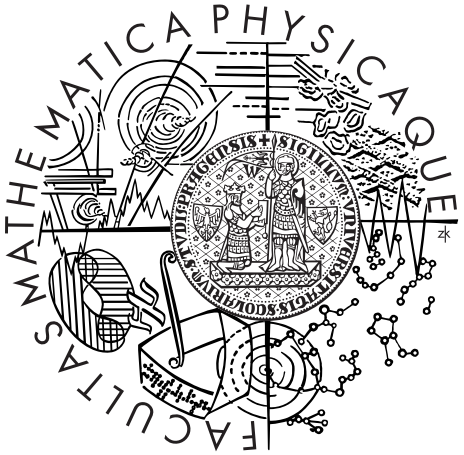
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Jan Mareš

Port of QEMU to HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký

Study programme: Informatics

Specialization: Software Systems

Prague 2015

I would like to thank my supervisor Mgr. Martin Děcký for the time invested in this thesis, to the HelenOS community for their support during the development of the thesis and to Jessica McFadden BA for all the Caro and the proofreading.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

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

In Prague on 27th of July, 2015 Jan Mareš

Název práce: Port QEMU na HelenOS

Autor: Bc. Jan Mareš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký

Abstrakt: QEMU je emulátor procesoru schopný emulovat různé hardwarové platformy jako jsou PC, PowerPC, ARM a SPARC. Úkolem této diplomové práce je portovat QEMU na HelenOS a tím umožnit vývojářům spustit emulaci HelenOS uvnitř HelenOS. Práce obsahuje podrobnou analýzu možných způsobů, jak aplikaci portovat (zahrnující portování knihoven, které jsou ke spuštění QEMU potřebné, nebo jejich součástí) a také analýzu toho, které funkce QEMU (rozumná podmnožina všech funkcí QEMU) jsou potřebné k dosažení cíle a které funkce lze v prototypu vynechat. Hlavním cílem je podporovat emulaci platformy PC (x86 a x86-64). Ačkoliv to není částí implementace prototypu, práce analyzuje možnost použití QEMU jako hypervizoru pro HelenOS.

Klíčová slova: QEMU, HelenOS, emulace, virtualizace, portace software

Title: Port of QEMU to HelenOS

Author: Bc. Jan Mareš

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký

Abstract: QEMU is a machine emulator that is able to emulate environment of various hardware platforms, including PC, PowerPC, ARM and SPARC. The goal of this master thesis is to port QEMU to HelenOS, thus allowing developers run the emulation of HelenOS inside HelenOS. The thesis contains a detailed analysis of the possible porting approaches (including the port of prerequisite libraries or their replacements) and an analysis of the features of QEMU (a reasonable subset of all features of QEMU) that are essential for achieving the goal and features that can be omitted in the prototype implementation. The primary focus of the implementation is to support the PC (x86 and x86-64) guest environment. Although not part of the prototype implementation, the thesis also focuses on analyzing the requirements for running QEMU as a virtualization hypervisor in HelenOS.

Keywords: QEMU, HelenOS, emulation, virtualization, porting software

Contents

Introduction

QEMU is an application that emulates real machines and thus allows a machine code targeted for one hardware platform to be executed and debugged on another platform. This renders QEMU a very useful tool for the development of any operating system. By facilitating QEMU to run on HelenOS, it will give HelenOS (with the already ported GNU C compiler) the ability to create a sufficient environment to develop and test not only itself, but also other operating systems.

QEMU offers virtualization on Linux hosts. This thesis will examine the steps necessary to implement virtualization in HelenOS and to prepare the ground for using QEMU as a hypervisor on HelenOS in the future.

Through the process of porting, HelenOS's support for the POSIX API will be extended and thus the porting of different POSIX compliant applications will be easier in the future. This thesis will also serve as a proof of the maturity and practicality of HelenOS as an operating system.

Goals

The main goal of this thesis is to create a prototype of a port of QEMU to HelenOS, thus allowing:

- emulation of PC platforms (x86 and x86-64) on HelenOS,
- testing of HelenOS inside HelenOS,
- interaction with systems running in QEMU,
- running of other operating systems inside HelenOS,
- extension of HelenOS and its support of the POSIX interface.

The thesis will discuss which features of QEMU are necessary to achieve the goals outlined above and what would be necessary for supporting the rest of the features of QEMU in the future. The prototype is focused on emulating PC platforms on PC platforms (x86 and x86-64).

1. Development context

This chapter describes HelenOS and the parts of it that are affected by this thesis. It also describes QEMU and its architecture.

1.1 HelenOS

HelenOS [1] is an operating system implemented on top of a microkernel called SPARTAN. It began as a project of the Faculty of Mathematics and Physics at Charles University and it has been developed since then partly thanks to the theses completed at the faculty and partly on a private basis by some of the members of the HelenOS community. HelenOS is one of a few microkernel based operating systems supporting a large variety of platforms – e.g. x86, x86-64, IA-64, PowerPC, ARM and MIPS.

1.1.1 Microkernel

The aim of a general microkernel based operating system is to move the responsibility for fulfilling low-level tasks to the user space part of the operating system, thus keeping the kernel as small as possible. This helps to keep the kernel code clean and lowers the probability of introducing bugs into the code. The features supported by the HelenOS kernel can, therefore, be limited mainly to virtual memory management, multitasking and inter process communication (IPC).

1.1.2 IPC and fibrils

One of the functionalities that user space applications need from the kernel is the ability to communicate between each other while preserving the security of the environment (tasks do not share their address spaces). As a means of communication between multiple processes, IPC is used on most of the modern microkernel systems. The kernel of HelenOS offers syscalls specifically designed to implement IPC. The C library (also called standard library) in the user space of HelenOS takes advantage of these syscalls to offer a well designed API to be used to communicate between user space tasks.

To achieve lighter parallelism compared to the kernel threads, and to optimize performance when IPC calls are used, fibrils were introduced in the C library of HelenOS. Every kernel thread executes one fibril. Whereas kernel threads are scheduled by the kernel in a pre-emptive way, fibrils are scheduled cooperatively and the kernel does not participate in their scheduling. Fibrils are switched on

the occasion of: "calling `fibril_yield` function, waiting for an IPC reply that has not yet arrived, requesting an IPC operation that results in blocking the underlying kernel thread, underlying kernel thread being scheduled [2]".

1.1.3 Synchronization primitives

With parallelism there comes a need to have synchronization primitives. HelenOS user space offers different synchronization primitives. For the synchronization of kernel threads HelenOS provides `futex` (fast mutex). Fibrils can be synchronized using `fibril_` synchronization primitives – e.g. mutex, condvar, rwlock. The fibril synchronization primitives use one futex to protect their structures and can therefore be used in an environment where fibrils are scheduled among multiple kernel threads.

1.1.4 Virtual File System

One of the user space services provided in HelenOS is the Virtual File System (VFS) service. The VFS serves as a proxy between the functions manipulating file descriptors in the C library and specific file system drivers. HelenOS supports numerous file systems – e.g. fat, ext, udf, tmpfs and locfs. Not every file system on HelenOS is used to store actual data. An example of such a file system is locfs which is designed to create something similar to a UNIX pipe between applications and services in HelenOS. Writing to (or reading from) `stdout` (or `stdin` respectively) when the console is used for a standard input/output (I/O) of an application actually results in writing to (or reading from) a file node dedicated to the console service in the locfs file system. More details about HelenOS's VFS service can be found in [3].

1.1.5 Coastline

The official repository of the HelenOS source code [4] contains the microkernel and essential user space applications and libraries. Prior to this thesis, there had already been successful attempts to port applications from platforms compliant with the POSIX specification [5] to HelenOS. These applications are not part of HelenOS, yet it is desirable to store the procedures of their cross compilation and to create a friendly POSIX compliant environment where new applications can be ported. This task is the responsibility of HelenOS Coastline.

HelenOS Coastline consists of packages, each of which contains a `HARBOUR` script that is used for the compilation, packaging and installation of the application to HelenOS. The packages can depend on one another and their sources can be downloaded from the Internet. The compilation, packaging and installation process is controlled by the `hsct.sh` script that executes functions in the

HARBOUR scripts of the manipulated packages. Each HARBOUR script can take advantage of the shell interpreter and standard utilities such as patch. The whole idea is inspired by the makepkg utility from the ArchLinux distribution. At the end of the implementation of this thesis every application and library which has been ported will have its own package in the HelenOS Coastline.

1.1.6 POSIX support

To create a POSIX environment for the ported applications the libposix library was created. This library holds functionalities typical for POSIX which were not desirable to be part of the C library. Some of the functions in the C library of HelenOS collide with functions from the POSIX specification – they have the same names, but different API and sometimes their functionality differs as well.

To be able to use these functions in the POSIX library, all the conflicting functions in the POSIX library are prefixed with text posix_. After linking of the POSIX library three libraries are created – libposix, libposixaslibc and libc4posix. The libraries libposixaslibc and libc4posix are used to link applications in Coastline and their symbols are redefined in such a way that the posix_ prefix is removed from the symbols in the libposix and a prefix __helenos_libc is added in front of the symbols in the C library. In this way, it is ensured that applications in Coastline use the POSIX version of the C library, but the HelenOS C library can be independent of the POSIX specification.

The proof of independence of the C library on the POSIX layer can, for example, be observed on the error codes. The C library in HelenOS makes use of signed returned value types. Usually, only positive values are returned when a function finishes gracefully, that is why negative values are used for the error codes. Error codes used by the POSIX specification are assigned positive values and the functions, therefore, use multiple ways of indicating an error, depending on their arguments and their standard return value. This is also one of the obstacles that needs to be resolved when porting POSIX compliant applications to HelenOS.

1.1.7 Plain C Unit Testing framework (PCUT)

The PCUT framework is a unit testing framework created by Vojtech Horky to facilitate the testing of applications written in the C language. It can be found at [6]. This framework is used in HelenOS to test applications and libraries in user space. One executable file is created for each library or application, this executable file contains multiple test suites, each test suite contains multiple tests. When the executable file is run in HelenOS, all the suites with all their

tests are executed. The number of failed tests for each suite is reported. Timeout can be set for each test to discover deadlocks or major performance issues in the code. It is desirable that this thesis will make use of this framework to test the newly implemented functionalities.

1.2 QEMU

As described in the Introduction chapter, QEMU is an application that gives users the ability to create an emulated or virtualized machine. Dynamic translation is used to achieve a good performance of QEMU in the emulation mode. There is no official documentation of the source code or overall architecture of QEMU. Therefore, a description of the architecture has to be based on exploring the code and reading the notes of the developers of QEMU, which usually refer to the older versions of QEMU.

QEMU offers additional features to help with the debugging of the guest code, one of which is the QEMU monitor. The QEMU monitor is a communication port, usually in the form of a window with a prompt, giving users the ability to retrieve information about the emulated machine – e.g. retrieve the actual values of the CPU registers, or to send special commands manipulating the emulated hardware – e.g. pause or stop the emulation.

Another feature QEMU provides to facilitate the debugging of the guest code is the in-built GNU Project Debugger (GDB) server. This server listens on the specified TCP port for a connection from GDB. GDB can then be used to connect to QEMU and to step the code that is currently being executed in QEMU or to print the values of the variables used in the executed code.

