Comenius University in Bratislava
Faculty Of Mathematics, Physics And Informatics

Testing Framework for HelenOS

Master's thesis

2013

Bc. Martin Sucha

Comenius University in Bratislava
Faculty Of Mathematics, Physics And Informatics

Testing Framework for HelenOS

Master's thesis

Study programme:	Computer Science
Field of Study:	2508 Computer Science, Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Jaroslav Janáček, PhD.

Bratislava 2013

Bc. Martin Sucha

82065340

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Martin Sucha
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Testovací framework pre HelenOS

Cieľ: HelenOS je mikrokernelový operačný systém, ktorý podporuje viac než poltucet rôznych procesorových architektúr. Pri vývoji takto multiplatformového kódu sa stáva, že kód na niektorých platformách prestane fungovať'.
Cieľom práce je preskúmať možnosti automatického testovania rôznych subsystémov operačného systému HelenOS, či už na reálnom hardvéri alebo v niektorom zo simulátorov, a navrhnúť systém, ktorý by umožňoval automaticky spúšťať a vyhodnocovať takéto testy.

Vedúci:	RNDr. Jaroslav Janáček, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	21.10.2011

Dátum schválenia: 02.11.2011 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....
študent vedúci práce

82065340

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Martin Sucha
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Testing Framework for HelenOS

Aim: HelenOS is a microkernel operating system that supports more than half a dozen of different processor architectures. In the course of developement of such multiplatform code, it may happen that the code stops working.
The goal of this thesis is to explore different possibilities of testing HelenOS subsystems, whether on real hardware or one of supported simulators, and design a system that will allow these tests to be automatically run and evaluated.

Supervisor:	RNDr. Jaroslav Janáček, PhD.
Department:	FMFI.KI - Department of Computer Science
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	21.10.2011

Approved: 02.11.2011 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

....
Student

....
Supervisor

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature.

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

. . . .

Acknowledgements

I would like to thank my supervisor RNDr. Jaroslav Janáček, PhD. for reading through the preliminary versions of this thesis and providing valuable suggestions.

I would also like to thank all the developers of HelenOS for making it an interesting project to contribute to.

Thank you.

Abstrakt

HelenOS je mikrokernelový operačný systém, ktorý podporuje viac než poltucet rôznych procesorových architektúr. Pri vývoji takto multiplatformného kódu sa stáva, že kód na niektorých platformách prestane fungovať. Rozhodli sme sa preskúmať možnosti automatického testovania rôznych subsystémov tohto operačného systému, či už na reálnom hardvéri alebo v niektorom zo simulátorov, a navrhnúť systém, ktorý by umožňoval automaticky spúšťať a vyhodnocovať takéto testy. Popísali sme základné súčasti operačného systému HelenOS, ktoré je potrebné poznať, aby bolo možné takýto system navrhnúť. Taktiež sme v krátkosti zhrnuli vlastnosti niektorých systémových emulátorov, ktoré sa dajú použiť na spúšťanie tohto operačného systému. Rozobrali sme vlastnosti, ktoré by mala naša implementácia frameworku na spúšťanie testov v HelenOS podporovať. Na základe našich záverov sme navrhli zmeny v tomto operačnom systéme a implementovali framework na automatické spúšťanie a vyhodnocovanie testov.

Kľúčové slová: HelenOS, operačný systém, testovanie

Abstract

HelenOS is a microkernel-based operating system that supports more than half a dozen of different processor architectures. In the course of developement of such multiplatform code, it may happen that the code stops working. We decided to explore different possibilities of testing HelenOS subsystems, whether on real hardware or one of supported simulators, and design a system that will allow these tests to be automatically run and evaluated. We described those basic components of the HelenOS operating system, knowledge of which is required to design such system. We also briefly summarized the properties of some of the system emulators that can be used to run this operating system. We analyzed which properties should be supported by our implementation of the framework. Based on our findings, we designed changes to the operating system and implemented a framework for running and evaluating tests.

Keywords: HelenOS, operating system, testing

Preface

During the course of developement of the HelenOS operating system, it happens that regressions are inadvertently introduced to the codebase. Since HelenOS supports a wide variety of different configurations and platforms, those regressions rest often unnoticed for a long time. Sometimes, features of the operating system stop working because of regressions in the system emulators that are used by the developers.

HelenOS contains simple support for running tests, but there is no support for launching the tests automatically. This means the tests are not executed as often as they could. We decided to create a framework for running tests of this operating system so that the tests can be automatically executed inside a virtual machine.

We also wanted to be able to test individual components of the operating system in various ways. Since the original code did not allow for some types of the tests that could be used, we saw an opportunity to extend the possibilities of testing of HelenOS.

Contents

List of Figures

List of Tables

Introduction

Quality assurance is an important aspect of developing a product — this applies also to software as to any other category of products. Testing allows to produce more stable and reliable software, which is why many software projects run a batch of tests before release. But tests are useful not only when releasing a product, but can be used also in an iterative way during multiple phases of a software develepement life cycle.

Performing a set of tests is a tedious task and when performed manually, it may also be prone to human errors, therefore commonly run tests are usually automated so that these tests can be run more conveniently and more often. Automated testing allows to run tests continuously, on every change of the source code, and is commonly referred to as continuous integration.

Testing a software that is very dependent on its environment — such as an operating system — is a complex task as one needs to ensure that the product works correctly in all possible configurations of the environment. While usually an operating system tries to abstract of those differences, the developers must be very careful to write code that works on multiple architectures. Despite various coding practices, it happens that new code is not as platform-agnostic as expected, such as when the developer uses incorrect-size integer types or expects certain assumptions to be true.

Operating systems usually contain a vast amount of generic code, which may be tested just like other software. There are, however, various special cases such as boot code, system specific routines, etc. that require more sophisticated approach to be tested automatically. Several generic operating system testing frameworks exist, that employ various methods to test the behaviour of an operating system from outside (usually involving the use of a system emulator).

