when creating the program's memory image. A segment containing the code is mapped as read execute to prevent the program from modifying itself. The data segment, on the other hand, is mapped as non-executable for similar security purposes.

The linker, controlled by a linker script, creates the ELF file in the last stage of compilation. The linker script describes the mapping of sections to virtual addresses.

Chapter 2

Analysis

In this chapter we present both Linux and HelenOS userspace and later we use the knowledge to describe several methods of software emulation on different systems.

2.1 The Linux userspace

We present the relevant concepts of the Linux userspace in this section to lay a foundation for the later chapters of this thesis.

2.1.1 Program loading

The Linux process organization forms a tree. Each process has to have a parent except of the root, which is called the init process. Hence, all processes are the init process's direct or indirect children.

A process creates a child process by duplicating itself using the *fork(2)* syscall. The new process has the responsibility to run the desired binary using some of the *execve(2)* syscall. We present this workflow in listing 1.

The first parameter of the *execve(2)* syscall is the absolute path to the executable file. This file can be either a script, where the first line has the following format - `#!/path/to/interpreter`, or an executable binary file in the ELF format.

We will focus only on the second case, the ELF format, because we are interested in executing binaries and not interpreting shell scripts in this thesis. The *execve(2)* replaces the memory content of the process with a new layout as described in the ELF file in the section 1.5.1.

When the loader prepares the process's memory image and loads all required and available shared libraries, it transfers control to the entry point specified in

Listing 1 Launching a new process in the Linux environment using the combination of the fork and execve syscalls

```
int main(int argc, char **argv){
    pid_t child_pid = fork();
    if(child_pid != 0) {
        printf("Hello from the parent\n");
        waitpid(child_pid, NULL, 0);
    } else {
        const char *const argv_child[] = {argv[1], NULL};
        const char *const envp_child[] = {NULL};
        execve(argv[1], argv_child, envp_child);
    }
}
```

This minimal example program launches and immediately forks. The child process executes the program specified in the first argument. Positional arguments and environment variables are passed as pointer arrays terminated by NULL. The parent prints the "Hello from the parent" string and waits for the child to terminate.

the ELF file. At this moment, the Linux ABI specifies the content of the stack as depicted in listing 2.

The `argc` and `argv` are traditional positional arguments passed later to the `main` function. The environment variables are passed as an array of null-terminated ASCII strings in the "NAME=value" format.

The `auxv` entries form the auxiliary vector, which passes runtime information from the kernel to the userspace program. The declaration of the `auxv_t` structure is in file `/usr/include/elf.h` and an example is in listing 3.

2.1.2 Syscalls

An application programmer rarely encounters the need to manipulate syscalls directly. The application programmer uses the standard library functions that wrap the syscalls. Section two of the Linux manual pages describes these functions.

We need to look one step deeper in this thesis, focusing on `x86_64` architecture. A syscall wrapper always ends in an assembly block that executes the `syscall` instruction. This instruction transfers the program control to an address specified in the `IA32_CSTAR_MSR` while also switching the CPU privilege mode.

In contrast to the interrupt handlers, all syscalls share one kernel entry point. The userspace writes the desired syscall number and its parameters into general

Listing 2 The content of the process stack at the moment when control is transferred to the entry point

| position | content | bytes (comment) |
|------------------|-------------------------------|-----------------|
| stack pointer -> | [argc = number of args] | 4 |
| | [argv[0] (pointer)] | 4 |
| | [argv[1] (pointer)] | 4 |
| | [argv[..] (pointer)] | 4 * x |
| | [argv[n - 1] (pointer)] | 4 |
| | [argv[n] (pointer)] | 4 (= NULL) |
| | [envp[0] (pointer)] | 4 |
| | [envp[1] (pointer)] | 4 |
| | [envp[..] (pointer)] | 4 |
| | [envp[term] (pointer)] | 4 (= NULL) |
| | [auxv[0] (Elf32_auxv_t)] | 8 |
| | [auxv[1] (Elf32_auxv_t)] | 8 |
| | [auxv[..] (Elf32_auxv_t)] | 8 |
| | [auxv[term] (Elf32_auxv_t)] | 8 (= AT_NULL) |
| | [padding] | 0 - 16 |
| | [argument ASCIIZ strings] | >= 0 |
| | [environment ASCIIZ str.] | >= 0 |
| | [end marker] | 4 (= NULL) |
| | < bottom of stack > | 0 |

Taken from the
<http://articles.manugarg.com/aboutelfauxiliaryvectors>

Listing 3 An auxiliary vector entry structure and some example types.

```
// file : include/uapi/linux/auxvec.h
/* Symbolic values for the entries in the auxiliary table
   put on the initial stack */
#define AT_NULL 0 /* end of vector */
#define AT_IGNORE 1 /* entry should be ignored */
#define AT_EXECFD 2 /* file descriptor of program */
#define AT_PHDR 3 /* program headers for program */
#define AT_PHENT 4 /* size of program header entry */
#define AT_PHNUM 5 /* number of program headers */
#define AT_PAGESZ 6 /* system page size */
#define AT_BASE 7 /* base address of interpreter */
#define AT_ENTRY 9 /* entry point of program */
#define AT_UID 11 /* real uid */
#define AT_EUID 12 /* effective uid */
#define AT_GID 13 /* real gid */
#define AT_EGID 14 /* effective gid */
#define AT_PLATFORM 15 /* string identifying CPU */
#define AT_HWCAP 16 /* arch hints at CPU capabilities */
#define AT_CLKTCK 17 /* times() increment frequency */
/* AT_* values 18 through 22 are reserved */
#define AT_SECURE 23 /* secure mode boolean */
#define AT_RANDOM 25 /* address of 16 random bytes */

#define AT_EXECPN 31 /* filename of program */

// file : /usr/include/elf.h
typedef struct
{
    long int a_type; /* Entry type (one of the AT_*) */
    union
    {
        long int a_val; /* Integer value */
        void * a_ptr; /* Pointer value */
        void (* a_fcn) (void); /* Function pointer value */
    } a_un;
} auxv_t;
```

Taken from the Linux header file include/uapi/linux/auxvec.h

purpose registers. The calling convention is given by the kernel syscall ABI, and it is very similar to the SystemV ABI. The exceptions are given by the fact that the syscall instruction destroys registers r11 and rcx. More information on this is in the *syscall(2)* manual page. For the Linux kernel, the syscall number and the return value is in the rax register. The arguments are in the following order in registers rdi, rsi, rdx, r10, r8 and r9.