1.2.1 Supported platforms

QEMU can be used on two different platforms. With the usage of the MinGW compiler chain QEMU can be compiled and run on Windows. However, its main targets are Linux distributions. This is also demonstrated, for example, by the fact that the virtualization mode is supported only on Linux hosts through the usage of the Kernel-based Virtual Machine (KVM) kernel module. QEMU's main dependency is the GLib library that supports both of the platforms. The functionality provided by this library is used to create a platform independent code in some parts of the source code of QEMU.

1.2.2 Architecture

QEMU needs to be responsive to react to user input, process emulated timers or to respond to commands in the monitor. On the other hand, the execution of a guest code should not be interrupted too often, in order to achieve the highest possible performance. To achieve this, QEMU combines two architectures that are used to implement responsive applications – parallel execution and event loop.

There are further reasons for this approach which are mentioned in [7] – "An event loop cannot take advantage of multiple cores since it only has a single thread of execution. In addition, sometimes it is simpler to write a dedicated thread to offload one specific task rather than integrate it into an event-driven architecture. Nevertheless, the core of QEMU is event-driven and most code executes in that environment."

QEMU's event loop

QEMU's event loop lies in the function `main_loop_wait` and its responsibility is to wait for file descriptors to become writeable or readable and to run event handlers of expired timers and bottom-halves. "Bottom-halves are like timers that expire immediately and are used to avoid reentrancy and overflowing the call stack [7]."

It is not desirable to block for a long time in the event handlers. To avoid this, worker threads are used, for example, in the implementation of asynchronous I/O operations. "When core QEMU issues an aio request it is placed on a queue. Worker threads take requests off the queue and execute them outside of core QEMU. They may perform blocking operations since they execute in their own threads and do not block the rest of QEMU [7]."

Executing guest code

QEMU has two mechanisms to execute guest code – Tiny Code Generator (TCG) and KVM. "TCG emulates the guest using dynamic binary translation, also known as Just-in-Time (JIT) compilation. KVM takes advantage of hardware virtualization extensions present in modern Intel and AMD CPUs for safely executing guest code directly on the host CPU [7]."

Both of these modes allow the control flow to jump into the guest code and execute it. In both cases QEMU uses a different thread to execute the guest code than the thread that executes the event loop. It is also interesting to note that when TCG is used to emulate multiple cores, all of these cores are scheduled in one thread. QEMU thus cannot make use of multiple cores on the host platform unless it is running in the KVM mode.

2. Analysis

In order to port QEMU to HelenOS it is necessary to support additional functionalities required by the features of QEMU and the libraries on which it depends. Each feature has its own requirements of the operating system, most of the requirements are POSIX related, but some of them can affect other parts of HelenOS such as the C library, the VFS service or the console service.

The resulting implementation of this thesis should be easy to integrate in the mainline of HelenOS. As HelenOS uses a microkernel, changes to the kernel should be kept to a minimum. The patches for the ported applications should be suitable for merging back to their official repositories. POSIX applications that are ported in the future should make use of the extensions created by this thesis. These requirements when combined imply that the implementation should meet the following criteria:

- extend the POSIX support in HelenOS as much as possible,
- lower the amount of changes in the HelenOS C library to a minimum,
- lower the amount of changes in the HelenOS services to a minimum,
- lower the amount of changes in the HelenOS kernel to a minimum,
- avoid changes in the code being ported.

The same criteria are therefore used to select the best solution from the options proposed in this chapter to implement the requirements imposed by QEMU.

2.1 Discovering the requirements of the features

The source code of a ported application can present three kinds of dependencies. A compile-time dependency creates a restriction on what types, macros and functions must be defined in order to compile the application. A run-time dependency is a dependency of the ported application on the ability to access a certain function or variable. A dependency on functionality is set out by the application, when it depends on a certain functionality provided by a set of functions and variables.

Let us assume that there is a universal tool that decides whether a function gets executed when sending a specific input to an application, without making any assumptions about the source code of the application. Without loss of generality it can be assumed that every application must call a function `exit` to end its execution. This tool could be used to find out whether the `exit` would be called

when the specific input is provided to the application, thus determining whether the application would halt or not and thus solving the halting problem. This contradicts the supposition that the halting problem cannot be solved.

This means that the only way to find out whether QEMU will require a function that is referred to from its source code, when using a specific set of features, is to execute the application and wait until the function is actually called or to do a comprehensive analysis that would rule out the calling of the function given the specific set of inputs. As a result of this, it is hard to estimate the time needed to port a limited subset of features of an application or library.

To identify different kinds of dependencies, appropriate mock-ups can be used. A compile-time mock-up is a definition of a type, macro, variable or function without its implementation. The task is simply to pass the compilation process. A run-time mock-up is an implementation without any functionality – e.g. empty functions, variables that are never initialized etc. A functionality mock-up skips the parts of the functionality that the ported application does not need – e.g. parameters of a function that are not expected to be used.

The important parts of creating these mock-ups are effectiveness and visibility. The distribution of the mock-ups is going to change rapidly, so it must be easy to place, move or remove them. To be able to remove the mock-ups their usage must be visible, especially if they are used somewhere where a mock-up will not be enough to support the feature. This is achieved using error codes and logging in the mock-ups of functions.

2.2 Core features

This chapter describes those features necessary for running QEMU and allowing emulation of basic hardware, and outlines how HelenOS is going to be extended in order to support them.

2.2.1 Configuration dependencies of QEMU

In order to get past the configuration process of a ported application its dependencies must be satisfied first. This is applied recursively for every library the ported application depends on. Using this method it is possible to create a dependency tree of the ported application. Figure 2.1 shows a dependency tree of QEMU. The dashed line is used to mark optional dependencies mentioned in this thesis. The grey background is used to mark dependencies that can be part of other packages.

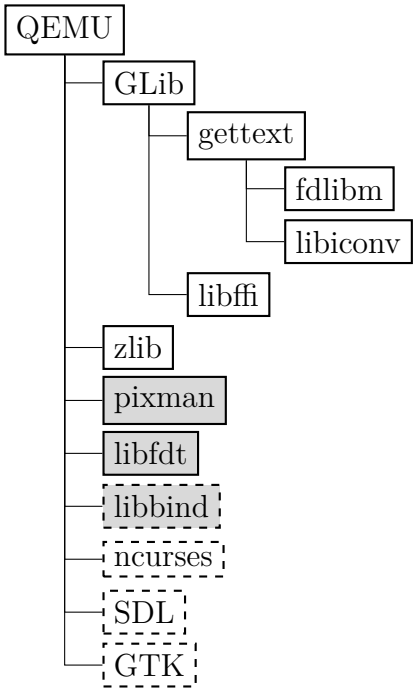


Figure 2.1: Dependencies of QEMU

Some of the libraries required by QEMU were already ported in HelenOS Coastline (see section 1.1.5) at the beginning of this project. The libraries that will have to be ported to pass the configuration process of QEMU are GLib, gettext, libiconv, libffi and libbind.

The source code of QEMU makes use of a functionality provided by the pixman and libfdt libraries. These libraries are not listed as official dependencies because the configuration script is able to compile them with the source code of QEMU. As there were problems with cross-compilation of the pixman library when compiled as a part of QEMU's source code, a separate package was created in Coastline and here cross-compilation was successful. The libfdt library does not offer a configuration script, and therefore cross-compilation in a separate package would be unnecessarily difficult. However, its compilation as a part of QEMU was straightforward.

The pixman library "is a low-level software library for pixel manipulation, providing features such as image compositing and trapezoid rasterization. Important users of pixman are the cairo graphics library and the X server [8]". The libfdt library is designed to manipulate binary files produced by the Device Tree Compiler (dtc) toolchain [9]. It is the author's opinion that libfdt has a very specific usage compared to the pixman library, which is another reason why pixman has its package in Coastline and libfdt does not.

Libbind is not officially requested by GLib, but serves as a provider of functionality to resolve DNS names through the usage of POSIX sockets. The POSIX networking is not supported in the prototype of QEMU proposed in this thesis, so this library is actually a run-time mock-up of functions needed by GLib. This

helped to keep the amount of run-time mock-ups connected with POSIX support for networking in HelenOS as small as possible. See section 2.4.1 for further information about QEMU and networking.

The libraries ncurses, SDL and GTK are optional requirements of QEMU which support features such as the VGA text buffer output and full VGA output. These features and their dependencies are discussed later in the text.

2.2.2 Threads and synchronization primitives

With usage of run-time and functionality mock-ups it was discovered that QEMU depends on threading support obtained from GLib. Different threads are used in QEMU to run the event loop and emulation of the CPU. GLib provides threading support built on top of WINAPI calls – e.g. `CreateThread`, `CreateMutex` etc. or on top of POSIX threads also known as pthread. GLib library is written in a way that does not support mixing WINAPI with POSIX functions. This, along with the fact that HelenOS does not support WINAPI, are the reasons why the implementation of pthread was selected as the right solution for providing threading support for QEMU.

The pthread library defines an interface to work with threads, synchronization primitives and thread specific values. The operations with threads include exit, join, detach and support for return value. The synchronization primitives defined by pthread include mutex, condition variable, sempahore and rwlock. The API connected with thread specific values is designed to allow creating, setting and retrieving values that are stored under a certain key, specific for the currently executed thread.

As mentioned in section 1.1.2, HelenOS provides two different ways of achieving parallel execution in user space. The first is through the usage of kernel threads and syscalls behind the API offered in the C library. The second option is to use fibrils implemented independently on top of the kernel threads in the C library, to provide light-weight threading support for user space applications. The main difference between these two approaches is that the latter one does not support pre-emptivness; instead fibrils are switched in cases of blocking, when another fibril dies or upon calling `fibril_yield`.

Both fibrils and threads in the C library lack the support of operations like detach and join. Despite the fact that fibrils support the concept of return value, the inability to wait for the fibril's execution to finish renders this feature useless. Fibrils, on the other hand, provide a variety of synchronization primitives such as condition variable, mutex and rwlock. In addition, most of the user space code in HelenOS (including all PCUT tests) uses fibrils, and the usage of fibrils and threads together is very limited (see section 2.2.4). These were the two main reasons why fibrils were selected as the foundation of the phtread implementation. Furthermore, this solution does not involve any changes in the kernel of HelenOS.

Thread internal structure

Each pthread will be provided with an internal structure to store its return value when it has exited. For simplicity, an instance of this structure with the underlying fibril functionality will hereinafter be referred to as a pthread thread. This structure must be stored in a space private to each thread.

HelenOS gives the user space applications an option to store values to the thread local storage. A statically allocated variable can be marked thread specific using the `__thread` modifier in front of its type. This functionality is available for threads and fibrils. The starting mechanism of fibrils in the C library then takes care of allocating the storage for each fibril and the values are initialized using the values in designated sections of the ELF file prepared during the compilation of the application. Variables in the thread local storage are accessible only from the current thread, but functions such as `pthread_join` need to access the internal structure from a different thread and, therefore, the internal structure cannot be stored in the thread local storage.

The internal structure of phtread threads could be stored in the `pthread_t` type that is used to refer to a pthread thread in the POSIX API. However, this is not a desirable solution because variables of this type are allowed to be statically allocated and the size of the internal structure would have to be deducible for applications using pthread functionality. This would expose the implementation details and possibly create problems whereby the internal structure would need to be changed in the future development of the POSIX layer in HelenOS. Using a C pointer to the structure would allow the internal structure to be hidden, but it would cause problems with reclaiming the memory addressed by the pointer.