HelenOS, a microkernel-based multiserver operating system, does currently contain very limited support for automated testing. HelenOS has some unit tests for kernel and user-space data structures and functionality, but there is no support for isolation of these tests or reporting the results outside of the virtual machine.

We would like to investigate various possibilities of extending testing of HelenOS and its various subsystems so that tests can be easily executed from a developer's machine or repeateadly executed on a continuous integration server, which should allow early notifications of possible regressions and/or errors in newly added code that fails on an architecture not tested by the developers themselves.

Text organization

The text of the thesis is organized in chapters as follows:

Chapter 1 introduces some key concepts of testing software.

Chapter 2 gives an overview of HelenOS and its subsystems relevant to the subject of this document.

Chapter 3 briefly describes emulators that can be used to run HelenOS and their features.

Chapter 4 focuses on what a testing framework for HelenOS should look like and refines goals for implementation.

Chapter 5 outlines changes that were necessary to make to HelenOS and challenges we faced. This chapter also provides implementation details of the framework.

Chapter 6 compares our implementation with other frameworks for automated testing of operating systems.

Chapter 1

Testing software

Testing software before release is a standard practice that helps to ensure quality of the software product. It is an important part during a developement cycle, which may require significant amount of effort and resources.

When the product is a piece of a large software, the complexity of possible inputs and outputs increases rapidly, because components may interact in different ways, which creates an instance of a large combinatorial problem. To ensure no bugs are present in a general software, the only option is to execute an exhaustive search over all possibilities and check them, which is not very viable for large software projects.

There are several possibilities how to test software for presence of bugs[24]:

Unit tests check low-level logic consistency of the code in small unit modules, e.g. an individual procedure. Those tests are usually written by the developer of the code they are supposed to test. As the code is being modified, the unit tests are updated to match in an iterative manner. It is important that the unit tests stay as isolated as possible so that when a test fails, it is possible to relatively quickly pinpoint the piece of code that contains an error.

Integration tests are done in a broader scope, targeted at a specific function of the system larger than a module (a whole subsystem, for example). These tests try to run all paths through code, for example by emulating an interface the subsystem expects, including various error handling code.

System tests check the behaviour of the entire system, with all subsystems, functions and interfaces as in the final product. These tests check if the system meets its specifications.

Performance tests are executed in a controlled environment which tries to minimize outside factors that could affect the results, to ensure the tests are repeateable. Stress tests are executed to test the behaviour of the system at higher loads.

Static analysis may be performed on the code to find some common programming errors without running the code in question, but these techniques also only provide approximations of the results because the halting problem is undecidable.

When testing a code that depends on other parts of code, the dependencies are commonly replaced with a piece of code allowing the tests to test effects of the tested code, while obeying the interface of the replaced code. Such stub code may support examination of state or behaviour of the tested code.

In order to test various error code paths, a technique called fault injection exists. This technique introduces faults at various points in the code to test code that is rarely executed, such as error handling paths[25].

A code coverage analysis may be performed to determine the amount of code that was excersized during a test. Code coverage tool may determine which branches of code are executed and therefore know which source code lines were executed during a test.

When testing operating systems, a system test may be executed in a virtual machine, observing its behaviour to different stimuli coming from the virtualizer. For example, the test may drive a virtual mouse to a specific location on the screen, try to press a button, and observe the changes that this action caused via the virtual computer screen. Other functionality of the virtual machine can be also used, including virtual network, serial ports and other devices.

Running tests inside a virtual machine is convenient, but the tests may be also executed on the real hardware as system simulators do not precisely simulate all of the aspects of the real hardware[26]. For example, a simulator may be more permissive and allow certain combination of instructions that the real hardware may not be able to handle. Developers of system simulators usually have to make a design decision to tradeoff between level of detail of the simulation (and corectness) and speed of execution. Compare for example a program simulating logical gates in the processor and an instruction level simulator. The former may emulate timing characteristicts of the system more precisely, while the latter may be executing faster at the expense of ignoring those details. It is therefore beneficial to execute the tests on the real hardware of the target platform, at least once in a while. However, running the tests on real hardware may require some special peripherals for capturing screen, etc. that can be more easily manipulated in a virtual environment.

As part of a continuous integration[12] practice, the tests of a codebase are executed often to ensure the latest versions of the code are still behaving as expected. Building and testing the software periodically, perhaps even every revision of the software commited to the source code repository, brings benefits of early notifications of build and test errors, which the developer may not have noticed prior to submitting the changes. A continuous integration server may even execute tests that are not available to the concrete developer, for example because the hardware necessary to execute the tests is not in their posession.

Software for supporting continuous integration may usually be configured to trigger a build/test run of the software depending on many conditions, from simple time-based events, to watching source code repositories for changes. After such an event occurs, the software proceeds to build the software, usually running its test suite in the process. The resulting artifacts of the build, such as executable binaries, installers, disk images, test results, etc. may be collected and made available to the users via the software. Reports of the results may also be reported directly to developers using e-mail, instant messaging or SMS notifications, depending on what the CI software supports.

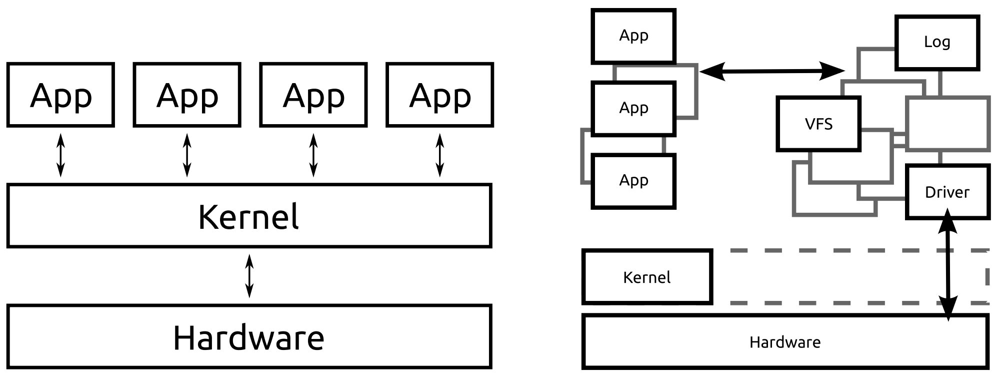
Chapter 2

HelenOS overview

HelenOS is a microkernel-based multi-platform multi-server general purpose operating system[1].