The kernel exports the numbers of its syscalls in a header file `sys/stdarg.h`.

According to the Linux *intro(2)* manual pages the library wrappers are described in DESCRIPTION, and the raw syscall details are clarified in the NOTES section.

Examples of syscalls

getpid(2) The `getuid` syscall is an example of a very simple syscall. It has no parameters, it returns the process id as the only value, and it always succeeds. HelenOS provides the syscall `task_get_id` with equivalent functionality.

readlink(2) The `readlink` syscall is a bit more complicated. It receives a path to a symbolic link and returns the destination of the link. It returns either the length of the result placed into the buffer or negative value of the error constant on error. Similar result is achieved in HelenOS by short communication with VFS server.

fork(2) The `fork` syscall is one of the most complicated syscalls in UNIX/Linux. It creates a new process by duplicating the current one. It has no direct equivalent in HelenOS.

2.2 Introduction to HelenOS

The HelenOS operating system is a multi-server microkernel operating system that originates at the Faculty of Mathematics and Physics at Charles University. Even though HelenOS contains many exciting concepts, we only describe those relevant to our thesis. In section 2.2.1 we describe the threads and tasks in HelenOS, in section 2.2.2 we describe program loading, and in section 2.2.3 we describe the architecture of syscalls.

2.2.1 Threads and tasks

The basic building blocks of HelenOS userspace are tasks and threads. For simplicity, we consider the HelenOS tasks to be equivalent to UNIX processes and use the terms interchangeably in this thesis. Although there are differences in

the implementation, the basic idea remains the same. A task consists of an address space, opened file descriptors and other metadata. It is the environment for running threads.

Threads are also similar on UNIX and HelenOS. They are “basic executable entities with some code and a stack”. A thread must always run in the context of some task.

These concepts are described in depth in the HelenOS design document on the official website¹.

2.2.2 Program loading

HelenOS application binaries and shared libraries are stored as ELF files.

HelenOS does not provide an equivalent of the UNIX *fork(2)* syscall. If a new process is being created, the parent program invokes a clean new instance of a *loader* server. The loader server is then instructed via IPC commands to load the required binary into memory, set up the environment, load shared libraries and then jump to the application’s entry point. In other words, the only binary loaded by the kernel is the loader server which then transforms itself.

The important fact to note here is that both the loader and the target application run in the same task. This also means that they share one address space, therefore the loader must be in a non-standard memory location to not interfere with the application code. We will use this knowledge later when constructing the prototype application.

2.2.3 Syscall architecture

The HelenOS kernel is indeed very minimal, respecting the generic microkernel principle as presented by Jochen Liedtke [1]. Most of the services are therefore not provided by the kernel itself but by the userspace applications called servers.

The primary responsibility of the kernel is to handle the Inter-Process Communication (IPC) and provide a reliable communication channel between the client and the server applications. Note that in this design, one application may be both client and server. For example, the filesystem server can be a client of the HDD driver server[2].

¹ <http://www.helenos.org/doc/design.pdf>

2.3 Windows Subsystem for Linux

The initial Microsoft Windows NT kernel, released in 1993, was designed to support multiple subsystems such as POSIX² or OS/2³. A subsystem was meant to be an interface between the userspace application and kernel. These early subsystems were userspace modules presented to the application as shared libraries. The default subsystem was called Win32, and it was and still is used to run native Windows applications.

Over time this design was used to create various subsystems to run unmodified applications from different systems. One such attempt was Microsoft's POSIX.1 subsystem which the Softway Systems company later used as a foundation for the OpenNT project. The OpenNT project aimed to bring a somewhat complete UNIX experience to the Windows/NT kernel users[3]. This project never gained wider popularity and was abandoned around 2016⁴.

Around 2013, a group of Microsoft research scientists leveraged the subsystem capability of the NT kernel in a project called Drawbridge⁵. Its purpose was to analyse the possibilities of the sandboxing mechanism to increase the security of operating systems and lower the risks which arise when running software downloaded from untrusted sources. As part of this effort, the group also implemented a simple POSIX environment and ran some Linux applications[4].

This research project found its way to Windows 10 in 2016, and now it is known under the name WSL (Windows Subsystem for Linux). We should note that at the time of writing this thesis, the original WSL is superseded by the WSL2. Although serving a similar purpose, the WSL2 uses a fundamentally different approach and the results of the Drawbridge project are no longer the core of this technology⁶. We always refer to the original WSL implementation in this thesis.

The whole WSL system does more than just translating syscalls. It manages the whole "session" of a running Linux system. The session manager is called the "LXSS Manager service". It keeps track of running Linux processes and manages their resources. figure 2.1 depicts this arrangement.

² <https://web.archive.org/web/20160304201832/http://www.microsoft.com/resources/documentation/windowsnt/4/workstation/reskit/en-us/poscomp.mspx>

³ <https://web.archive.org/web/20160506054822/http://www.microsoft.com/resources/documentation/windowsnt/4/workstation/reskit/en-us/os2comp.mspx>

⁴Last related post by the OpenNT developer Stephanos is dated 12th November 2016. <https://web.archive.org/web/20210119144548/https://stephanos.io/archives/21>

⁵ <https://www.microsoft.com/en-us/research/project/drawbridge/>

⁶See the <https://docs.microsoft.com/en-us/windows/wsl/compare-versions> for a thorough comparison