The chosen solution is based on creating a global hash-map containing the internal structure for each thread created by `pthread_create`, where each structure would be retrievable using its ID. This ID will be used as a value for variables with the `pthread_t` type.

Initializing the synchronization primitives

One of the pthread functionalities used by GLib and QEMU is static initializers of synchronization primitives. Pthread specification [5] describes static initializers of some of the synchronization primitives. Fibril synchronization primitives are also provided with static initializers, but there is a slight difference between these two kinds of static initializers. The difference is shown in figure 2.2.

```
pthread_mutex_t mutex_p = PHTREAD_MUTEX_INITIALIZER;
fibril_mutex_t mutex_f = FIBRIL_MUTEX_INITIALIZER(mutex_f);
```

Figure 2.2: Illustration of the difference between fibril and pthread static initializers.

The fibril initializer needs the name of the variable being initialized as its parameter. Every synchronization primitive in fibril has a list of fibrils waiting for the primitive to be released. The name of the variable is used to initialize the head of this list, so its `prev` and `next` pointers point to the head itself, thus forming an empty list. Without the name of the variable, the initializing macro is not able to get the addresses of the parts of the structure being initialized. Let us refer to the static initialization, where the name of the variable is passed as an argument of the initialization macro, as initialization with context.

In order to be able to initialize fibril synchronization primitives without context, the list structure would have to be changed in a way that it could be initialized with plain constants, without the need to compute the addresses of its parts. A generic double-linked list implementation from the C library is used to implement the waiter list in synchronization primitives. The same structure is used all over HelenOS's user space. Changing it in such a way that it would support context free initialization would impose new checks in the structure to detect whether or not the right pointers had been prepared in the head element. This would endanger the performance of the present applications using the list structure. Implementation of a separate list to be used only for fibril synchronization would increase redundancy of the source code.

It is possible to initialize a pthread synchronization primitive which is wrapping a fibril synchronization primitive statically without context, and letting the fibril primitive be initialized later using a call to appropriate initialization function. The way to do this is based on keeping a flag inside the pthread structure of the primitive, marking whether or not the primitive has been initialized. Whenever a function manipulating such a primitive is called, the initialization flag is checked. If the primitive has not yet been initialized, its initialization function will be called. By using this method, the check that would be used to look for initialization of the waiters list is moved from the C library to the POSIX layer library.

As the functions manipulating the primitive are most likely to be called from different threads, the process of checking the initialization flag and the initializing itself is a critical section. The first idea was to protect the section using the compare and swap (CAS) function to set an atomic variable denoting whether the section had been entered or not. That caused a race condition when the atomic variable was set, but the initialization process was not yet finished, as shown in figure 2.3. A condition variable could be used to wait for the initialization process. Mutex offers the same functionality as both the CAS function and condition variable, and for this reason one mutex per each kind of synchronization primitive was used.

```
if(CAS(&initialized,0,1)){
        initialize();
}
/* Here we need to ensure that the
 * function initialize has finished. */
```

Figure 2.3: Demonstration of the problem with the initialization check

The critical section is only among the threads trying to manipulate the same primitive. If the mutex was stored in the structure of the pthread primitive, it would need to be initialized statically, but there is no such mutex. If there was such a mutex, it would not be necessary to do this kind of check in the first place. For this reason, one mutex must be used to protect the critical sections of all the primitives amongst all the threads. This reduces the amount of parallelism that may be achieved. On the other hand, the critical section consists only of operations to compare and set values of the fibril synchronization primitives, and as such the probability of being blocked in this section is quite low. This solution was chosen in the end because it provided a way to implement the functionality whilst avoiding major changes in the C library of HelenOS.

Recursive mutex

The API and the functionality offered by fibril synchronization primitives are very close to the API and the functionality requested by the specification of pthread. The functionality that was requested by QEMU and is not supported by fibril synchronization primitives is a recursive mutex. A recursive mutex is a mutex that can be locked more times by the thread that has acquired it. The POSIX specification [5] also states that a recursive mutex must be unlocked as many times as it was locked before another thread can acquire the mutex.

Implementation of `fibril_mutex` does not provide a recursive mode. One possible solution is to extend the implementation of `fibril_mutex` with a recursive mode. However, the problem with this solution is that it involves changes in a very exposed part of the C library of HelenOS.

Another solution would be to develop a recursive mutex separately in the POSIX layer library of HelenOS. In order to be able to work with the list of fibrils ready for execution as the other fibril synchronization primitives do, this list would have to be accessible from outside the C library. That would make the POSIX layer library and the C library in HelenOS very tightly coupled, thus decreasing the quality of the design of HelenOS user space. This solution would also create significant code redundancy in the functions locking and unlocking the mutex. Due to these drawbacks, the solution whereby `fibril_mutex` would be extended to support a recursive mode was selected.

Thread specific values

Another feature of pthread that GLib, and by extension QEMU, depend on is called thread specific values. As mentioned previously, this feature offers an application the ability to store different values for different threads identified by the same key. The application will first create a key that identifies what a thread specific value will represent, for example this could be a name of the thread. Then, each thread will be able to store and retrieve its specific value associated with this key. When a thread exits, its specific values are destroyed. A user of the API can also specify a destructor for each key. This destructor will be called before the storage of the values of the exiting thread are destroyed, with the value as its argument. Figure 2.4 shows an example of the usage of thread specific values to create independent IDs for threads entering the `thread_entry` function.

```
static pthread_key_t id_key;
static atomic_t id_counter;

static uint32_t get_my_thread_id(){
        return (uint32_t)pthread_get_specific(id_key);
}

static void *thread_entry(void *parm)
{
        phtread_set_specific(id_key,(void*)atomic_inc(&id_counter));
        ...
        uint32_t my_id = get_my_thread_id();
}

void main(int args, char** argv){
        pthread_key_create(&id_key, NULL);
        ...
}
```

Figure 2.4: Example of the usage of thread specific values

To support this feature, the POSIX layer of HelenOS will need to be aware of the created keys and of the mapping between the keys, threads and the values set. A global hash-map of the keys and one hash-map per thread to store the key-value mapping will be sufficient to provide this functionality.

The second hash-map will need to be stored in a place that is private to each thread. The thread local storage presented in section 2.2.2 appears to be a good candidate. The pointer to the hash-map can be stored to a variable allocated in the thread local storage. However, pthread threads already have a private storage provided by their internal structure. To loosen the dependency on the C library and to facilitate debugging of pthread functionality the internal structure of pthread was chosen as the place to store the hash-map containing the thread specific values.

2.2.3 The event loop and timing

As well as pthread, QEMU uses another POSIX feature through its calls, and through GLib as well, that is not present in HelenOS. Functions `pipe` and `select` together form a functionality that is easier to describe using the equivalent functions from WINAPI called `CreateEvent` and `WaitForMultipleObjects`.

As the name suggests `CreateEvent` (`pipe` in POSIX) is used to create an event object that can be fired in any thread at any time. Function `WaitForMultipleObjects` (`select`) is used to wait for the event to be fired. The difference between this functionality and the functionality offered by condition variable is that `WaitForMultipleObjects` (`select`) can be used to wait for multiple events and will block until at least one of those events is fired.

This functionality is used in the event loop of QEMU (see section 1.2.2) to wait for events from various emulated hardware. When an event is fired, a callback attached to the event is called to handle it. If there are no pending events, QEMU's event loop is blocked for the time given to the `select` function as timeout. Emulated hardware timers in QEMU have their time set when the next interrupt should be delivered to the emulated CPU. The soonest of these times is used to specify the timeout of `select` function called in the event loop [7]. In this way it is ensured that QEMU does not block the event loop for too long, but that it does block for some time. Further details about the importance of event loop blocking timeout can be found in section 2.2.4.

The ambiguity created by the usage of POSIX functions lies in the fact that the operations to create, fire, reset or wait for an event are done using a file descriptor as an ID of the event. For example, to fire an event in POSIX created by `pipe` or `eventfd`, function `write` is used. The very same function is used to write data in a file. This originates in POSIX describing a UNIX-like operating system. In UNIX, file descriptors are not used only for files, but also to identify other stream-like structures – e.g. network connection, pipe etc. In the opinion of the author of this thesis, and of the HelenOS community, implementing this functionality in HelenOS's VFS service would pollute its API.

Another option would be to implement the WINAPI functions `CreateEvent` and `WaitForMultipleObjects` instead of the POSIX functionality. The biggest problem with this approach is the way in which the code of QEMU and GLib is forked to support WINAPI and POSIX. Especially in the case of GLib, where these two parts could almost qualify as two different libraries with the same API. Many more functions from WINAPI would have to be implemented in HelenOS, and in many cases these would be functionalites that already exist in POSIX. Also, WINAPI suffers with a similar problem as POSIX, but in this case instead of file descriptors, IDs known as handles are abused to identify structures of different type and usage.

There is a way of restricting the API pollution just to the POSIX layer of

HelenOS. The first step is to distinguish what structure is identified by the obtained file descriptor. This could be done by special marks in the top left bits of the file descriptor. This should not create any additional constraints on the real file descriptors, as applications already have a constraint on how many real file descriptors they can use. This limit is set in macro `MAX_OPEN_FILES` in VFS and it was 128 at the time of implementing this thesis. Please note that a similar trick would have to be used when implementing WINAPI functions to distinguish different kinds of handles.

With the ability to distinguish different kinds of file descriptors the different versions of `write` can be called to serve the request appropriately to the kind of file descriptor received. The same approach can be used to implement other POSIX functions like `read` or `close`. As this is the most straightforward solution that does not pollute the API of non-POSIX parts of HelenOS, it is the solution that was chosen to implement this functionality.

2.2.4 Concurrency level and yield

QEMU uses an endless loop in the thread responsible for the emulation of CPU. If the last block of instructions is not waiting for an I/O event, this loop does not block at any synchronization primitive. As fibrils do not offer a pre-emptive way of switching the currently executed fibril it can occur that code executed in the emulated machine gets stuck. One of the ways this can happen is that the executed kernel is actively waiting for some memory to be changed due to an interrupt coming from a timer. As explained in section 2.2.3 expired timers are handled in QEMU's event loop. This event loop is usually blocked for a specified timeout. Due to the fact that the emulated code is technically not waiting for an I/O event, and because fibrils do not offer pre-emtpivness, the event loop will be blocked forever and the emulated machine will be stuck in an endless loop.

When discussing with the HelenOS community the limitations of usage of threads and fibrils together, it was discovered that there could be an option to use kernel threads as execution containers for fibrils. This would still mean that pthread is implemented using fibrils, but during the process of initialization of a pthread thread, or on demand, there would be a possibility to run more kernel threads and these threads could participate in the execution of the created fibrils. This would create a kind of thread pool, where fibrils become tasks distributed among the threads in the thread pool, and where tasks can enforce a change in the list of tasks currently being executed by blocking at synchronization primitives. The level of concurrency would be derived from the amount of kernel threads executed. The true concurrency would be achieved once every fibril had its own kernel thread.

Two threads are sufficient to run QEMU. As mentioned at the start of this section, QEMU uses a busy loop that can stop blocking and yet the code executed in the busy loop expects that the memory of the emulated machine will

be changed. Other threads started by QEMU are blocked when they are waiting for an external event. Therefore, one thread is needed for the fibril executing the busy loop, while another thread will execute all the other fibrils.