Operating systems can be categorized by their composition, depending on which parts of the code are executed in the privileged mode of a processor. Traditionally, operating systems with monolithic kernels put much of the code directly to the kernel. This means that programming errors in a single subsystem or even a single device driver may cause a catastrophic failure and render the whole system inoperable. In microkernel-based family of operating systems, where HelenOS belongs, the subsystems that are not required to run in privileged processor mode are executed as ordinary user-space tasks. Yet there is no strict line between monolithic and microkernel-based operating systems and hybrid implementations exist that combine the two approaches. Even systems that claim to be strictly monolithic started to allow some drivers to be implemented in user-space (e.g. Linux and its FUSE infrastructure for filesystem drivers).

Some microkernel-based operating systems put only as little code to the kernel itself as possible, going to the extremes such as even switching tasks is implemented in user-space (these may be called nanokernels or picokernels by some people). HelenOS



(a) A monolithic system (b) A microkernel multiserver system

Figure 2.1: Difference between monolithic and microkernel multi-server operating systems

Table 2.1: Architecures supported by HelenOS

Platform	Bits	Endianness	SMP support
amd64	64-bit	little endian	Yes
arm32	32-bit	little endian	No
ia32	32-bit	little endian	Yes
ia64	64-bit	little endian	Yes
mips32	32-bit	little/big endian	No
ppc32	32-bit	big endian	No
sparc64	64-bit	big endian	Yes

is on the opposite side of this spectrum, where the kernel is responsible for initialization of all the processors, task switching, memory management, interprocess communication, etc. HelenOS kernel may be even compiled with support for a debugging console that allows for the kernel to be controlled by keyboard commands. The kernel console features a set of commands useful to developers, which allow to inspect various aspects of kernel state, ranging from version information, list of CPU and memory resources to displaying state of a slab allocator. `kconsole` is only a debugging aid for the developers and is not present in production releases.

Unlike many microkernel-based operating systems, HelenOS supports a variety of processor architectures and platforms. The supported architectures differ in endianness, native integer size, support for SMP[1] and other factors. HelenOS can be run on different hardware configurations, from modern desktop computers, servers, different developement boards or a mobile phone to a computer resembilng old 486 processor architecure. A short overview of processor architectures HelenOS can run on may be found in Table 2.1.

HelenOS has a multi-server userspace. This means that different subsystems and drivers are separated in their own task and communicate via kernel- provided IPC[2] mechanism. In case a device driver misbehaves in such system, only the task belonging to the driver is killed by the operating system and the rest of the system remains unaffected. Some operating systems, such as MINIX 3, allow to automatically restart such tasks[16]. HelenOS does not implement this policy as its authors belive that the drivers should work in the first place.

HelenOS tasks can be divided between servers, which are tasks that provide some services and applications, which consume those services and are usually started by the user. HelenOS has many servers that together provide various level of functionality, ranging from essential IPC services to whole subsystems, such as virtual filesystem or device driver framework, which also launches driver servers on demand[17].

Each task in HelenOS may be composed of one or more independently preemtively scheduled execution threads. HelenOS libc library also has a notion of fibrils,

[1]symmetric multi-processing
[2]inter-process communication

which are lightweight execution entities cooperatively-scheduled by libc among the task's threads. The kernel is not aware of fibrils as they are entirely implemented in userspace.

In this chapter we describe concepts of the HelenOS IPC and servers that are essential in the HelenOS' userspace environment and which every program expects to be present. Additional information about the user-space environment relevant for building our framework, such as initialization process of the system, is also included. A description of original means to execute tests is included for completeness.

2.1 HelenOS IPC

In HelenOS, inter-task communication[3] is based on an abstraction of a telephone dialogue between a man and an answering machine[18].

Each task has several phones and a single answerbox used for communication[4]. A phone may be connected to an answerbox of a different task[5]. When the phone is connected, the task may make a call that is stored into the (remote) answerbox.

Every call has to be picked up and forwarded or answered at some time after it is stored in the answerbox. Eventually, the call will get answered. The answer is stored in the answer box of the original caller task which completes the call. Answers are not answered and the task has to make a new call if it wants to respond to an answer.

At the most basic level, a call is composed of a few integers of size native to the architecture HelenOS is running, which is either 32-bits or 64-bits wide on the currently supported architecures. One of the integers is special because it contains an identifier of an IPC method that this call represents.

Most methods are defined by the receiving task, but the first 512 method identfiers are defined by the kernel and expose special functionality of the IPC mechanism. If the kernel observes that a task sent a call with such method, it reacts according to the semantics of the method.

For example, if a task wants to copy a memory buffer to another task, the kernel notes the position and size of the buffer specified in the call and when the call is answered with acknowledgment, it copies the buffer to the location specified by the receiving task in the answer. If the task answers with an error, nothing happens.

This way, the kernel exposes IPC functionality to:

- send/receive large data
- offer/ask to share memory between tasks

[3]HelenOS uses the term task to refer to what is known in other systems as a process. IPC is used in other contexts and the abbreviation is used in HelenOS, hence a slight inconsistency in terminology

[4]a separate kernel answerbox may be used for debugging

[5]it is also possible for the phone to be connected to an answerbox of the task owning the phone, but this setup is not used in HelenOS

- share phone connections
- notify a third task of an agreement to change state

Apart from calls and answers, the HelenOS IPC also supports kernel notifications, which are one-way messages from kernel that are not answered. They are used for example to notify device drivers about interrupts.