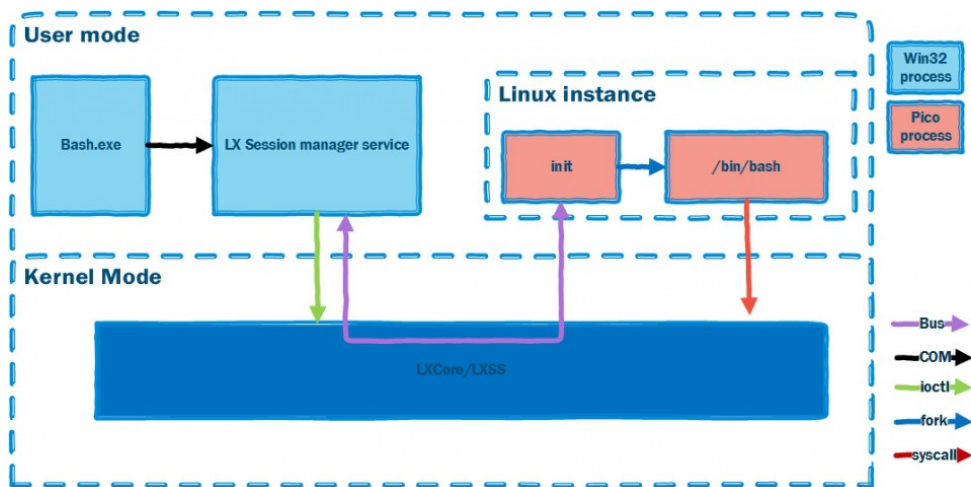


Figure 2.1 The WSL architecture (The image is from the Microsoft documentation <https://docs.microsoft.com/en-us/archive/blogs/wsl/windows-subsystem-for-linux-overview>)

2.3.1 Pico process

Pico processes are the foundation of the WSL functionality. They are one of the results of the Drawbridge research.

A Pico process is specialised so that when its code executes a syscall, the kernel does not handle it. Instead the kernel redirects it to the associated kernel drivers as presented in figure 2.2. The drivers for the Linux subsystem are called *lxss.sys* and *lxcore.sys* respectively. The syscall parameters are unmarshalled, possibly some preprocessing is done, and then the syscall is passed to the NT routines for further processing. The result of a syscall goes back to the userspace through the same driver, which manages the result translation in the other direction as well.

2.4 Syscall User Dispatch

The Windows NT kernel is not the only one providing a foundation for system emulators by its Pico processes. The Linux kernel provides similar functionalities under the name Syscall User Dispatch.

The high-level trap-and-emulate concept remains the same, but the implementation differs. The syscall is dispatched back to the userspace in contrast to WSL, where the syscall is dispatched to the kernel driver.

The applications that use this API are called “Multiple-Personality applications” in the Linux terminology.

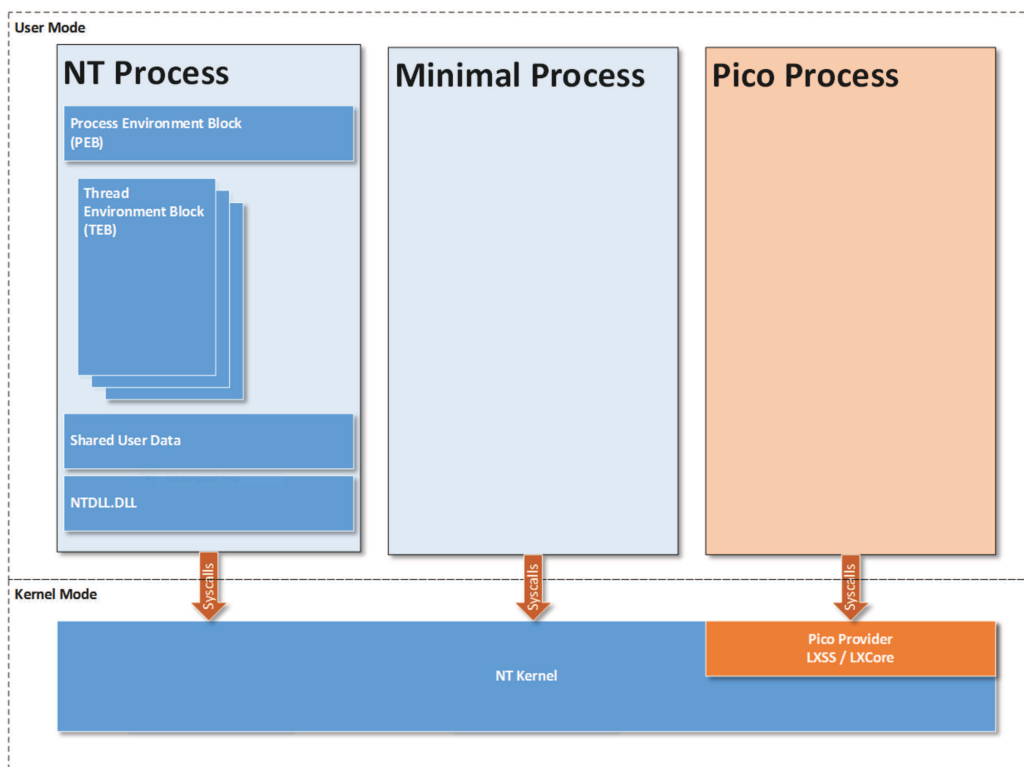


Figure 2.2 The comparison of traditional NT process and Pico process. (The image is from the Microsoft documentation <https://docs.microsoft.com/en-us/archive/blogs/wsl/pico-process-overview>)

The application uses a native syscall to configure a memory region from which the kernel will interpret the syscalls. Syscalls performed anywhere in the rest of the application's address space are dispatched to the application-defined handler.

The complete documentation of this Linux kernel feature is in the Kernel documentation project⁷.

2.5 Wine

Wine is a compatibility layer for running Windows applications on UNIX desktops. At the moment, Wine is compatible with Linux, macOS and various BSDs.

The program userspace is slightly more complicated on Windows than on UNIX-like operating systems. The Windows NT kernel's Native API is undocumented, and the existing userspace programs communicate with the kernel using one of the subsystems.

The subsystem is a userspace DLL (Dynamically loaded library, the Windows equivalent of a UNIX shared libraries) that provides a documented set of APIs. The default subsystem is Win32, and it is used to run native Windows applications. Early Windows NT based operating systems offered further subsystems for POSIX and OS/2 operating systems, but were discontinued over time.

When loading a program, the Windows kernel analyses the application's executable and decides what subsystem will run it.

The Wine project takes advantage of this arrangement and allows execution of the Win32 applications on top of the UNIX compatible kernel.