The true concurrency can be substituted by manually switching the currently executed fibril through usage of the function `fibril_yield` in the busy loop. In this way the busy loop will be interrupted and other fibrils will be given a chance to run. It is hard to determine the best place to put a call to the function `fibril_yield`. It is not desirable to interrupt the loop too often when there is nothing else that other fibrils could work on. On the other hand, it is also not desirable to place a call to the function `fibril_yield` too low in the callstack of the busy loop, because other loops inside the busy loop can, under specific circumstances, also become busy.

Both of these approaches have been tried, but both have failed. There were problems with both of the functionalities needed. The fibril version of yield operation did not properly wake up those fibrils waiting for timeout imposed by the timed version of the API of synchronization primitives, primarily those waiting for a condition variable. This problem was discovered and fixed as part of this thesis. It was attempted to resolve the problems with using threads as execution containers with the HelenOS community at the end of the implementation of this thesis. As a result, two bugs in the HelenOS kernel and a race condition in function `fibril_switch` were reproduced and then fixed later by the HelenOS community. The first bug in the kernel had been reported two years before this thesis was finished and is described here [11].

The second bug was discovered during the implementation of this thesis, when a thread crashed whilst another thread was waiting for a futex held by the crashed thread. The application `taskdump` was called to printout the stack trace and other debug information of the crashed task using the `udebug` module in the kernel of HelenOS. However, the thread waiting for the futex was not stoppable and therefore the `taskdump` and the crashed task ended up in a deadlock. An attempt to kill the crashed task caused a kernel panic, as the `udebug` structures of the blocked thread were found in an unexpected state. This bug was resolved by marking the thread waiting for a futex as stoppable in the `udebug` module.

The race condition discovered in the function `fibril_switch` was caused by unprotected critical sections when shutting down a fibril and not holding the `async_futex`. This proves that the implementation of fibrils could not have been used effectively within an environment running more kernel threads, prior to the implementation of this thesis. This race condition was fixed by protecting the identified critical sections. However, there remains a race condition in either the POSIX layer of HelenOS or in the implementation of fibrils. For the sake of the prototype implemented in this thesis, the solution with the usage of fixed `fibril_yield` seems to be sufficient.

2.2.5 Modules

Another feature used directly by QEMU but missing in HelenOS is not a POSIX feature, but rather a feature of the GNU C compiler (known as GCC) and the C library of the target system. QEMU uses a constructor attribute to mark functions that are in charge of the initialization of different kinds of modules of QEMU. This attribute specifies that the functions marked with it are called in the process of executing an application, just before calling its main function. Thanks to this, the code of QEMU's module does not need to be referenced in the rest of the application and thus it does not have to be linked with it. If the module is linked with the application, the initialization function is called before the main function and the module can, for example, register its parts in the lists of devices QEMU offers to emulate [10]. That is why this missing functionality manifested in the fact that QEMU did not offer any hardware to emulate.

The GNU C compiler stores the pointers to the functions marked by the constructor attribute (or destructor attribute) to a special section of an ELF file produced in the compilation process. This section is called `.init_array` (or `.fini_array` respectively). Please note that C constructors do not receive any parameters and do not have any return values, thus their execution is different to the execution of the main function. Also, an endless loop created in a constructor function (or destructor function respectively) can cause the application to get stuck before calling the main function (or after returning from the main function respectively). Calling `exit` in a constructor function can result in the main function never being executed. That is why it is not desirable to use this functionality outside the POSIX layer of HelenOS. To implement this functionality, the linker script and the C library in the HelenOS user space would have to be changed.

A way of avoiding changes in the linker script used to compile applications into the user space of HelenOS would be to change the source code of QEMU in such a way that it would not rely on the constructor attribute. These changes would prevent the possibility of linking QEMU's modules optionally. The reintegration of the altered code of QEMU back to the official repository of QEMU would be almost impossible. For this reason, the first solution outlined was selected as the one to be implemented. Another benefit of implementing this feature in the C library is not connected with this thesis, and it is that support for C++ in HelenOS will be extended as a by-product. The function pointers in `.init_array` (or `.fini_array` respectively) are used in C++ to call a constructor (or destructor respectively) corresponding to the class of a statically allocated instance.

2.3 Key features

This chapter describes the features of QEMU that are not necessary to run QEMU, but are necessary to meet the goals of the thesis, which means allowing for

the testing of an operating system on HelenOS.

2.3.1 Input and output

QEMU offers minimalistic input and output connected to the emulated machine through an emulated serial port. This feature is enabled by the option `-nographic` and uses `stdout` to print characters written by the emulated machine to the serial port and `stdin` to wait for input that would be passed to the machine through the serial port. To wait for the input to be accessible, QEMU uses `select` in a similar manner as when it waits for the events on pipes described in section 2.2.3. The file descriptor of `stdin` is mixed up with file descriptors of pipes in the event loop.

As mentioned in section 2.2.3 the VFS service in HelenOS does not offer `select` functionality and the same section also describes why it is not desirable to implement this functionality in the VFS service. To allow input to be processed by QEMU it is necessary for the source code of QEMU to be able to wait for events on the terminal along with the events connected with the emulated hardware. GLib solves this problem on Windows, where waiting for events on files is also not supported, by creating a special thread that is blocked in the read operation; when the read operation gets unblocked the data read is stored in a special buffer belonging to this thread and the read event is fired. Thus, `select` will unblock marking `stdin` in a way that shows an event is pending and the application can read from it without blocking.

Similar arguments as in section 2.2.3 were used to select this as a solution instead of implementing support for `select` in HelenOS's VFS service. `stdin` in HelenOS uses locfs – a special file system that supports communication with HelenOS's services. Locfs does not support parallel reads and writes, which means that while a task is reading from a service it cannot write to the same service. In this solution, there is going to be a thread that is blocked in the read operation of the console service most of the time. Any attempt to write to the console service will block until a key is pressed. This, however, is not a desirable functionality. There also exists another way to access the console service directly without using locfs as a proxy. The function `console_get_event` was used and the events were translated to characters stored by the polling thread.

2.3.2 Block devices

In order to be able to debug HelenOS in QEMU, its image has to be loaded in QEMU's emulated CD-ROM drive. To support emulation of block devices QEMU uses functions `pwrite` and `pread` in addition to the functions provided by the VFS service of HelenOS. As POSIX specification states, the difference between these functions and standard `write` and `read` is that `pwrite` and `pread` use

a position in the file specified by the caller and do not alter the position used and updated by `write` and `read` functions [5].

There are two ways to implement these functions. The first is to implement `pwrite` and `pread` with the usage of `lseek`. `lseek` would first retrieve the position associated with the file descriptor, then the new position would be set again by `lseek`. The standard version of `write` or `read` would be used to do the actual writing (or reading) and when they return, the previous position would be restored using `lseek` once again. Expensive syscalls are not used to communicate with the VFS service on HelenOS, but IPC calls can be quite costly as well. There are three IPC calls on top of a call to `write` or `read` to simulate the functionality of `pwrite` and `pread` using `lseek`.

Another solution would be to implement `pwrite` and `pread` directly in the VFS service of HelenOS. Although this solution will affect the source code of HelenOS's service and the C library, it was the solution that was implemented, due to the increased performance of QEMU and the fact that the changes will be relatively small and non-invasive.

2.3.3 VGA text buffer

When testing Linux kernels in QEMU, a serial port was sufficient to support output (or input) from (or to) the user space. Unlike Linux, HelenOS does not provide a user space console using a serial port to print the output and retrieve the input from the user. To allow output and input from and to the HelenOS console, the VGA text buffer will have to be supported at least. To configure HelenOS to use this buffer for I/O of the user space applications, option `CONFIG_EGA` must be set to 'y' and option `CONFIG_FB` must be set to 'n'.

QEMU allows the VGA text buffer to be displayed using the curses library when run with `-curses` option. The curses library then also captures the input using `select` function in a similar way that QEMU does when the serial port is used. Running QEMU with the `-curses` option in comparison with the `-nographic` option also gives the ability to switch between three QEMU windows. The first displays the VGA text buffer, the second allows the use of the QEMU monitor that can manipulate the emulated machine and display various information about it. The third window displays the output from the serial port.

To make this feature a part of the prototype presented by this thesis, library *ncurses* will need to be ported to HelenOS. The curses library uses ANSI terminal commands – e.g. move cursor to position, clear screen, scroll screen etc. These commands need to manipulate an instance of the `chargrid_t` structure corresponding to the service providing the output for the task currently being run.

In HelenOS two services are used to provide console output, depending on

what graphical mode HelenOS is running in. Service `vterm` is created every time an instance of the application `vterm` is started in the GUI mode of HelenOS (HelenOS is configured to use framebuffer). When HelenOS is configured to avoid using framebuffer, it uses VGA text buffer instead. In this case, one instance of the `console` service is started, delegating the output to multiple instances of the `chargrid_t` structure, based on which console is actually selected.

There is already quite a high code redundancy between these two services. This is why the HelenOS community is planning to re-factor the console subsystem of HelenOS, and that means that any solution proposed by this thesis will have to be re-implemented in the near future. Therefore, the source code of these services will be altered in a non-invasive way.

2.4 Possible extensions of the feature set

Some of the features offered by QEMU were not selected for the prototype, as their implementation would have been difficult and the goals of this thesis were achievable without them. This chapter describes the most interesting of these features and how they could be implemented in the future.

2.4.1 Network

As mentioned in section 2.2.1 QEMU's emulation of network devices is not supported in the prototype presented by this thesis. At the start of the implementation of this thesis, networking was implemented in the C library of HelenOS in a similar fashion to POSIX networking. That version of networking was considered as insufficient by the HelenOS community and was re-factored at the time that this thesis was almost completed. The API of the HelenOS networking changed completely, therefore any work that would be done to support networking in the POSIX layer of HelenOS would be proved redundant.

To be able to support the emulation of network devices, support for the POSIX networking API would need to be brought back to life, this time solely in the POSIX layer of HelenOS with the usage of the new networking API offered by the C library. The *libbind* would have to be tested, and if the *libbind* would not be portable, then the API it offers would also have to be implemented in the POSIX layer.

Once HelenOS supports the networking API and functionality required by QEMU, it will allow the emulation of network devices and the usage of the GDB server that QEMU provides. In this way, the code running in QEMU could be debugged on HelenOS once the GDB was ported to HelenOS, or in the host of HelenOS if HelenOS was being run in a virtualized or emulated machine, using TCP port redirection.

2.4.2 Full VGA support

To support full graphical output, QEMU offers integration with the libraries GTK and SDL. HelenOS Coastline is not prepared to offer the Graphical User Input (GUI) to the ported applications. To give the ported applications the ability to access the compositor subsystem that is used in HelenOS to create GUI, the `gui` and `draw` libraries would need to be accessible in HelenOS Coastline. As Coastline uses the POSIX version of the C library, includes of these libraries will have to be fixed in a similar way to the includes of the original C library, in order to support their usage in Coastline.

The GTK library depends on the functionality offered by the X server. The SDL library offers the creation of separate rendering drivers to support rendering on different platforms. This is the reason why the SDL library would be more suitable to port in order to support the full VGA output of the machine emulated by QEMU. Although the full VGA support is not implemented in the prototype created by this thesis, it will most probably take place after the completion of this thesis as it is a feature requested by the HelenOS community.