2.1.1 Asynchronous framework

Since calls are delivered to single answerbox of a task, it is necessary to route calls and answers to correct fibrils. Since implementing a state automaton to do the routing using callbacks is not convenient, HelenOS provides a framework doing all the state automata and route the messages to the correct fibril. If the fibril waits e.g. for IPC answer, the async framework may schedule other fibrils in the meantime. When an answer arrives and is processed, the framework eventually switches back to the fibril this answer is destined to.

2.2 Spawning and connecting to other tasks

We have described how the tasks can use their phones, but how do they obtain connections to services they depend on?

This section aims to explain how the two services that are used for this purpose work and how the naming service is involved in spawning new tasks.

2.2.1 Naming service

The most low-level of servers is the naming service `ns`, which allows other processes to estabilish connections to singleton[6] servers and a loader service. These servers correspond to different subsystems such as the filesystem layer, logging service, device driver framework, network, etc. The naming service also plays an important role in the process of spawning new processes and waiting for their completion.

Let us first describe how a process estabilishes a connection to another service by its identifier. Every process has an initial IPC phone connected to the naming service, which allows it to make a call asking for the connection. The naming service then looks up its internal dictionary of services that have registered and, if it finds a matching entry, forwards the connection request to the target service. The connection request may then be accepted or denied by the target service, as it sees fit. Actual work of estabilishing a new phone is done in the kernel, which observes the communication between the parties and figures out whether they have agreed to estabilish a new connection. In case the service that was requested has not registered

[6]there is only a single instance of the server running in the system

with the naming service yet, the connection request is postponed until the service registers itself. This process is shown in the figure 2.2.

Spawning a new program is a two step process. If someone wants to spawn a new process, they must first obtain a connection to the loader service. This is not a singleton service, but instead an identical clone of the service is spawned in a new process for each of the clients separately. When a new connection to the service is requested, the naming service instructs the kernel to spawn a new copy of the loader task, which then registers at the naming service. At this point, the naming service learns about the task id of the loader. The naming service then proceeds to forward the waiting connection.

When a connection to the loader service is estabilished, a client then proceeds to customize the new task. The loader allows to set the name of the task, arguments, path to the executable, file descriptors and finally to load the program.

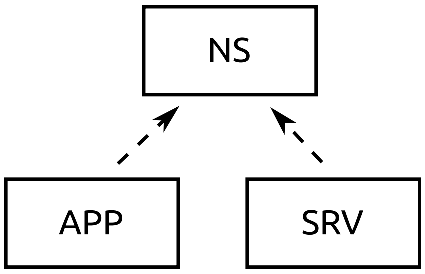
The client may then want to wait until the new task finishes and check its return value. This facility is also provided by the naming service. It allows every task to set its return value before it exits. In case a task fails to set its return value, the naming service learns about this fact, too, because it is the receiving end of the initial phone. A phone automatically sends a hangup message whenever its connection is about to be terminated, which happens at the latest when the task exits. As soon as the naming service knows either the return value or the fact that the task unexpectedly terminated, it replies to tasks waiting for the return value. This allows server tasks to set their return value just after initialization, essentially starting to run in background when run from a shell.

2.2.2 Location service

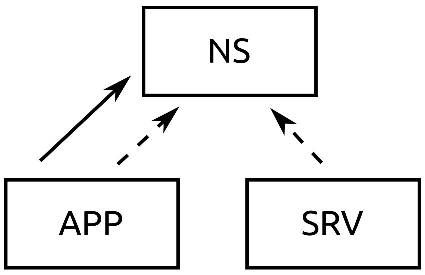
A higher-level counterpart to the naming service exists in HelenOS. Just like the naming service, location service enables clients to connect to other services. Instead of using a flat namespace of numeric service IDs, the location service allows services to be identified by string names.

Each fully qualified service name (FQSN) consists of an optional namespace, separated from the rest of the string by a slash, followed by the name of the service. Namespaces are not hierarchichal, therefore `locsrv` does not allow multiple slashes to be present in the FQSN. If no slash is present in the FQSN, the namespace is an empty string. The FQSN uniquely identifies the service, so an attempt to register an already registered name fails.

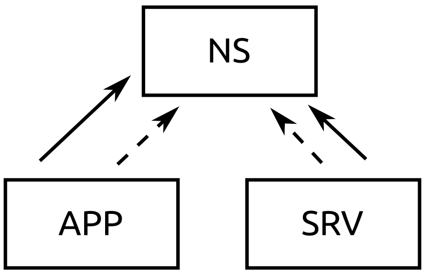
Naming the services by string names is not the main purpose of the location service though. Apart from the list of services, the location service maintains a list of categories. Categories are identified by simple string names and their purpose is to group services with compatible interfaces. For example, a category named `serial` contains a list of services representing different serial ports of the computer. The location service maintains a M:N relationship between categories and services so not



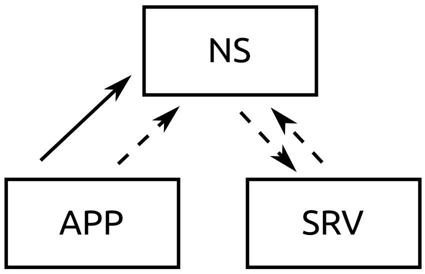
(a) Only initial phones are connected.



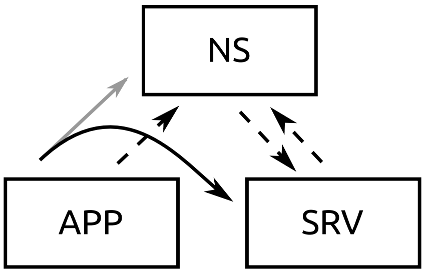
(b) APP makes a call to NS requesting connection to SRV, but it may not be available yet.



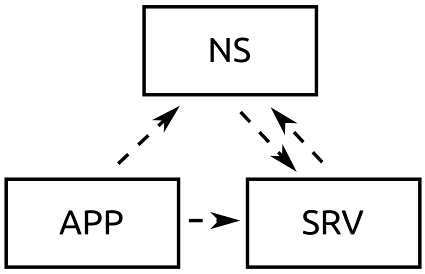
(c) SRV registers at NS.