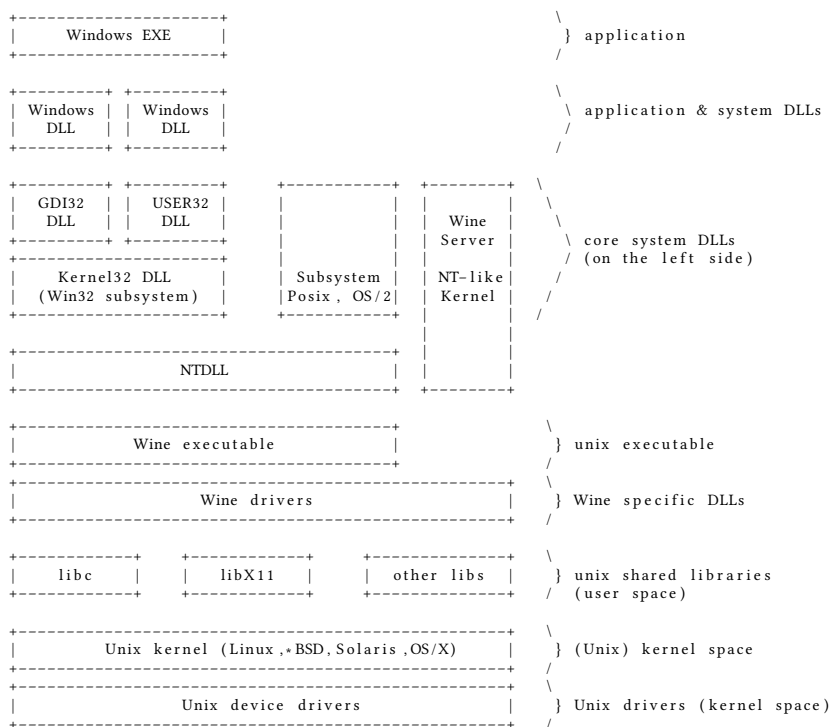
If the user attempts to run a Windows application with Wine, the first thing that is loaded is the Wine executable. The Wine executable then loads the Windows application and its dependencies in the form of DLLs. The DLLs are either official or provided by the Wine project.

When a user executes a first Windows application using Wine, Wine executes a daemon that manages the Windows instance⁸. This daemon is called the Wine server. All other Windows applications are then connected to the same session that allows their interoperability. It manages Windows registers and other tasks that are normally managed by the Windows kernel itself.

⁷ <https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html>

⁸ https://wiki.winehq.org/Wine_Developer%27s_Guide/Architecture_Overview

Listing 4 The architecture of a Wine emulated Windows process



The “image” is from the official Wine documentation https://wiki.winehq.org/Wine_Developer%27s_Guide/Architecture_Overview

2.6 libposix

Not only Windows uses the shared library approach. Most Linux programs depend on the standard C library that provides syscall wrappers and other functionalities. This library is loaded into the application's address space by the kernel loader, and the vast majority of syscalls is routed to the kernel through this code. A typical application will never call a syscall directly. It uses higher-level wrappers instead.

Several standards define the API of the C library. The standard C library, sometimes called the ISO C library, is a subset of the POSIX C library that describes additional functionalities, such as `fork`, related mainly to process management and filesystem access.

A well-written application that depends solely on the library's functionality granted by the standard is portable by definition. It can be ported to another platform by simply providing it with a different runtime library shared object (as described in section 1.4.1) conforming to the same standard. This approach is similar to what Wine does for Windows applications on Linux.

Therefore, the HelenOS community developed a replacement for a subset of the POSIX C library called `libposix`. If a HelenOS user executes an application, this library is linked to it, and it translates the calls to HelenOS native syscalls. This works well unless the application needs to perform some system-related syscall such as `fork(2)`. The `fork(2)` lacks the appropriate equivalent in the HelenOS system, and its support requires a different approach.

HelenOS developers created a project called Coastline to automate the recurring tasks that are necessary to port the software. This project is described in the official HelenOS documentation⁹.

As the required functional subset of the library grew with the rising complexity of the ported software, it became evident that either full implementation of the POSIX C library will be necessary or that a different approach must be taken. The different approach is to take the existing C library and change only the OS interface.

2.6.1 liblinux

The C library is a complicated piece of code. Multiple different implementations already exist, each developed for a different purpose. The most notable implementations are the *GNU libc*¹⁰ intended for GNU/Linux, the *MSVCRT.DLL*¹¹ being

⁹ <http://www.helenos.org/wiki/PortingSoftware>

¹⁰ <http://www.gnu.org/software/libc/>

¹¹ <https://docs.microsoft.com/en-us/cpp/c-runtime-library/c-run-time-library-reference>

part of the Microsoft Visual C compiler or the *musl*¹² that aims to be a lightweight replacement for *glibc* on Linux systems. Writing its entirety to port applications to HelenOS would be a very tedious task.

Hence, it is desirable to take some existing implementation of the C library and change only the parts that differ between HelenOS and the original library hosting platform. Modifying the GNU *libc* would require a heroic dedication, and so HelenOS developers choose to stick with the *musl* library. When compared to the *glibc*, the *musl* library provides the most straightforward implementation of the required functionality.

Only syscall wrappers were modified to mimic the original semantics of native HelenOS syscalls. This modified functionality was put into the library called *liblinux*.

Porting the application to HelenOS then required to compile a *musl* with *liblinux* and then link the Linux application with such modified *libmusl*.

2.7 Other implementations

The previous sections do not provide a comprehensive enumeration of all existing implementations.

Other notable implementations include the *linux.ko* kernel module that provides the Linux emulation layer for the FreeBSD kernel, *MkLinux*, *Lites* and the *GNU/Hurd*.

The *MkLinux* [5] and *Lites* [6] were attempts to run Linux and BSD userspace respectively on top of the Mach [2] kernel. The *MkLinux* effort was backed around 1997 by the Apple Computer to bring the Linux operating system on the contemporary PowerPC based Macintosh computers.

The GNU/Hurd [7] was intended to become the ultimate operating system of the GNU movement, but once Linux acquired this position after 1991, the interest in Hurd became marginal. Over time the GNU/Hurd was ported to L4 based kernel, the Carnegie Mellon University Mach kernel and one microkernel was even developed solely for Hurd. Although Hurd currently provides a usable version based on the Debian distribution, it is not as widespread as intended by its authors.