2.5 QEMU as a hypervisor for HelenOS

One of the goals of this thesis is to inspect the possibility of using QEMU as a hypervisor for HelenOS. QEMU only offers a virtualization mode on the Linux hosts. The reason for this is the tightly coupled source code of QEMU to the functionality offered by Linux KVM. That is also why virtualization in QEMU is called KVM mode. KVM in Linux is itself a hypervisor, which allows the creation of new virtual machines and the emulation of the guest code in an virtualized environment.

As mentioned in the documentation of the KVM API [12], the API is centralized around file descriptors. The communication between a user space program and the KVM module on Linux is built on sending commands to the KVM module using the `ioctl` function on file `/dev/kvm` that is present on Linux when the KVM module is loaded properly. A `KVM_CREATE_VM` `ioctl` command on this handle will create a virtual machine file descriptor which can be used to issue `ioctl` commands to the virtual machine. A `KVM_CREATE_VCPU` `ioctl` command on a virtual machine file descriptor will create a virtual CPU and return a file descriptor pointing to it. Finally, `ioctl` commands on a virtual CPU file descriptor can be used to control the virtual CPU, including the important task of actually running the guest code.

```
KVM_CREATE_VCPU,
KVM_RUN,
KVM_CREATE_DEVICE,
KVM_INTERRUPT,
KVM_GET_DIRTY_LOG,
KVM_REGISTER_COALESCED_MMIO,
KVM_SET_USER_MEMORY_REGION,
KVM_UNREGISTER_COALESCED_MMIO,
KVM_SET_REGS,
KVM_GET_REGS,
KVM_SET_ONE_REG,
KVM_GET_ONE_REG,
KVM_GET_FPU,
```

Figure 2.5: Some of the KVM `ioctl` commands used by QEMU

To be able to port QEMU's KVM mode to HelenOS, a special hypervisor service would have to be created in HelenOS. This service would be able to create virtual machines in a similar way that KVM does on Linux. Calls to `ioctl` in the source code of QEMU would have to be substituted with IPC calls directed to this service. File descriptors would have to be substituted with other means of identification. Some of the `ioctl` commands used by QEMU are presented in figure 2.5. Their semantics are explained in [12].

One of the ways to substitute file descriptors would be to start a different service wherever a file descriptor is returned when using KVM. When an equivalent of a `KVM_CREATE_VM` command is issued to the hypervisor service a virtual machine service would be started and `service_id` would be returned to allow the client to issue commands to this service. A similar approach would be used to run a virtual CPU.

If this hypervisor service is implemented in HelenOS, QEMU could make use of its API to start and create virtual machines. QEMU could also be used to emulate hardware and process input and output of the virtual machines. To be able to port QEMU without too much difficulty, quite strong constraints on the API of the proposed HelenOS hypervisor will be imposed by QEMU.

Problems with concurrency may occur again during the eventual porting of QEMU whilst using the proposed hypervisor in HelenOS. QEMU runs code in KVM in a separate thread. This thread is similar to the thread executing emulated code described in section 2.2.4. The difference is that when using the TCG emulator instead of KVM, control flow returns to QEMU's code periodically. This introduces a possibility to insert a call to `fibril_yield`. When KVM is used, the control is returned only when there is an I/O operation pending or when an interrupt was not processed by the guest code. This may be insufficient to switch fibrils and to let expired timeouts be processed.

3. Implementation

This chapter describes the implementation of those solutions proposed in the Analysis chapter as the preferred solutions to achieve the goals of this thesis. This includes changes and extensions of the POSIX layer in HelenOS, the C library in HelenOS, the Virtual File System service and the ported applications and libraries.

3.1 Changes in the ported code

During the porting of QEMU to HelenOS there were changes that needed to be implemented in the source code of QEMU and the libraries on which it depends (see figure 2.1). A Git source code repository (see attachment A) was used to store the history of the changes in the source code of the ported QEMU. As an initial point for the source code of QEMU the branch containing release 2.2.0 was used.

The configuration script of QEMU was extended to support HelenOS as a target system. Thanks to this, the macro `CONFIG_HELENOS` was exposed to the source code of QEMU. This allowed non-invasive changes in the code that should not harm the stability of the changed code when it is compiled on other targets. Therefore, the source code of the ported QEMU should be suitable for merging back to its official repository.

When porting GLib library modules, gmodule and gio were skipped as QEMU does not need their functionality and they also depend on dynamic loading of shared libraries, which is still considered as an experimental feature in HelenOS and only works on x86 platforms. After finding the right options for the configuration, the libraries gettext, libiconv and libffi were compiled successfully. Their code has been patched and run-time mock-ups were used to identify the missing functionalities in the POSIX layer when the specific features of QEMU were tested.

As a part of this thesis, packages `qemu`, `pixman`, `libffi`, `glib`, `gettext`, `bind`, `iconv` and `ncurses` were created in HelenOS Coastline. These packages contain source code patches and options for the configuration script allowing the cross-compilation of the applications or libraries which they represent.

To support the configuration script of QEMU the script `hsct.sh` in HelenOS Coastline was extended to provide the applications being built with a version of `pkg-config` script suitable for cross-compilation. Other commands were introduced in the script such as `clean-tree` and `reinstall-tree` to work with the dependency tree of the package that was already built.

3.2 Implementing pthread in the POSIX layer

The functionality offered by the pthread library is necessary to support threading in GLib and QEMU. This functionality was implemented in the POSIX layer of HelenOS on top of existing fibril functionality from the C library of HelenOS, as proposed in section 2.2.2.

Fibrils in the C library lack a direct equivalent of the functionality specific for functions `pthread_join` and `pthread_detach`. The POSIX specification [5] of these functions suggests that a thread created by `pthread_create` can be found in four different states. A thread that is running and can be joined is *joinable*. A thread that was running when function `pthread_detach` was called on it becomes *detached*. A thread that exited while not being *detached* is *exited*. A thread that exited and its return value has been retrieved using `pthread_join` or discarded using `pthread_detach` is *terminated*. Please note that a thread can be created in the *detached* state using the `PTHREAD_CREATE_DETACHED` attribute or in the *joinable* state using the `PTHREAD_CREATE_JOINABLE` attribute. If the attribute is not specified a thread is created in the *joinable* state.

Taking these states into account, the implementation of the `pthread_join` function can be described in the following way. The function blocks the caller until the thread specified as an argument of the function changes its state from *joinable* to *exited*. When the thread switches to the *exited* state its return value is retrieved and its state is changed to *terminated*. If the thread is not *joinable*, the error code `EINVAL` should be returned as stated by the POSIX specification [5].

Similarly, `pthread_detach` can be implemented through manipulation of the state of the specified thread. When `pthread_detach` is called on a thread that is in the *joinable* state, its state must be changed to *detached*. In the case that the thread is *exited* its state will be changed to *terminated*. If the thread is already *detached* the error code `EINVAL` will be returned as requested by the POSIX specification [5].

Thread internal structure

To implement these operations as described, the state of a thread will need to be stored. An ability to wait for a change in the state will need to be provided. A return value also needs to be stored, in order that it can be retrieved later upon calling `pthread_join`. To achieve this, every fibril created with `pthread_create` was provided with an internal structure called `pthread_internal_t` as discussed in section 2.2.2.

The internal structure must be retrievable by the ID of a pthread thread in order to allow funtions `pthread_detach`, `pthread_join` and `pthread_exit` to access its state. That is why the hash-map `pthread_threads_hash_ta-`

ble was used to hold internal structures using ID as their key. When a thread is created using pthread_create its internal structure is allocated and initialized. The underlying fibril is created using the fibril_create function and the returned ID of the created fibril is used to return the ID of the created pthread thread, and to insert the internal structure to the hash-map.

Starting and termination

In order to confirm with the POSIX API and to be able to store the return value, every fibril started by pthread_create uses pthread_internal_entry as an entry point. This function takes as an argument a structure holding a pointer to the start routine passed to pthread_create as an argument. When the start routine returns, this function also ensures that pthread_exit is called to avoid redundancy of the source code that is responsible for shutting down the thread.

As the POSIX specification states: "The pthread_join or pthread_detach functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed [5]." The reason for this is that if a pthread thread was not joined or detached its return value must be preserved even after the thread has finished its task. This is implemented using a *terminated* state introduced in the aforementioned text. The function pthread_exit blocks the fibril until the state changes to *terminated*, only then it is safe to reclaim the memory taken by the pthread internal structure and the underlying fibril.

Thread specific values

The discussion in section 2.2.2 proposes a way to implement the functionality required by internals of GLib library. This functionality depends on creating a key that is later used to store and retrieve different values for different threads. As proposed in the analysis, two kinds of hash-maps were used to implement this functionality.

The function pthread_key_create takes two arguments, the first is a C pointer used to return a key created when the function finishes, and the second is a C pointer to a destructor function. As per POSIX specification [5], a destructor function is called on every non-null value associated with the created key. The function first allocates the ID for the newly created key, which is done by incrementing an atomic variable last_key. The structure of the key is allocated afterwards to store both the ID and the destructor. This structure is then stored to a statically allocated pkey_hash_table hash-map. The ID of the stored keys is used as a key of this hash-map.

The function pthread_setspecific takes as its arguments the ID of the

key under which the value should be stored, and the value to be stored for the currently executed pthread thread. Due to the fact that the values for one key may not be shared amongst the pthread threads, they have to be stored in a space private to each thread. The aforementioned phtread thread's internal structure fits this purpose. Thus, another hash-map is allocated in the internal structure of each pthread thread. This hash-map is used to store structures carrying the specified key ID, the specified value and a copy of the destructor pointer associated with the key. The function `pthread_getspecific` can later retrieve the value from this hash-map using the ID of the specified key.

The POSIX specification states: "At thread exit, if a key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the value of the key is set to NULL, and then the function pointed to is called with the previously associated value as its sole argument [5]." This is the reason it is necessary to store the pointer to a destructor function within the structure holding the value. The key could be deleted sooner than the thread exits by a call to function `pthread_key_delete` and the information about which function should be called, when the values ought to be reclaimed, would be lost.

The POSIX specification also requires that: "If, after all the destructors have been called for all non-NULL values with associated destructors, there are still some non-NULL values with associated destructors, then the process is repeated. If, after at least `PTHREAD_DESTRUCTOR_ITERATIONS` iterations of destructor calls for outstanding non-NULL values, there are still some non-NULL values with associated destructors, implementations may stop calling destructors, or they may continue calling destructors until no non-NULL values with associated destructors exist, even though this might result in an infinite loop [5]." This requirement was omitted in the implementation of this thesis, although it may be seen as POSIX compliant where `PTHREAD_DESTRUCTOR_ITERATIONS` is set to one.

Combination of pthread threads and plain fibrils

The two kinds of hash-maps used for the thread specific values need to be initialized. The static hash-map `pkey_hash_table` is initialized with a first call to function `pthread_key_create`. The second kind of hash-map needs to be initialized in the process of initialization of each thread. For this reason the initialization is done in function `pthread_create`.

It was discovered that GLib also uses functions `pthread_setspecific` and `pthread_getspecific` in the fibril that enters the main function of the application. This fibril is created by the C library that is not (and should not be) aware of the internal structures created for pthread threads by the POSIX layer in HelenOS. Therefore, the storage for thread specific values is not accessible. In order to overcome this issue, one internal structure that is allocated statically was created. This internal structure is inserted in the `pthread_threads_hash_table`

hash-map when the function `pthread_create` or `pthread_setspecific` is called for a first time.