(d) This created a connection in the other direction.



(e) The original call from APP is forwarded to SRV via the new connection.



(f) A new phone is connected.

Figure 2.2: Connecting to a service via IPC. The dashed lines represent connected phones. Solid lines represent unanswered calls.

only a category may contain multiple services, but a single service may be enlisted in several categories. This is useful if a single service provides different interfaces to a single resource, for example a multi-function printer with an integrated scanner may be present in both `printer` and `scanner` categories.

The location service allows clients to browse its directory of connected services, namespaces and categories. This allows a generic tool `loc` to be present so that users may easily discover the services that are present in the system.

When a service is added or removed from a category, the location service notifies any clients that have registered to observe those changes. This allows different classes of services to be discovered by relevant parties. For example when a human interface device is plugged into the system, it registers in the relevant category and the server processing input events may start using the device automatically.

2.3 Virtual filesystem service

A central component of the filesystem support framework in HelenOS is the virtual filesystem server (VFS). Every application that wants to access the filesystem must first connect to `vfs` and then use the connection to send filesystem commands.

Files are organized in a hierarchical namespace with single root directory, not unlike UNIX-like operating systems behave. A filesystem may be mounted over a directory to expose its contents. The filesystem operations are implemented to an extent that applications can work with the filesystem (reading/writing files/directories). However, it currently lacks support for features such as symbolic links, access rights or storing file modification times.

The virtual filesystem server keeps separate file descriptors for each process. If a client wants to hand a file descriptor off to another task, it has to use a special IPC protocol so that the virtual filesystem server is notified of the change (and both the receiver and the vfs authorized such an operation).

Support for each filesystem type is provided by separate servers. These servers are called filesystem drivers and conform to an interface which the virtual filesystem server uses to handle requests from user applications. Each driver registers with the virtual filesystem server upon startup and announces information about the filesystem type it supports.

Usually[7], the filesystem is backed by some block device, either a ramdisk or a partition on a hard-drive. Block devices are also provided as a service by separate servers.

[7]The `tmpfs` filesystem is an exception

2.4 Logging service

Another one of the essential services that are present in the HelenOS userspace is a possibility to log different events in the system. The `logger` server provides exactly that.

Each message that is sent to the logger is tagged with a log level. This is a number indicating the severity of the message. Currently, 6 levels are supported ranging from severity of a fine debugging message to catastrophic failure.

The service maintains a hierarchy of log channels, each having a string name, optional parent log channel, and severity level. When a message is sent for processing, it is targeted to a specific channel and the effective log level of the channel is computed. This value is then compared to the severity level of the message and if the message is not severe enough, it is discarded.

Channel's effective log level is computed as follows:

1. If the channel's log level has been set explicitly, use it as its effective log level.
2. If the log level has not been set and the channel has a parent, compute the effective log level of the parent and use it.
3. If the channel has no parent, use the global default log level.

If the system determines that the message needs to be logged, it writes it to the kernel output character buffer and to a file, if available. Each of the top-level log channels has a corresponding file under the `/log` directory. Messages destined to channels deeper in the hierarchy go to the file of the corresponding top-level channel and are labeled with a fully qualified name of the channel[8].

2.5 User-space initialization

A set of initial user-space tasks is spawned by the kernel upon startup. These tasks are passed to the kernel by the boot loader. There are three special cases:

- all spawned tasks are connected to the first one
- loader task is not spawned
- if the image is not executable, it is used as the initial ramdisk image

There are a few differences between tasks that are spawned during initialization and tasks spawned later:

- The initial phone connection of the first task is not created as there is no task to connect it to.

[8] a slash-separated list of log names

- Boot-loaders may pass task command line to the kernel. However, the kernel passes tasks' arguments via kernel sysinfo mechanism instead of setting them to be available to a task via arguments of its `main` method. This is a known issue.
- No file handles are passed to the tasks.
- Task's name is prefixed with `init:` for user's convenience

The tasks that are built into the boot image and used during initialization are the naming service (`ns`), the location service (`locsrv`), the virtual filesystem server (`vfs`), `logger`, ramdisk block device (`rd`), a filesystem server implementing support for the filesystem format used in the initial ram disk image (`ext4fs`, `fat` or `tmpfs`) and `init`

The `init` task initializes parts of the userspace environment that are not required during the bootstrap process. This task is responsible for spawning other userspace services such as `clipboard` server, input and output servers, mounting other filesystems, etc. depending on the configuration.

2.6 Tests

The original code for running tests is divided into two independent components.

The kernel console contains a command called `test`, which is a very simple test runner for kernel tests. Another kernel console command, `bench`, can execute a single test multiple times and report kernel accounting information. The tests may be run either individually or all safe tests at once and report results as text output into the kernel console, which is useful when running the tests manually. On the other hand, an automated program may not be able to easily extract test results and progress information from the kernel. This program supports marking tests as unsafe. This is for tests such as `fault1`, which cause fatal harm to kernel and therefore would disrupt execution of the batch.

The amount of kernel tests is currenly small, 18 tests are present in HelenOS mainline, revision 1396. Parts of kernel covered by tests include printing using printf, semaphores, SLAB tests, memory mapping, frame allocator and kernel data structures test including AVL and B-tree implementations.

For testing user-space, HelenOS contains a simple application that runs tests. As with kernel test command, the userspace `tester` can execute either a single test or all tests marked as safe. Each test is a function that either returns success or an error message. The tests output status information to the standard output of the tester process, which is also the only place the results of the tests are reported.

Because the tests are executed in the same process as the logic that is reponsible for launching the tests, any crash in the test causes the whole tester to terminate.

2.7 Communication

HelenOS has support for framebuffer and keyboard/mouse. This is a primary user-interface that is used by most HelenOS users. Despite being a microkernel OS, HelenOS has some framebuffer drivers in the kernel. However, efforts are underway to move all those drivers to userspace.

In userspace, experimental support for GUI has been added by Petr Koupý (previously, HelenOS only had a text console). On some platforms where a framebuffer is not available, text-only console is still available.