¹² <https://musl.libc.org>

The L4Linux project presents an effort to run the Linux kernel on top of the L4 microkernel. The project is still maintained, but it is rather a research project than a mainstream system. The L4Linux claims full binary compatibility with unmodified Linux binaries. It is achieved by a combination of the Wine/WSL approaches [8]. If the application is dynamically linked, it is provided with the patched C library, which translates library calls to native L4 IPC service requests. Syscalls are trapped by the kernel for statically linked applications and returned to the userspace handler, which then emulates the call using the L4 native approach¹³.

The Genode project used to have a UNIX execution environment called Noux¹⁴. In the latest versions, the Noux is merged with another component of the Genode framework. This environment comprises build and execution parts. The build environment is similar to the HelenOS Coastline project. The binary can run as a native Genode process after being build in this environment. The execution environment is a userspace UNIX server providing UNIX kernel functionality to Noux processes. A Noux process maintains a session on the UNIX server using the RPC mechanism. Each hosted program is linked against a special Noux libc plugin that catches all libc calls that would typically result in a system call. It then transparently forwards the syscall to the UNIX server.

¹³ <https://www.cs.cornell.edu/courses/cs614/2007fa/Slides/kernel%20architectures.pdf>

¹⁴ https://genode.org/documentation/release-notes/11.02#Noux_-_an_execution_environment_for_the_GNU_userland

Chapter 3

Design and implementation

This chapter introduces the extension of HelenOS kernel and a userspace application that we developed to run unmodified Linux binaries on HelenOS.

3.1 Program lifecycle

First, a task memory image is created. This is usually done by loading an executable file from some storage, but it also involves the *fork(2)* syscall. An environment is prepared for the code execution. Then control is transferred to the task's code. This code executes until it needs to interact with the outer environment. Such interaction may be network communication, reading or writing files, knowing the time or just the intent to finish (i.e. the *exit(2)* syscall).

When need for outer interaction occurs, the program performs a syscall by issuing a special instruction to the CPU. This instruction transfers control to the kernel. The kernel then decides what to do based on the syscall parameters passed in the registers or the stack. Once the kernel finishes processing the syscall, it restores the CPU state and returns control to the process. This is performed in a loop until the task termination.

We may identify the possible problems occurring when we try to launch a non-native application in the above.

The process may fail in the very first step if the executable file is in an unknown format. This is not the case for HelenOS/Linux compatibility as both systems use the same ELF format (section 1.5), but generally, it might happen (e.g. both macOS and Windows use different formats).

In the second step, the program loader prepares an execution environment. Linux programs expect to find an auxiliary vector on the stack while HelenOS uses the Program Control Block. The purpose is mostly the same – to pass some data about the environment to the application code – but the data format differs.

The code execution itself generally does not depend on the operating system and should be unaffected¹.

Most of the complexity is hidden in the syscalls. A number identifies each syscall. These numbers are different for each operating system and platform and are part of the kernels ABI. If the process requests a syscall that is unknown or has different semantics, it will lead to termination or undefined behaviour.

These are the issues we want to address in this work. Regarding the principle of minimality [1] we also aim to add as little complexity as possible to the HelenOS Spartan kernel.

3.2 Architecture

In chapter 2 we described different ways of the non-native software execution: the Wine approach with patched runtime libraries for dynamically linked programs and the WSL approach with trap-and-emulate processing of the syscalls. The trap-and-emulate has two variants based on whether the emulation is done in the kernel directly (WSL described in section 2.3) or forwarded back to the userspace handler (Userspace Syscall Dispatch described in section 2.4 or L4Linux described in section 2.7).

The Wine approach was already implemented in the HelenOS environment by the libposix and liblinux (section 2.6.1). Our work focuses on implementing the WSL trap-and-emulate method instead. We do not implement the in-kernel emulation to avoid polluting the micro-kernel with unnecessary functionality. With this in mind, only the L4Linux way remains.

As described in the study of Wine (Windows server, section 2.5) or Genode Noux (UNIX server, section 2.7), maintaining the illusion of an entire system may require an additional component. The functionality missing in the host operating system determines the responsibilities of the server. Implementing a similar server will likely be necessary in the future, but we do not aim to provide a complete system emulation in this work.

Our prototype implementation consists of a thin emulation layer inside the process address space and a kernel modification that allows the trap-and-emulate functionality for syscalls.

We divide the practical implementation into three parts: The first part, which is described in section 3.3, introduces the extension to the kernel to provide a trap-and-emulate functionality for userspace syscall dispatch. We call it the “The

¹This statement ostentatiously ignores the existence of instructions whose behaviour is controlled by the kernel or switches the CPU execution modes in userspace. These instructions are all defined in the Linux API, but HelenOS lacks the support, and so we do not consider them here.

Listing 5 Thread context structure

```
typedef struct masq {
    uint64_t rax, rbx, rcx, rdx;
    uint64_t rsi, rdi, rbp, rsp;
    uint64_t r8, r9, r10, r11;
    uint64_t r12, r13, r14, r15;
    uint64_t fsbase, gsbase;
} masq_t;
```

Masquerade API”. The second part, presented in section 3.4.2, contains a Linux compatible program loader. In the third part, which is described in section 3.4.3, we use the API provided by the kernel extension and implement the lifecycle emulation.

3.3 The Masquerade API

Since both the application and the emulator run in the same task, we need to distinguish the native syscalls performed by the emulator and the non-native performed by the application. Further, we need to transfer control to the emulator after a non-native syscall is performed. This approach is usually called trap-and-emulate.

We introduce a new native syscall named *masq*. This is an abbreviation for *The Masquerade API* which is how we call the provided functionality. We find this name pretty accurate. The syscall switches the kernel to the mode in which it pretends to be someone else.

The idea is that the application invokes the syscall with some description of the required CPU state. This description has the form of a structure describing register values. As such, this structure is tightly coupled with the underlying architecture. The actual implementation for the x86_64 architecture is in listing 5.

HelenOS syscall handling starts after the syscall instruction is executed in the userspace. This transfers control to a predefined procedure in the kernel. This procedure usually starts with an assembly block doing the low-level heavy-lifting of swapping segment registers or pushing the userspace registers on the stack.

Once all this is ready, the generic C routine is called, the syscall handler procedure is selected from the table of handlers and called with syscall arguments. The return code of this handler is then passed back to the assembly block through the generic routine. The assembler code then prepares the CPU context for the userspace and switches the CPU mode.