This imposes a limitation on the usage of pthread threads and plain fibrils together. If thread specific values are used there can be only one plain fibril using functions `pthread_setspecific` and `pthread_getspecific`.

3.2.1 Synchronization primitives

As discussed in section 2.2.2 the implementation of synchronization primitives offered by pthread specification is a straightforward redirection of pthread API to the API of the fibril synchronization primitives, with a few exceptions.

The first exception is a problem with static initializers, as discussed in section 2.2.2. Every synchronization primitive that needs to be initialized statically was provided with a field called `initialized` on top of a fibril structure that it wraps. This field is set to `false` when the pthread synchronization primitive is initialized using a static initializer. A call to every function from the API of these primitives checks whether the primitive has been initialized. If not, its initialization process is started.

Section 2.2.2 describes the setting of `initialized` and initialization itself as a critical section that needs to be protected by one mutex shared by all primitives. To avoid a reduction of parallelism a fast path optimization was introduced. This optimization is based on making `initialized` an atomic variable and testing this variable outside the critical section protected by the mutex. This serves as a hint – if the atomic variable is set, the initialization has finished and locking the mutex can be avoided, otherwise the initialization may or may not have been finished and it is necessary to enter the critical section and test the variable again. This optimization is demonstrated in figure 3.1. The synchronization primitives that are initialized using initialization functions (e.g. `pthread_mutex_init`) have this variable set to `true`. This, along with the fast path optimization, avoids the locking of the mutex in such cases.

```
if(!atomic_get(&st->initialized)){
        lock(&mutex);
        if(!atomic_get(&st->initialized)){
                init(&st);
                atomic_set(&st->initialized,true);
        }
        unlock(&mutex);
}
```

Figure 3.1: The optimization of initialization of synchronization primitives

Another difference between the pthread API specified in POSIX and the API of fibril synchronization primitives is the way in which timeouts are specified. The

API of fibril synchronization primitives uses relative timeouts, the POSIX API on the other hand uses absolute timeouts. The reason for using absolute timeouts is explained in the rationale of the POSIX specification: "First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts [5]." The race condition is demonstrated in the small example shown in figure 3.2.

```
clock_gettime(CLOCK_REALTIME, &now);
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

Figure 3.2: Example of the race condition when using relative timeouts

"If the thread is preempted between the first statement and the last statement, the thread blocks for too long [5]." As the prototype proposed by this thesis does not support pre-emptiveness, this is not a significant issue. In addition, one of the secondary goals presented in the Analysis chapter is to avoid changes in the C library. Nevertheless, this could be one of the changes in the C library of HelenOS to be considered at a future time, when the pre-emptiveness of fibrils is officially supported.

There is another problem potentially arising from the difference in these two APIs. As the structure `timespec` is used to specify the absolute timeout, the relative timeout as a result of the conversion may not fit in the timeout argument of the `fibril_condvar_wait_timeout` function. To avoid this problem `SUSECONDS_T_MAX` is used as a timeout if the value overflows. If this happens and the call timeouts, it is restarted with an updated timeout based on the value of the absolute timeout specified when calling `pthread_cond_timedwait`.

Semaphore

The functionality that fibril synchronization primitives miss completely, and which is requested by QEMU, is the functionality of semaphore. This functionality was implemented using `fibril_mutex` and `fibril_condvar`. The structure for semaphore was created containing value, value protecting mutex and a condition variable signalling to the waiters that the value has lifted from zero.

The function `sem_post` was implemented in such a way that it increments the value of the specified semaphore and wakes up one waiting thread using `fibril_condvar_signal`, provided the value was zero before it was incremented. If the value of the semaphore is greater than zero, function `sem_wait` decrements the value. Otherwise it waits on the semaphore's condition variable until the thread holding the semaphore calls `sem_post`. The function

sem_timedwait uses an absolute timeout, which could cause problems similar to those described in the previous section. This is why a solution similar to the one presented in the previous section was applied to it. The critical sections when the value of the semaphore is tested and then altered are protected by the semaphore's mutex.

As described in the POSIX specification [5], semaphore can be created in a shared mode by calling the function sem_init with the argument pshared set to one. A semaphore in a shared mode can be used to synchronize multiple processes. This feature is not implemented, and when the argument pshared is set to a non-zero value an error code is returned to signal that. This is sufficient for the prototype implemented in this thesis as QEMU does not require this feature.

Recursive mutex

As per the discussion in section 2.2.2 a recursive mutex was implemented as an additional mode of fibril_mutex. The POSIX specification [5] describes the behaviour of the desired recursive mutex in the following way: "If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire [5]."

The actual implementation differs a little from the description, but it offers the same functionality. The reason for this is that the original code of fibril_mutex was meant to be extended in a non-invasive way. Therefore, the fields flags and recursion_counter were added to the structure of fibril_mutex. The field flags is used to set the mode of the mutex.

The behaviour of the fibril_mutex_lock was changed in such a way that if a mutex that is already owned by the currently executed fibril is attempted to be locked, the recursion_counter will be incremented. A similar change was applied to the function fibril_mutex_trylock.

The implementation of the fibril_mutex_unlock was altered in the following way: if the recursion_counter of the specified mutex has a non-zero value, the recursion_counter is decremented. These three changes are only applied to a mutex that has the FIBRIL_MUTEX_RECURSIVE flag set in its flags field.

The function fibril_mutex_init_flags can be used to initialize the fibril_mutex with flags. This function was created to preserve the existing API and it uses the function fibril_mutex_init to initialize the structure of a mutex without initializing the flags field, to limit the code redundancy.

3.2.2 Fix of the `fibril_yield` function

As mentioned in section 2.2.4 the function `fibril_yield` did not work as expected. The problem manifested when the busy loop of the fibril emulating the CPU in QEMU kept going without blocking at any synchronization primitive. To ensure that other fibrils were going to get a chance to execute their code, the function `fibril_yield` was put in the busy loop. This function is supposed to enforce the currently executed fibril to switch to another fibril that is ready to run, assuming that there is such a fibril.

It was discovered that even though there were fibrils ready to run because the timeout they specified when calling the function `fibril_condvar_wait_timeout` had expired, the function `fibril_yield` behaved as though there were no fibrils ready to run. As a result, the event loop of QEMU stopped being scheduled and the machine stopped reacting to events like interrupts coming from timers or user input. After analysing the functionality of `fibril_switch` in the C library of HelenOS it was observed that the expired timeouts were handled in the `async_manager` fibril. The `async_manager` was scheduled only when there were no other fibrils ready to run, which was not the case in this scenario.

To resolve this problem the `fibril_yield` function was modified in such a way that it first schedules the `async_manager` fibril that will deal with the expired timeouts. As a result of this, fibrils with expired timeouts are put in the list of fibrils ready to run. As there are fibrils ready to run, the `async_manager` skips waiting for IPC calls and switches back to the first fibril previously added to the ready list due to expired timeouts.

3.3 Support for GLib event loop

The discussion in section 2.2.3 identifies the POSIX functions `select` and `pipe` as necessary to support the event loop functionality provided by GLib. The function `pipe` creates two file descriptors, the first one is supposed to be written to, the second one is expected to be read from. Put simply, what is written to one end can be read on the other. Despite the fact that the POSIX function `pipe` is designed to pass data between a parent process and a child process on UNIX-like systems, it is used in QEMU to serve as an event object.

QEMU creates a pipe and sets it to `O_NONBLOCK` using the `ioctl` function. If the `select` function is called on such a pipe it blocks until something is written to the pipe. In this usage it does not really matter what is written to the pipe, it is merely the functionality to wait for this event that is used. Implementing this functionality in HelenOS's VFS service was ruled out in section 2.2.3. The solution to use WINAPI functions `CreateEvent` and `WaitForMultipleObjects` was presented because these functions are intended solely to create a functionality that QEMU needs and nothing else. However,

this solution was ruled out for other reasons as specified in section 2.2.3.

As mentioned in section 2.2.3 the difference between this functionality and a simple condition variable is the ability to wait until one event of the many is signalled, as described in the name `WaitForMultipleObjects`. For this reason, the library `libevent` was implemented to offer this functionality. This library serves as a background of the `pipe` and `select` functions, but can be used in the future to implement functions `CreateEvent` and `WaitForMultipleObjects`, if a WINAPI layer is created for HelenOS.

The `libevent` library offers object `event_t` representing an event object similar to objects created by the `CreateEvent` function. A part of the API of this library is shown in figure 3.3.