HelenOS contains user-space and debugging-purpose kernel serial port drivers. Only driver for NS16550-compatible serial ports is available via HelenOS' user-space device driver framework so far.

A network stack is also available in userspace, although it still lacks many features available in other operating systems. Drivers for three different network interface cards are already available. Some simple network utilities such as ping are available as well as a demo web server. `remcons` server allows to connect to HelenOS using telnet protocol.

Information may be also written to storage devices such as ATA or SATA based hard drives. HelenOS has support for several filesystems, including ext4, mfs and fat. Filesystem driver for reading ISO 9660 filesystem is also present.

Last but not least, HelenOS contains a USB stack that allows to use USB devices from HelenOS. Currently, human interface device class (keyboards, mouse devices) and mass storage class are supported. No driver for USB serial port adapter or network adapter is present in HelenOS.

Chapter 3

Machine emulators and their interfaces

In this chapter, we briefly look at emulators that could be used to run HelenOS and describe interfaces to control them and observe the system state.

3.1 QEMU

QEMU[2] is the emulator of choice of most HelenOS developers. It supports most of the architectures that HelenOS can run on and some others as well. HelenOS can be run in QEMU using system emulation for ia32, amd64, arm32, ppc32 and sparc64[1]. While both QEMU and HelenOS support mips32, the exact machine types supported are different. There are plans[23] for HelenOS to support a machine emulated by QEMU, though.

Apart from emulating processors, the virtual machines emulated by QEMU have also peripheral devices including a video card, keyboard and mouse devices, storage devices such as hard drives and optical drives, serial ports, network interface cards and more.

Those devices can be configured via QEMU's command-line arguments when launching the virtual machine. Some of the devices may be added, removed, reconfigured or manipulated even when the virtual machine is running.

QEMU supports several different interfaces that allow other programs to communicate with the system in the virtual machine. Video, keyboard and mouse can be accessed using VNC or SPICE remote desktop protocols. Additionally, QEMU monitor allows users to execute commands to control the emulator. These commands also allow to set mouse position, send key presses via the virtual keyboard, or save an image of the current screen video output to a file.

Since QEMU monitor interface was originally designed for humans to use, it is not well-suited to use by computer programs. Fortunately, a monitor interface designed to be controlled by programs was added. QEMU monitor protocol[4] is an asynchronous

[1]sparc64 support in QEMU is rather experimental[3]

protocol to execute QEMU monitor commands and receive asynchronous event notifications. Every message in this protocol is a JSON formatted line, which allows programs to parse the contents using libraries available for a lot of programming languages.

Virtual serial ports may be connected to various host devices, including pipes and sockets that could be conveniently opened by a program. QEMU supports exposing multiple serial ports if the simulated machine supports it.

Network may be configured either to use QEMU user-mode network stack that creates a virtual network with access to outside networks using network address translation or create a virtual network device in the host. Network traffic can be logged to a file in a libpcap format that can be read by common network analyzing tools.

It is possible to directly manipulate memory of the virtual machine either by monitor commands to dump memory, or by connecting an instance of GNU debugger (`gdb`) to QEMU, which also allows writing to the memory of the virtual machine (if the virtual machine is currently stopped). QEMU also supports setting code and memory access breakpoints using the debugger.

3.1.1 Controlling gdb

The GNU Debugger connected to QEMU can be controlled using GDB machine interface[5] (GDB M/I). This is a line-based text interface designed for usage by programs where the debugger is only a part of the entire solution. Is is used for example by various integrated developement environments.

This interface allows to use GDB features to find location of symbols in memory, read and set values of variables, set breakpoints, etc. and receive events to observe state of the debugger and the debugger program.

3.2 VirtualBox

VirtualBox[6] is a system virtualizer for ia32 and amd64 architectures. It also supports emulation of common hardware for video, keyboard/mouse, serial ports, network, etc. It uses parts of QEMU code to emulate devices and the network stack is very similar.

VirtualBox provides its own API to manipulate its state to an extent similar to QEMU, except that it only contains an unsupported internal debugger[7].

3.3 VMWare Workstation

VMWare Workstation is a commercial virtualization product for amd64 architecture. It supports similar user-facing functionality as previous tools. Notable feature of this software is a mechanism to instrument the virtualizer or the virtual machine using small scripts called VProbes[8]. This functionality could be used to inspect and modify the state of the operating system running inside the virtual machine without needing to pause it (as is the case when a debugger is used to add a breakpoint).

3.4 MSIM

MSIM[9] is an instruction simulator for the mips32 architecture. It has support for simulating several simple hardware devices including a keyboard and character printer. The emulator does not currently support a serial port, but it is listed as a planned feature in the project's TODO file. MSIM has integrated debugging facilities as well as a connector allowing to connect GNU Debuger (see also 3.1.1). This emulator also can be executed from within HelenOS.

Table 3.1: Some of the devices emulated by emulator by platform[3, 7, 10]

Emulator / Platform	Video	Keyboard/Mouse	Storage	Serial	Network
QEMU					
PC (ia32, amd64)	PCI VGA	PS/2, USB	IDE HDD	NS16550A	e1000, rtl8139, ne2k_isa
Integrator/CP (arm32)	PL110 LCD	PS/2	MMC SD card	PL011 UART	SMC 91c111
PowerMac (ppc32)	PCI VGA	ADB	IDE HDD	N/A	ne2k_pci
sparc64	PCI VGA	PS/2	IDE HDD	NS16550A	N/A
VirtualBox					
PC (ia32, amd64)	PCI VGA	PS/2, USB	IDE, SATA HDD	NS16550A	Am79C973, e1000
MSIM					
MIPS R4000 (mips32)	N/A	custom kbd	custom block	write-only	N/A

Chapter 4

Analysis

In this chapter we look at what is the intended usage of a testing framework in HelenOS, from the point of view of target users – contributors to the HelenOS codebase – as well as what are the problems we need to tackle and their high-level solutions.