In the simplest case, adding a new syscall is just adding the respective handler to the syscall handlers table. It is almost plug-and-play unless the syscall is supposed to somehow alter the syscall handling procedure. Our *masq* syscall does exactly this.

The kernel initially pushes the userspace registers to the stack and pops them back after returning to the userspace. The core functionality of our syscall is hence to swap the contents of the provided `masq_t` structure with the registers on the stack and mark the actual thread as being masqued. The asm block before the `sysret` instruction pops the registers from the stack, and the `sysret` instruction will transfer control back to the userspace. Since the instruction pointer is one of the popped registers, it will correctly transfer control to the new location.

When the masqued thread performs the next syscall, the content of the registers is swapped again, which causes the following `sysret` instruction to transfer control back from where the *masq* syscall was initially called.

In the end, the semantics is similar to `longjmp/setjmp`.

For performance reasons, the kernel did not restore all general-purpose registers but only a well-defined subset. This optimization led to a situation in which the masqued context was not completely set to the thread; hence the application observed unexpected changes in the register values.

3.3.1 Segment registers

The segment registers are part of the legacy of the Intel x86 architecture.

Their original purpose was to divide available memory to isolate the running applications. Their semantics changed with the transition from 16 bits CPUs like the Intel 8086 to 32bit CPUs with protected mode. In this mode, memory management was monopolized by the paging mechanism, and most of the segment registers lost their meaning in the current x86_86 architecture.

Two notable exceptions are the FS and GS registers used to implement the thread-local storage in both applications and kernels. The modern processors simplified the manipulation by providing access to the loaded value of the segment base by the FSBASE and GSBASE MSR registers.

The Linux kernel provides a syscall *arch_prctl* which allows the userspace to set its values in these registers.²

HelenOS kernel misses this functionality which was a problem for running the Linux applications. It is possible to make the values of `fsbase` and `gsbase` members of the `masq_t` structure and set them in the kernel before moving to the masqued code. The problem is that since the HelenOS kernel itself does not

²Even more modern processors provide special instructions for reading and writing these MSR registers without privileged `rdmsr/wrmsr` instruction

expect the userspace to do this, it does not restore the value of these registers after the syscall, but more importantly, neither after interrupt handlers.

To make this emulation reasonably working we also had to modify the interrupt handler to restore the `fsbase` register. Although this is less than ten rows of assembler code, it adds a `wrmsr` instruction to the code hot-path. The `wrmsr` instruction is implemented in microcode, does dozens of privilege checks, and its execution takes up to hundreds of CPU cycles. Adding it to the interrupt handler might thus bring some performance penalty. Even worse, the interrupt handler does not have an easy way of knowing that it has been triggered from a masked thread, and thus, the `fsbase` restoration is done unconditionally.

3.4 Linux emulator

Extending the kernel is only the beginning. We need some client application to use this extension.

The idea is that for each emulated system, there will exist one such application. In this work, we implement only the Linux emulator and only for the `x86_64` devices. The responsibility of this application is to load the guest application and then emulate the syscalls.

3.4.1 Memory layout

The emulator binary will share the address space with the guest application. For this reason, the emulator app must be linked somewhere so the guest application does not interfere with it.

Luckily for us, the memory layout for HelenOS and Linux is similar, at least on the `x86_64` architecture. Although the `x86_64` processors are 64bit at first sight, the current generation supports only a 48-bit wide address bus. The CPU requires that each valid address is in the canonical form, meaning that all not-implemented bits (i.e. 63 to 48) are of the same value. These facts divide the entire virtual address space into three parts.

The lower area is traditionally dedicated to userspace programs, while the kernel uses the upper one³. This way, the kernel is mapped to each process's address space and syscalls are not required to switch the address space mapping. Such an approach leads to a performance gain since the TLBs are not flushed, but its use is somewhat more complicated after the Meltdown [9] and Spectre exploits.

³ https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

In this arrangement, the userspace has 2^{48} bytes of virtual memory available. There is still enough memory for the guest application with the expected upper limit of 1MiB on the whole emulator binary.

3.4.2 Program loader

The program loader itself is built using the existing HelenOS C library. The library has a module for loading ELF files to memory. We will use this functionality since the Linux binaries are also ELF files.

The HelenOS C library also provides the *rtld* (run time loader), which can load shared libraries to the process memory. Our prototype implementation does not use a run time loader service, and therefore, it can load only statically linked Linux binaries.

Once the executable and required libraries are present in memory, the loader must prepare the execution environment.

First, the loader must establish a stack for the emulated application. The current prototype uses a statically allocated area in the emulator data section, but more sophisticated solutions with a stack protector are also possible. The stack size should be similar to one of a real Linux system. This value can be obtained on a Linux system by running *ulimit -s*, and on present time computers, it is about 8192 kilobytes. When the stack overflows, the emulator data might get corrupted, and the whole program likely crashes.

Once the stack is created, the loader initializes the data structures present on the stack as described in listing 2.

Finally, we prepare expected values for the general-purpose registers, namely the instruction pointer and the stack pointer.

We called our prototype loader *lilo* (Linux loader⁴). The execution of the command line tool is similar to tools like *time(1)* or *strace(1)*. The first argument is a path to the desired Linux binary, and all other arguments are passed as arguments to the binary.

At the moment, the *lilo* itself does not support any runtime options.

Example execution is in listing 6.

3.4.3 Syscall emulation

Once the program execution environment is ready, the loader uses the Masquerade API to switch to the emulated context.

Control transfers back to the loader after the emulated program perform a syscall instruction. All of this looks like a single function call from the emulator's

⁴The name is inspired by the legacy Linux bootloader

Listing 6 Example output of the HelenOS Linux loader

```
/ # lilo busybox uname -a
Linux HelenOS PC 4.19.0-14-amd64 #1 SMP Debian 4.19.171-2
(2021-01-30) x86_64 GNU/Linux
/ #
```

This command launches unmodified Linux binary busybox in the HelenOS operating system. From the output we can see that the emulated personality presents itself as the Debian 10 with kernel version 4.19. The nodename field is set to HelenOS PC.

perspective.

After the *masq* syscall succeeds, the emulator decides what to do based on the content of the registers.