```
int event_set(event_t event);
int event_reset(event_t event);
size_t event_is_set(event_t event);
int event_wait_first_mark(const event_t *events, size_t event_count,
    fibril_mutex_t *mutex, const struct timeval *timeout, event_t *
    revents);
int event_wait_all(const event_t *events, size_t event_count,
    fibril_mutex_t *mutex, const struct timeval *timeout);
```

Figure 3.3: The API of the `libevent` library

The function `event_set` is used to set (or fire) the event. The function `event_reset` is used to clear the event if it is pending. The function `event_is_set` is used to determine whether the event is pending. The function `event_wait_first_mark` is used to wait until the first event of the specified events is fired. The function `event_wait_all` is used to wait until all the specified events are fired.

The functions `event_wait_first_mark` and `event_wait_all` use optional arguments such as `mutex`, `timeout` and `revents`. The `mutex` argument is used to specify a mutex that will be unlocked when the waiting starts and locked again when the waiting finishes, thus providing a similar functionality as condition variable does. The `timeout` argument is used to specify an absolute timeout for the waiting time. The `revents` argument is used by function `event_wait_first_mark` to mark the events that have been fired during the waiting time.

As a condition variable is the only signalling primitive present in HelenOS it was used as a constituent part of the `libevent` library. The implementation of the `event_wait_first_mark` function was the biggest problem, because HelenOS does not support anything like it. The problem is that the issuing fibril has to be blocked at a single condition variable, yet the setting of any one of the specified events has to result in the signalling of that variable. To resolve this issue the subscription model described in [13] was used.

The idea is that the fibril waiting for an event subscribes itself to the list of the signallers and waits (goes out of the ready list and switches to another fibril). When an event is fired, it traverses its list of signallers and signals all of them, thus waking-up all the fibrils that have been waiting for that event. The function `event_wait_first_mark` is implemented in such a way that it sets the same signaller to multiple events and waits. When any of the events are set, the signaller is set, and the fibril is woken-up. The `libevent` library uses only the structures and the functionality provided by the C library in HelenOS and is therefore suitable for being included in the C library. However, the decision whether or not to include it in the C library was left to the HelenOS community.

As a result of the discussion in section 2.2.3 the `pipe` function was implemented in such a way that it returns file descriptors which can be easily distinguished from file descriptors corresponding to real files. The functions `read` and `write` were forked so that a different behaviour is used for file descriptors created by the `pipe` function. Pipe as an object is represented by a structure containing a memory buffer and three events – a read event, a write event and an exception event. The reason for this is to match the arguments `readfds`, `writefds` and `exceptfds` of the `select` function as described in the POSIX specification [5].

When the `read` function is called on a pipe, part of its buffer is read and its write event is set to signal that there is space to write to in the pipe. When the `write` function is called on a pipe it checks whether there is space in the pipe, and if so the data is written to its buffer and its read event is set to signal that there is data to be read in the pipe. If there is no space in the pipe, it waits on the write event until a fibril reads something from the pipe and thus releases a part of its buffer. If the flag `O_NONBLOCK` is set on a pipe, `read` and `write` do not block but return an error instead. Please note that the `pipe` function does not support sharing the file descriptors among multiple tasks, as this functionality is not requested by QEMU.

3.4 Support for C constructors in HelenOS

Section 2.2.5 describes the dependency of QEMU on the constructor attribute used in the GNU C compiler. Functions which are marked with this attribute are used by QEMU to initialize its modules before the main function of the application is executed. Pointers to these functions are stored in the ELF section named `.init_array`.

In order to implement this functionality, a linker script used to link the user space applications had to be changed to allow the linker to store the `.init_array` section in the resulting ELF file. A fragment of linker script designed for such a task on a generic Linux platform was used. The source code in the `elf_load` module in the C library is responsible for loading the ELF file. The section `.init_array` has to be recognized by this module and its content has to be stored to an appropriate location.

The elf_load module was changed in such a way that the header of .init_array is recognized in the section_header function and in the process of loading an ELF file the content of this section is stored with its size in the elf_info_t structure. The content of this section is then copied to the PCB (Program Control Block) in the elf_create_pcb function. In this way it will be accessible after the ELF file has been loaded and the execution has started in the __main function in the libc module of the C library. This function receives a pointer to the PCB as its argument.

The very same function is responsible for calling the main function of the linked application. It is this function that was changed in a way that the functions addressed by pointers in .init_array are called before the main function. Similar changes were applied to support the .fini_array, with the difference being that functions addressed by the pointers contained in this section are called when the function exit is called. This function is guaranteed to be called when an application is terminated gracefully (function abort is not called).

3.5 Monitoring stdin

It was discovered that the select function is used to wait not only for pipe objects used as event objects, but also for events on stdin as well. As described in section 2.3.1 one of the events QEMU waits for in the event loop is an event triggered when data is available in the stdin data stream.

The solution chosen to implement this functionality is based on creating a new fibril that is blocked and waits in the C library version of the read operation. Each time the C library version of the read operation finishes and the fibril gets unblocked, the read event is signalled so that the event loop can process this event. Every file descriptor that is monitored in this way is provided with a structure containing: a buffer and a mutex that protects it, flags set by the ioctl function, information, how much data is requested to be read and three events – a read event, a write event and an exception event, to support the generic mechanism of the select function.

Data read after returning from the C library version of the read operation is stored in the buffer of the structure associated with the monitored file descriptor. The version of read from the POSIX layer of HelenOS uses a different source code to read from file descriptors that are monitored in this way. If the buffer associated with the monitored file descriptor has more data than requested, the data is read and removed from the buffer. If not, the requested field is updated by the amount of data requested when the read function is called and the code waits for the read event of the monitored file descriptor.

As mentioned in section 2.3.1 the locfs file system in HelenOS does not support parallel reads and writes, which is why instead of the C library version of the read operation the function console_get_event was used for reading from

the `stdin` in the end. The events returned by this function were represented as characters pushed in the buffer of the monitoring structure.

3.6 Implementing `pwrite` and `pread`

QEMU block devices make use of the functions `pwrite` and `pread`. To implement these functions in the C library of HelenOS this functionality must be supported in the VFS service in HelenOS. Therefore, the function `vfs_rdwr` in the VFS service has been extended to support a position specified by the caller instead of using the position associated with a file descriptor.

As the function `vfs_rdwr` already served multiple purposes it was not desirable to add another `true/false` argument and an enum `rdwr_flags` was used to substitute the original and the newly added argument. Using this method it is clear which functionality is desired from this function when looking at the function calls, without the need to go through the description of its parameters.

3.7 ANSI terminal commands

In order to port the `ncurses` library ANSI terminal commands had to be implemented. For the sake of the prototype the parsing of ANSI commands was done by adding a new module in the console service and the `gui` library in HelenOS. This ANSI module exports one function called `ansi_try_parse`. The function `ansi_try_parse` is expected to be called before the processing of a character which is to be written to the console, with this character as an argument. If the character obtained is part of a potential terminal command this function signals to the console that the character does not need any processing by returning a one. In case of an error in the middle of the currently parsed command, a zero is returned and the characters processed since the last successfully processed commands are printed to the console, to make the debugging of terminal commands easier.

The parsing itself is implemented using a simulator of a finite-state machine (FSM) with the additional ability to parse numeric arguments automatically. Macros are used to simplify the definition of the FSM and to achieve better readability of the code. As this is an invasive solution in terms of the console service and the `gui` library, it is considered as a temporary solution to support the prototype implemented in this thesis. A new service to support the terminal commands will be created in the future when the API of the console subsystem in the C library of HelenOS undergoes the changes planned by the HelenOS community.

3.8 Integration of the source code

The port presented in this thesis is a part of the HelenOS project. HelenOS uses the Bazaar version control system to track changes in the source code of HelenOS and to allow branching of the source code. A special branch (see attachment A) was created to implement the prototype presented by this thesis, to allow the main branch of HelenOS to be developed by the HelenOS community at the same time. To ensure that the changes in this branch are easily appliable to the main branch, the changes from the main branch were merged to the branch of the prototype at regular intervals.

A similar approach was taken with the source code of Coastline packages and the source code of QEMU. The modified source code of QEMU is published in a GitHub repository created for the sake of this thesis. The Coastline was branched in a similar way as the main branch of HelenOS and its branch can found on Launchpad in the HelenOS project.

4. Evaluation

At the end of the development of this thesis, the `libposix` library in HelenOS was extended and the `libevent` library was added to the system. The packages `qemu`, `pixman`, `libffi`, `glib`, `gettext`, `bind`, `iconv` and `ncurses` were created in HelenOS Coastline. This chapter describes the evaluation of the functionality added to implement the prototype of QEMU with the features described in the Analysis chapter.

4.1 PCUT tests

The PCUT framework proved to be very helpful when debugging and testing the newly implemented functionalities. It allowed testing for correct behaviour, finding deadlocks and the missing memory reclamation. The new test suites connected with the pthread, `pipe`, `select` and `sigjmp` functions were added to the `libposix` self test. A new test suite was created for the library `libevent`. The C library self tests were extended with tests for the bug connected with `fibril_yield`, tests for functions `pread` and `pwrite` and tests for the constructor and destructor attributes. Table 4.1 shows the exact numbers of tests added to different libraries in the process of the implementation of this thesis.

Library	The number of added PCUT tests
`libposix` – pthread	25
`libposix` – other	15
`libevent`	11
the C library	6

Table 4.1: The number of tests introduced to each library

The tests for the pthread functionality include stress tests creating over 100,000 threads in a row where each of them will be joined or detached before another thread is allocated. This helped in looking for a memory that had not yet been reclaimed.

4.2 Using GLib tests

Another way to test the pthread functionality and the `pipe` and `select` functions was through the usage of GLib's self tests. A part of the source code of the GLib library contains tests designed to test the library itself. As the configuration process is quite mature in the GLib library it is not hard to cross-compile these tests as a part of the `HARBOUR` script in the Coastline's GLib package.

The tests helped to discover, for example, problems with thread specific values, and the functionality offered by pthread discussed in section 3.2. The GLib test named `rec-mutex` helped the HelenOS community to fix two bugs in the kernel of HelenOS and one race condition in the C library of HelenOS, as mentioned in section 2.2.4. Some of the tests could not be used as they were dependent on the function `fork` typically used by POSIX compliant applications that is not supported by HelenOS, or on the pre-emptiveness of pthread threads.

4.3 Testing QEMU

QEMU as an application was tested using different operating systems which were given to QEMU to emulate. Different Linux kernels were used together with images of HelenOS and QEMU disk images from QEMU's wiki [14] which are recommended for testing QEMU. Each operating system makes use of its resources in a different way. This manifested in different problems connected with different operating systems. The problem that was discovered most often was a deadlock when QEMU was emulating instructions, but the emulated operating system was stuck in a loop. This kind of deadlock was discovered, for example, while emulating DOS 6.22. To resolve this deadlock another call to the function `firbil_yield` had to be added higher in the call-stack of the loop emulating the CPU, as explained in section 2.2.4.

Another problem that manifested during testing of the emulation of the MINIX operating system (in version 2.0.4 which is one of the QEMU testing images) was of a performance kind. It was discovered that the functions `pread` and `pwrite` are used heavily during the boot process of MINIX, at the moment when the file systems in the booted image are checked for corruption. These functions were first implemented using `lseek` as described in the first proposed solution in section 2.3.2. However, with this implementation MINIX was not able to boot properly in less than 30 minutes. For this reason the second solution proposed in section 2.3.2 was chosen. With this implementation MINIX was able to boot in less than two minutes.

Table 4.2 shows how long it takes the prototype of QEMU implemented in this thesis to boot different operating systems and comparison of these values taken from different platforms. The values are merely illustrative as the measurements were done using a stopwatch and the platforms used for testing were not completely uniform. Unless otherwise stated, a CD-ROM is used for loading the operating system, and the time measured is that between bootloader starting and a first expected user input. Based on these values this prototype is on average 8.7 times slower than QEMU when compiled for Linux or Windows.

System	HelenOS	Linux	Windows
HelenOS (VGA text buffer)	124,63	11,93	13,16
HelenOS (serial port) [1]	107,18	10,97	11,02
Linux kernel (3.2.41-2) – log [2]	49,04	5,73	5,5
Linux kernel (3.2.41-2) [3]	52,67	8,27	8,58
Minix 2.0.4 [4]	68,22	3,24	3,98
Dos 6.22 [5]	4,71	4,51	4,57
Tiny-Core Linux 6.3	103,4	16,99	17,05

[*] all values are in seconds
[1] values express the time until the first `getterm` task is spawned
[2] booted using `-kernel` and `-initrd` options, time from Linux kernel log used
[3] booted using `-kernel` and `-initrd` options
[4] booted from prepared hard drive image
[5] booted from prepared floppy disk image

Table 4.2: Comparison of the boot times of different OSs in QEMU

As a result of a slower emulation and dependency of QEMU on precise timing, the guest code often struggled with timers. The boot process of HelenOS ended up with kernel panic unless the frequency of the host system timer was increased. An image of Tiny-Core Linux starts printing a timeout error message when `udev` is trying to kill one of its child processes. However, this message disappears a few seconds after the `bash` shell is started.

As Tiny-Core Linux contains the `time` command, a utility intended to measure the wall-clock time of a started process, it was used to compare the speed of the prototype implemented in this thesis, when the operating system is successfully booted. Table 4.3 shows the wall-clock time of different commands in Linux, executed inside QEMU running on different platforms. These values should be considered as merely illustrative, as the platforms used for testing were not completely uniform.

Command	HelenOS	Linux	Windows
`time find /`	76.82	3.31	4.14
`dd` random to `file_4M`[1]	98.17	6.71	6.4
`time md5sum file_4M`	0.91	0.23	0.18
cpu and memory load [2]	242.63	11.47	13.24

[*] all values are in seconds
[1] `dd if=/dev/urandom of=file_4M bs=4M count=1`
[2] `time head -c 4m /dev/urandom | gzip > /dev/null`

Table 4.3: Comparison of execution times of different Linux commands in QEMU

All the tests were run with the default settings of QEMU in emulation mode, which means with one CPU core and 128 megabyte of RAM. HelenOS had an advantage over the other two platforms because it was loading the files from its ram disk.

4.4 Supported platforms

As mentioned in the Introduction chapter, the prototype implemented in this thesis should focus on supporting platforms x86 and x86-64. It is important to distinguish two kinds of platforms – a guest plaform and a host platform. The guest platform in the case of this thesis refers to a platform that is emulated by QEMU. The host platform, on the other hand, is a platform on which HelenOS is running. As HelenOS and QEMU support multiple platforms (e.g. x86, x86-64, ARM, MIPS, SPARC and PowerPC) testing of all the combinations would be very difficult. Table 4.4 shows the combinations of guest-host platforms that have been tested during the evaluation of this thesis (guest platforms in rows, host platforms in columns).

Guest	x86	x86-64
x86	T	T
x86-64	T	T
ARM	D	NT
MIPS	D	NT

[T] tested
[D] deadlock
[NT] not tested

Table 4.4: Testing of combinations of guest-host platforms

Table 4.4 mentions deadlocks when emulating ARM and MIPS platforms. Debugging of the prototype of QEMU discovered that these deadlocks are probably caused by the problems with pre-emptiveness explained in section 2.2.4. To test platforms x86 and x86-64 the images mentioned in table 4.2 were used. Guest platforms ARM and MIPS were not tested on the x86-64 host platform, however, their compilation was successful. The same applies to guest platforms SPARC, SPARC64 and PowerPC on either x86 or x86-64.

5. Conclusion

The port of QEMU implemented in this thesis is by no means a stable equivalent of QEMU built on Windows or Linux. Most importantly, it lacks the support of full VGA emulation and it does not support networking. Only a subset of those platforms emulated have been tested. These features were omitted because they are not required by the goals of this thesis and because their implementation would take a lot of time. It is, however, a prototype that is sufficient for testing operating systems on top of HelenOS including HelenOS itself.

The port of QEMU presented in this thesis can emulate a code of the same or a different platform to the platform of the host. The code of the emulated machine can be loaded through an emulated CD-ROM or hard-drive. The output of the emulated machine can be displayed to the user through the serial port or the VGA text buffer. The input is provided through the serial port, or through an emulated keyboard when the VGA text buffer is used as an output device.

This thesis also serves as a proof of the maturity of HelenOS. During the implementation, two bugs in the kernel of HelenOS were fixed as parts of this thesis were able to help the HelenOS community to reproduce these bugs. A bug in the user space threads, known as fibrils, was fixed and a race condition in the management of fibrils was discovered by the HelenOS community thanks to the code implemented. At the end of this thesis it was clear that HelenOS is mature enough to execute other operating systems on top of itself.

The primary goal of this thesis was to create a prototype of a port of QEMU for HelenOS that would allow the running and testing of HelenOS inside HelenOS itself. This was achieved as the port presented by this thesis can run HelenOS together with other operating systems – e.g. DOS, MINIX and Linux.

The secondary goal was to extend the POSIX layer in HelenOS and thus facilitate the porting of other POSIX compliant applications in the future. This was proved at the end of the implementation of this thesis when it was attempted to port the application `bash` to HelenOS. There were previous attempts to port this application that had failed when `bash` tried to call the `select` function on `stdin`. As this was one of the functionalities ported to HelenOS during the implementation of this thesis, `bash` was successfully compiled in a few hours. During the testing of `bash` it was discovered that some of the functionalities were working, for example the `echo` command gave the expected output. However, spawning new tasks did not work because it is dependent on the function `fork`, which is not supported by HelenOS and was not required by QEMU.

The thesis meets the goals specified in the Introduction chapter. The missing features of QEMU do not prevent the user from using the port to test different operating systems including HelenOS.

5.1 Possible extensions

The port presented in this thesis is a prototype and therefore, there are features that could be ported in the future. When the networking is supported in the POSIX layer it would allow the emulation of network cards and the usage of the GDB server inside QEMU. The emulated network hardware is convenient for testing the networking of the tested operating system. The GDB server allows for debugging of the source code of the guest, using GDB.

Another feature with which the port could be extended is support for the full VGA output. As discussed in section 2.4.2 a suitable solution would be to port the SDL library to HelenOS. As graphical user input (GUI) has never been used when porting POSIX applications or libraries to HelenOS this would require changes in the HelenOS Coastline. The special rendering driver for the SDL library could then be added to make use of the GUI in HelenOS.

A shared mode could be added to the function `pipe` that was implemented in the POSIX layer of HelenOS as a part of this thesis. When the problem with concurrent reads and writes to the locfs file system is fixed, each pipe could be represented by a special service. This would allow usage of the real file descriptors and for the source code of monitored `stdin` and created pipes to be joined.

The pthread functionality implemented in the POSIX layer of HelenOS can be extended to support pre-emptiveness, once the problem with executing fibrils in an environment with multiple kernel threads is fixed. The function `pthread_setconcurrency` could then be implemented to support controlling of the desired level of concurrency by starting new threads or by shutting down the already started threads.

Bibliography

[1] HelenOS community. *HelenOS*, http://www.helenos.org/

[2] HelenOS community. *PC – HelenOS*, http://trac.helenos.org/wiki/IPC

[3] Jermář, Jakub. *Implementation of file system in HelenOS operating system*, http://www.helenos.org/doc/papers/HelenOS-EurOpen.pdf

[4] HelenOS community. *HelenOS sources*, http://www.helenos.org/sources

[5] *The Open Group Base Specifications Issue 7*, http://pubs.opengroup.org/onlinepubs/9699919799/

[6] Horky, Vojtech. *PCUT: Plain C Unit Testing mini-framework*, https://github.com/vhotspur/pcut

[7] Hajnoczi, Stefan. *QEMU Internals: Overall architecture and threading model*, http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html

[8] *Pixman*, http://www.pixman.org/

[9] *dtc/dtc.git – The Device Tree Compiler*, https://git.kernel.org/cgit/utils/dtc/dtc.git/

[10] Yufei, Chen. *QEMU Source Code Notes*, http://chenyufei.info/notes/qemu-src.html

[11] *Ticket #507 – Kernel assertion fail at phone_deallocp()*, http://trac.helenos.org/ticket/507

[12] *The Definitive KVM (Kernel-based Virtual Machine) API Documentation*, https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt

[13] Nagarajayya, Nagendra and Gupta, Alka. *Porting of Win32 API WaitFor to Solaris*, http://www.madchat.fr/windoz/coding/winapi/waitfor_api.pdf

[14] *Testing – QEMU*, http://wiki.qemu.org/Testing

A. Source code

The HelenOS community uses the Bazaar version control system. The official source code repository of the core of the operating system and the kernel can be found at:

```
bzr://brz.helenos.org/mainline
```

The official repository of the Coastline containing the packages of the ported applications can be found here:

```
bzr://helenos.org/coastline
```

QEMU uses the Git version control system and the repository containing its source code can be found at:

```
https://github.com/qemu/qemu
```

The development branches of HelenOS are accessible on Launchpad, again using the Bazaar version control system. This thesis respects this custom and therefore, the source code of the core of HelenOS with the changes presented in this thesis can be found in this repository:

```
lp:~maresja1/helenos/qemu_th
```

The Coastline repository containing the new packages can be found at:

```
lp:~maresja1/helenos/coastline_qemu_th
```

The source code of QEMU based on version 2.2.0 with changes applied during porting can be found here:

```
https://github.com/maresja1/qemu-helenos
```

Branches of the core of HelenOS and HelenOS Coastline were regularly merged with the official branches to facilitate the incorporation of the code of the development branches into the official branches. The branches of the Coastline and the core of HelenOS are in a feature freeze state to avoid problems whilst testing this thesis. However, a fresh version is accessible in the following repositories:

```
lp:~maresja1/helenos/qemu_porting
```

```
lp:~maresja1/helenos/coastline_qemu_porting
```

The process of compilation of the sources and execution of the prototype is described in chapter B.

B. Compilation and execution

To successfully compile the prototype implemented in this thesis, the source code of HelenOS core and HelenOS Coastline must be compiled using a cross-compilation tool-chain. The tool-chain is expected to be found in `/usr/local/cross/ia32/`. The cross-compilation tool-chain is expected to consist of binutils in version 2.23.1, GCC in version 4.8.1 and optionally GDB in version 7.6.1. HelenOS core is provided with a script to facilitate the installation of the cross-compilation toolchain. The script can be found at `tools/toolchain.sh` in the root of the HelenOS core source code repository. To install the tool-chain for x86 platform run:

```
tools/toolchain.sh ia32
```

The script will check the dependencies of the tool-chain, download the source code of its parts from their official repositories, compile it and install it to the aforementioned folder.

When the cross-compilation tool-chain is prepared the compilation of HelenOS and the prototype of QEMU can take place. Let us assume that the repository of HelenOS was downloaded to folder `helen_os` and the repository of Coastline to folder `coastline` and these two folders have the same parent folder. In order to compile Coastline, a new folder has to be created in the same parent folder, let us call it `coastline_build`. The sources of HelenOS can be compiled using Coastline and that is also the recommended method when packages from Coastline are going to be compiled. The following procedure should compile HelenOS, QEMU and its dependencies.

```
cd coastline_build
../coastline/hsct.sh init ia32 build
../coastline/hsct.sh install qemu
../coastline/hsct.sh install ncurses
```

At this point everything should be compiled. To ensure that the applications installed by Coastline into the `helen_os` folder are included in the CD-ROM image of HelenOS it is advised to run `make` in this folder.

To start the image of HelenOS in QEMU, run the following command in the `helen_os` folder:

```
qemu-system-i386 -cdrom image.iso -m 1G
```

The whole process of building, including the cross-compilation tool-chain, is captured in the script `build.sh` on the attached CD-ROM, for details see chapter C.

C. Content of the attached CD

The CD-ROM attached with this thesis contains the following files and folders:

- `thesis.pdf` – the text of this thesis
- `image.iso` – pre-built image of HelenOS containing the prototype of QEMU created in this thesis
- `build.sh` – script designed to facilitate the compilation of the HelenOS core and QEMU in HelenOS Coastline, the first argument of the script specifies in which folder the compilation should take place
- `run.sh` – script designed to facilitate the execution of the image of HelenOS, the first argument specifies the folder used when calling `build.sh` script, if it is omitted the pre-built image on the CD-ROM is used
- `sources/helenos_core.patch` – a patch containing the differences between the branch of HelenOS core used to implement this thesis and the trunk of HelenOS core
- `sources/helenos_coastline.patch` – a patch containing the differences between the branch of HelenOS Coastline used to implement this thesis and the trunk of HelenOS Coastline
- `sources/qemu.patch` – a patch containing the differences between the branch of QEMU used to implement this thesis and the branch corresponding to version 2.2.0 in its official repository
- `sources/helenos_core` – a content of the branch of HelenOS core used to implement this thesis
- `sources/helenos_coastline` – a content of the branch of HelenOS Coastline used to implement this thesis
- `sources/qemu` – a content of the branch of QEMU used to implement this thesis
- `data` – scripts to facilitate the starting of different operating systems in QEMU on HelenOS
- `data/images` – images of different operating systems used to demonstrate the abilities of the prototype of QEMU implemented in this thesis

The scripts in the `data` folder are copied in the final image when using the `build.sh` script or the pre-built image of HelenOS CD-ROM. The usage is simple, for example, to start an emulation of the attached image of HelenOS use:

```
batch data/start-helenos.bdsh
```

in the console of HelenOS.