4.1 Intended use cases

There are several use cases when a testing framework may be used during developement of an operating system such as HelenOS. As HelenOS is a microkernel-based operating system, a great portion of developement is focused on user-space components. Efforts are underway to allow HelenOS to be self-hosting in a sense that HelenOS can be compiled from within HelenOS. As of now, HelenOS contains a port of a portable C compiler project (PCC) that allows to compile simple user-space applications directly inside HelenOS. A port of the GNU Compiler Collection may be expected in the near future. Support for other programming languages than C is also planned for HelenOS – several projects to support programming languages such as Python, Go or Sysel are in various stages of developement. As code is developed within HelenOS, developing and using accompanying tests should be possible to aid the developer.

HelenOS already supports a few processor architectures and the developers strive to cover most of the platforms that are common or pose interesting challenges to solve. When porting the operating system to a new platform, the developers need to port the platform-specific code early. After the kernel is ported and seems to work, it is good to check if all the subsystems work as expected. In debug build, the kernel contains a debugging console that allows the developer to interact with the kernel even when the userspace is not available yet. However, having such a debugging tool in the kernel requires drivers to be present in the kernel, which goes a little bit against the microkernel design. A developer may want to still run tests before proceeding to work on the userspace part of the porting effort, though.

Since most developers only use a single platform most of the time, they may not test their code as thoughroughly on architectures or configurations other than those they prefer. A useful use-case is to execute all the available tests as part of a continuous integration process on as many architecures as possible. Running tests automatically in this way is not supported and is one of the main motivations for this thesis. The practice to execute tests on various architectures could help the developers spot errors in code as they are introduced.

4.2 Services of a testing framework

A typical testing framework usually provides:

Test enumeration Because tests may be added and removed as necessary, a test framework must be able to discover tests that are available and convey this information to the user.

Managing a life-cycle of tests The testing framework manages the entire process of running the tests. Meaningful subsets of all available tests can be selected to execute. When a test is about to be executed, the necessary environment is prepared first, then the test is executed, evaluated and the environment is prepared to run the next test.

Isolation of tests Tests are usually isolated from the framework so that adverse effects of bugs the tests are supposed to check for, or simply an error in the test itself affect only the test itself and not the entire framework. Tests are also isolated from each other so that execution of one test does not affect the results of other tests. The latter is necessary to be able to execute only a subset of the tests, in any order.

Providing a test interface A framework usually also provides application programming interface to the developers of the tests that allows to conveniently write the tests. This API defines how the the tests look, how they return their results and provides procedures to help doing common tasks when writing tests.

Reporting the results When the tests are finished, a testing framework informs the user of the results of the tests and reasons why some tests possibly failed. The testing framework may also generate log in formats understandable by other software or produce reports intended for human consumption.

Since our test framework will run tests in a virtual machine, we need to look at:

Controlling the virtual machine The framework needs to execute a virtual machine, observe its state, control its execution and shut it down when tests are finished.

Communication Since the tests are run inside HelenOS, it is necessary to be able to communicate with the system and observe its state from outside (a computer running the virtual machine or possibly another computer connected to the real-one running the tests)

4.3 What to support?

We need to decide which types of tests the framework will support, and how the support will be achieved.

Unit tests test small pieces of code such as procedures. As such, a unit test may be simple procedure that calls a function and checks its return values. In some cases, it may be necessary to replace implmentations of some lower-level functions, in this case it is possible to build a special version of the tested function to the test, either by using preprocessor macros or replacing symbols during linking. In kernel, modifying the code to be tested by replacing symbols may not be feasible, however, as this may disrupt the kernel environment (some functions require execution of special priviledged instructions or modify global kernel state).

Since kernel tests need access to the code they test, it is not possible to isolate the tests in a separate adress space. On the other hand, in user space, the test runner may spawn a new task that runs a test and reports the results to the framework.

There are several ways how to approach integration tests in HelenOS. One of them is to create stub libraries that will replace parts of API for different subsystems to test them separately. Since HelenOS is a microkernel multiserver operating system, the subsystems are already separated in the userspace. Each subystem resides in a separate server or a group of servers. We can exploit this fact and test the behaviour of the subsystems by replacing some of them with stub servers. This allows to check both whether a server behaves as it should and whether a client library code that wraps the low-level IPC into a higher-level API adheres to the IPC protocol.

If we want to be able replace servers which a program connects to, we could replace a function that creates the connection to the server. This will require modification of the code we aim to test, e.g. link a different library. It should be possible to redirect the very first IPC connection every task uses to obtain other connections. This approach should allow a single component to redirect all connections to servers the tested code requests and return versions used for testing. Having such a facility would even help to isolate the tests from the framework (and the rest of the system) even more.

To run system tests, it is necessary to test the system end-to-end. Things that could be tested include for example that after plugging a device to the system, the device is correctly recognized and is working. Such tests require some action to be taken from outside of the virtual machine to add the device or an action by the user if the tests run on a real hardware. Because of this, we will need each of the tests to be able to request such an external action and wait for it to be completed before proceeding further in the test. While one may think that such functionality is not necessary in the kernel, there might be some devices that could be tested this way. Consider for example that adding a hot-pluggable processor might require some actions from the kernel that could be tested.

When the tests can pause at some point, upon resuming, they shall receive information whether the waiting was successful. This can be used by the external tester to indicate that the requested operation failed or is entirely not supported (e.g. because some virtual machines do not support some operation). This notion can be extended and we can use such points to allow the tests to be aborted (a test has to be able to exit at this point anyway, if the waiting failed).

Computing code coverage requires to determine which branches of code got executed during tests. This may be achieved in several different ways[27].

- The code may be dynamically augmented before execution to store the outcome of branching instructions. This approach requires a program that is able to modify the machine code. Such an operation is architecture dependent so it will require a separate implementation for each platform. It also requires careful attention to details because it may cause the original program to misbehave. For example, certain instructions on some architectures require another instruction to be present in the immediate surroundings and if such an instruction is not there, the system may behave in an uninteded way.