The emulator contains a table of pointers to the syscall handling functions indexed by the Linux syscall id. Most syscalls except a few exceptions are called by simply indexing to the array and calling the proper function. The exceptions are *exit(2)*, *exit_group(2)* and *vfork(2)*.

Some syscalls, such as the *getpid(2)*, are easy to emulate, while others are rather complicated. The emulation is performed by translating the calls to native HelenOS IPC services. This workflow is depicted in the figure 3.1.

The emulation partially reuses the code from the liblinux library⁵ (see section 2.6.1). The liblinux library already provided mapping between Linux syscalls (although only for the 32 bit ABI) and HelenOS native ones. However, it was never merged into the upstream, and since the work was done a few years ago, the HelenOS C library has changed, therefore the code has to be updated to reflect these changes.

Filesystem API

The Linux/UNIX filesystem API is built on top of the file descriptors. A file descriptor is a process's handle for opened files.

Although HelenOS uses a similar concept, it uses a different API for I/O on standard input and output. The emulator maintains a separate list of opened file descriptors and their types and a structure with pointers to proper I/O functions to hide this difference. When the emulated program opens or closes the file, the respective filehandle is added or removed to/from the list.

This is the code taken from the liblinux library.

⁵ <https://github.com/vhotspur/helenos/tree/liblinux/ospace/lib/inux>

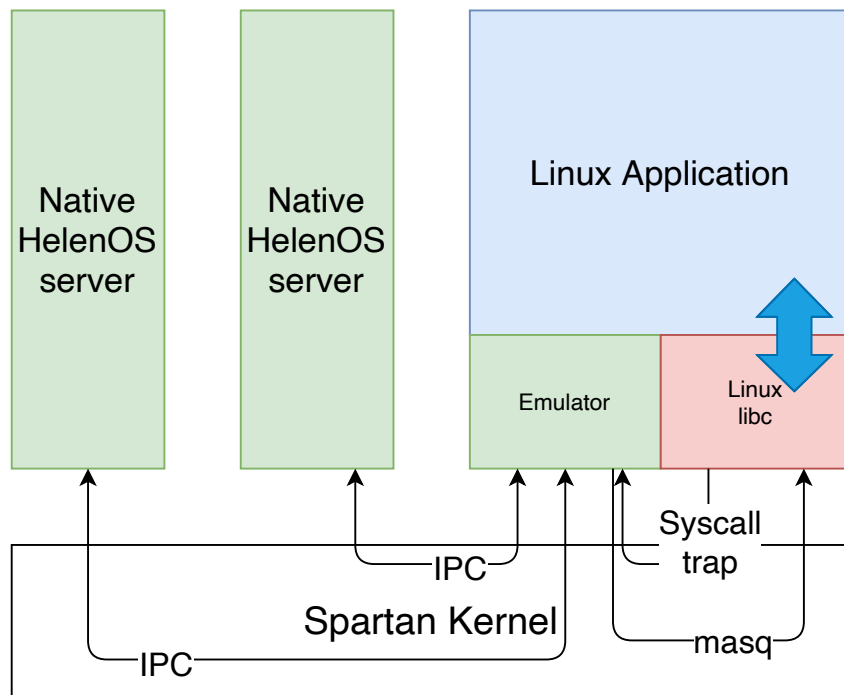


Figure 3.1 Linux emulator lifecycle

vfork

Our initial analysis stated that emulating *fork(2)* syscall is near to impossible with the standard HelenOS userspace tools.

We wanted to run an unmodified GCC compiler inside the HelenOS to demonstrate the functionality of our implementation. The GCC itself is only a composer that launches child processes such as the actual *cc1* C compiler, the *as* assembler and the *ld* linker. The child processes, in turn, do all the hard work of producing the executable from the source code.

This seems like an exact use case of the *fork/exec(2)* combo. Luckily for us, the GCC developers decided to substitute the *fork* syscall with a *vfork*. The *vfork* behaves similarly, except that it does not duplicate the parent process memory space unless the *execve* is executed. This also includes the stack. The child cannot modify the stack between its birth and a call to the *execve(2)*.

This fact greatly simplifies the emulation. Since the child is already aware of its limitations, the emulator can duplicate the process state after the *vfork* is issued. Then it will speculatively jump back to the emulated process with *vfork* return value of 0, signalling that this is the child. Suppose the next syscall is *execve*. In that case, we emulate this by launching a new instance of a *lilo*

program with correct argument values so that the child is correctly executed⁶.

3.5 Considerations

Memory protection

We do not have to care about the memory protection of the kernel. The emulator application runs in the userspace already, and standard protection mechanisms apply here.

However, we should consider that the emulator and guest application run in the same address space and with equal privileges. Theoretically, the guest application may tamper with the memory of the emulator. In that case, the behaviour becomes undefined.

To find the correct handling of such a situation, we should discuss how such a situation may occur.

The guest application may use only the memory acquired with either *mmap(2)* or *brk(2)* syscalls. Our application is in charge of dispatching these syscalls, and therefore, unless our emulator decides to commit suicide, such a memory would not interfere with the emulator memory.

If the guest application uses, intentionally or not, some memory that was not acquired correctly, it can fail, which is consistent with the native environment behaviour.

FPU context

The emulator application is compiled with `-mgeneral-regs-only` and therefore does not use any FPU registers nor any vector registers. Therefore, it is not necessary to care about the FPU context as it is correctly handled by the preemption mechanism of HelenOS itself.

Other incompatibilities

The incompatibility between Linux and HelenOS is not only given by different syscalls. Linux applications might further expect the UNIX filesystem hierarchy, which is not present on HelenOS. The problem is with well known files like `/dev/random`, `/dev/null` or `/etc/passwd`, which must be treated differently when the application tries to open them.

⁶The combination of *vfork* and *execve* is generally not recommended, and the manual pages recommend replacing the combo with a *posix_spawn* syscall.

License

Our prototype implementation emulates the API/ABI of the Linux kernel. The interface consists of syscall numbers and some structure definitions.

The definitions are present in the kernel header files published under the GPL license. HelenOS itself is developed under the BSD license, and so these headers cannot be directly copied to the source directory of the syscall emulator. Our code provides only symbolic links to the system locations to overcome this obstacle; therefore, none of the GPL protected files is present in our code.

It limits building the HelenOS system to Linux systems, and a more robust solution should be adapted in the future.

3.6 Evaluation

Our prototype implementation emulates over thirty syscalls. Some of them are implemented thoroughly, some of them are just mockups.

The not emulated syscalls varies in complexity. Many of the file system related will just translate the calls to HelenOS native APIs. Socket related syscalls are more complex, but also possible to implement using similar translation methods. User related syscalls are going to remain mockups, since HelenOS does not have the concept of users at the moment. Pipes and POSIX signals will need either kernel support or emulation using some UNIX server. Although Linux has over three hundreds of syscalls, some of them are just shortcuts (e.g. *sendfile(2)*) or slightly updated versions of standard syscalls (e.g. *dup2(2)* and *dup3(2)*) which does not add significant complexity.

The evaluation began with GNU/coreutils. We managed to run the *cat*, *cut*, *base64*, *cksum*, *head*, *tail*, *tac*, *wc*, *uniq*, *sort*, *uname* and *seq*. These programs do not require a lot of environment support. Most of them only read files and print the output.

We achieved similar results by running the *busybox*⁷ binary with respective subcommands. The *busybox* binary contains several shell subcommands. This gives us hope that we can run a full fledged shell instead of the limited HelenOS *bdsh*. This would however require adding a complex functionality such as pipes, sockets or *ttys* to our emulator. Since there is no clear mapping between these requirements and HelenOS APIs, we decided to not support the interactive terminals.

In the end, the implemented syscall subset was enough to run the unmodified GCC compiler and its child processes correctly. The GCC executes numerous

⁷ www.busybox.net

helper programs of which we had only the *cc1* C compiler, but this was sufficient to translate the C program to assembler.

Conclusion

In this thesis we successfully reached all of our goals. We analyzed the existing solutions of syscall emulation and application binary portability. We proceeded by building a prototype implementation of a Linux syscall emulator and program loader. We were able to execute simple statically linked programs compiled for Linux on the HelenOS system.

The work presents only minor changes to the HelenOS kernel without significantly increasing its complexity. As a result our contribution is not expected to harm the existing kernel functionality in any way.

The work lays ground for future implementations of syscall emulation layers for various operating systems.

If merged to the HelenOS upstream, our contribution presents new opportunities in practical usages of HelenOS.

Future work

The API of mature operating systems, either Linux, BSD or Windows, evolves over time. As a result, the APIs are clobbered with legacy options, containing multiple syscalls for achieving similar results – similar, but still with subtle differences. This fact poses a great challenge for implementing the complete emulation layer. A genuine-looking API will certainly require a lot of improvements to the system emulator code.

So far we discussed only the interfaces for a single process. Further extensions may implement some kind of UNIX server that will take care of interconnecting different Linux processes and therefore providing them with a feeling of a fully operational Linux system. This concept is present in both WSL and Wine.

Such a server could be afterwards extended by its own Virtual File System instance to provide a separate filesystem root for the Linux runtime.

An interesting task would definitely be to manage to run Windows applications inside Wine on the Linux compatibility layer inside HelenOS. However, this requires many improvements in the current emulator code as the Wine uses some advanced functionality of the Linux kernel.

Bibliography

- [1] Jochen Liedtke. “On micro-kernel construction”. In: *In SOSP*. 1995, pp. 237–250.
- [2] Mike Accetta et al. “Mach: A New Kernel Foundation for UNIX Development”. In: 1986, pp. 93–112.
- [3] Stephen Walli. “OPENNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem”. In: *Large-Scale System Administration of Windows NT Workshop (Large-Scale System Administration of Windows NT Workshop)*. Seattle, WA: USENIX Association, Aug. 1997. URL: <https://www.usenix.org/conference/nt-97/unix-application-portability-windows-nt-alternative-environment-subsystem>.
- [4] Jon Howell, Bryan Parno, and John (JD) Douceur. “How to Run POSIX Apps in a Minimal Picoprocess”. In: *Proceedings of the USENIX Annual Technical Conference*. USENIX, 2013. URL: <https://www.microsoft.com/en-us/research/publication/how-to-run-posix-apps-in-a-minimal-picoprocess-2/>.
- [5] N. Stephen and F. Reynolds. “Linux on the OSF Mach 3 microkernel 1 François Barbou des Places”. In: 1996.
- [6] Johannes Helander. *Unix under Mach: The Lites Server*. 1994.
- [7] M. Bushnell. “Towards a New Strategy of OS Design”. In: (May 2021).
- [8] Hermann Härtig et al. “The Performance of μ -Kernel-Based Systems”. In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP ’97. Saint Malo, France: Association for Computing Machinery, 1997, 66–77. ISBN: 0897919165. DOI: 10.1145/268998.266660. URL: <https://doi.org/10.1145/268998.266660>.
- [9] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

Appendix A

Running Linux app in HelenOS

The source code of the thesis is provided in the attached archive. Its latest version is online at <https://git.kabele.me/vitkabele/helenos>.

First it is necessary to choose the application that will be executed in HelenOS. The current implementation limits the programs to those that can be linked statically. This is not always easy to achieve, even with the traditional `./configure` script. In GCC, for example, the latest version that we managed to statically compile was 4.6, which is really ancient.

For this reason we suggest using the BusyBox¹ which can be easily statically linked even in the last version. Its binary also contains multiple functionalities, including several built-in shells (the shells are not expected to work, see section 3.6).

The application is located in the HelenOS sourcetree at `uspace/app/lilo`. Before compilation please check that all symlinks in the `include/*` are valid on your distribution (tested on Debian 10).

When all symbolic links are fine, please follow the official Building HelenOS guide².

Then boot the produced ISO file, either on a real machine or using QEMU. For QEMU, the `tools/ew.py` script might be used. Please note that some BusyBox subcommands (e.g. `cksum`) expect the vector instructions to work. Proper support is however missing in HelenOS and so the command will fail. In QEMU, this can be avoided by emulating old enough CPU model (tested with `-cpu Westmere`).

Once booted, run the busybox in terminal by typing for example:

```
# lilo busybox cut -d' ' -f 3 /demo.txt
```

¹ <https://www.busybox.net>

² <http://www.helenos.org/wiki/UsersGuide/CompilingFromSource>