- The code may be augmented during compilation. This requires a separate version of the code to be built when it is tested. Augmenting the code may still cause problems in the kernel, as it can disrupt its operation. Therefore, code coverage for only parts of the kernel could be computed this way.

- Modifying the virtual machine to determine branch information transparently to the executed programs[28]. One of the advantages of this approach is that it does not require modifications of the tested code, as the determination of branch outcome is done by the virtual machine. This approach has its drawbacks though. While this approach can easily determine branches in the kernel, determining outcome of branches in the user-space code will most likely require the code to be aware when kernel switches task contexts. Moreover, adding this functionality to the machine simulator will probably require adding support for every architecture separately.

Adding support for computing code coverage seems to be a significant amount of effort to do properly for different architectures and we will not pursuit this path in this thesis.

Considering possibilities to communicate with the system running inside a virtual machine and the test runner, we should be able to use serial port and network stack for our communication needs, at least on ia32 and amd64 platforms. When a serial port driver is added for an architecture where it is currently missing, we should be able to use it for communication. Fortunately, there is an interface that serial ports implement, so while we can focus on ia32 architecture, it is reasonable to expect that drivers for serial ports supporting this interface will be added in the future for other architectures as well.

But we shouldn't limit the possibilities of the communication only to serial ports. As computer networks are everywhere in this age, allowing the communication to happen over this channel seems natural. There is yet another possibility how to extract test results and other information from a virtual machine.

Since drivers for serial ports and the network stack resides in the user-space it is not possible to use it directly from the kernel to report test results. Instead, the user-space test runner should be able to enumerate kernel tests that are available and execute them just like the tests from the userspace.

Both QEMU and MSIM claim to support debugging via GDB, it should be possible to use the debugger to read tests results from the virtual machine and even to change several kernel variables – the most interesting is changing init task parameters even on architectures where the boot loader does not allow this.

During testing the framework should record information about the events happening in the system and store them in case test failures need to be analyzed. Those events should be also reported outside the virtual machine along with test results.

It is possible to drive the tests either from outside of the virtual machine or by a component running inside HelenOS. If the main component is a test runner outside of the virtual machine, it can send commands that are executed in the system. In the latter case, the program that runs outside of the virtual machine acts as a auxiliary component that provides services to the main runner. Advantage of the second approach is that tests can be run directly from within the tested operating system even if no virtual machine is used.

4.4 Summary

To sum up, using our framework, it should be possible to:

- Execute tests from outside the virtual machine as well as using a command directly in HelenOS, main test runner will be the user-space one.
- Allow running kernel and user-space tests and report results.
- Extract log messages and test results using serial ports and/or a debugger.
- Store log messages and possibly other events alongside test results for further examination.
- Allow tests to wait for external action to happen.
- Isolate the user-space tests inside an IPC sandbox.

Chapter 5

Design and implementation

In this chapter we present design decisions and implementation details that led to the creation of our testing framework for HelenOS. Since the framework should execute kernel and user-space tests, and optionally report results ouside of a virtual machine, it spans these three environments. The chapter contains three sections describing the changes we made in them. Last section contains a description of the protocol we used for communication between user-space test runner and its counterpart running at the developer's machine.

In order to execute tests in a virtual machine and record their results, it is necessary to have a program that figures out correct parameters for the virtual machine, configures it and executes the operating system. Log events and information from running the test needs to be obtained. The first section contains details about this component.

Changes in the kernel were required to suport all the features of the framework. It was necessary to prepare a logging subsystem that would allow to work with log messages in a unified way across kernel and userspace. Since test run in kernel cannot be isolated very well, the basic principle of the original testing code is preserved. However, we have rewritten the kernel test runner from scratch so that it is decoupled from the kernel console, supports features for system tests and can be controlled from user- space. The changes are described in section 5.2.

We made also changes to various user-space components, where the main test runner resides. Section 5.3 explains how runtime information is collected, how the tests are executed and describes changes necessary to report the test results. This section also contains information about how we approached isolation of the tests and what kind of new tests it makes possible to use in HelenOS.

5.1 External environment

In order to run tests inside HelenOS, it is necessary to first setup the emulated environment in which the operating systems executes. This is one of the tasks that `tools/run.py` program is responsible for.

The first thing the program does is to determine the actual configuration of the system. Since the configuration script itself is also written in python, it was not a problem to re-use the functions for loading the configuration, although it required some tweaks.

The program then executes the system in a virtual machine, monitoring and processing all incoming events such as log messages, status of running tests, etc. All such events are also logged to a file in case they need to be analyzed by a human.

The runner is structured into several modules. The most basic module is an asynchronous core, which runs the program's main loop and waits for events such as I/O or timeouts. These events are then dispatched to appropriate handlers that were registered during program initialization. The core provides facilities to manipulate input/output channels in a unified way and plug them into other modules to use.

The `core.py` module provides two implementations of the core interface, one using the traditional UNIX `select` call. However, `select` has a limitation in its API because it uses a bitmask to specify file descriptors to be monitored. This causes it to be linear with respect to the highest file descriptor number monitored. The other implementation uses a call to `poll`, which has slightly different interface which is linear with respect to the number of file descriptors monitored. If the latter implementation is available, it is used by default, otherwise the program falls back to the former one. There are other cases when another type of core may be used. For example, a graphical application written using the GTK toolkit may use main loop implementation from the library. The current design allows to write such a module and use it instead of one of the current core implementations.

A channel is a unidirectional or bidirectional stream of bytes. Each channel holds a list of clients to be notified when new data arrives from the channel and a buffer holding bytes to be written to the channel. The bytes from the buffer are flushed by the core as soon as the operating system indicates that the channel is able to accept data. There are implementations for socket client, socket server and file-based channels. The only restriction when implementing another channel type is whether the channel is backed by a file descriptor usable by operating system's `select` or `poll` calls.

As the program processes various events asynchronously, we added a separate module where events and supporting code resides. The `events.py` module contains definitions of classes for all event types and a simple event dispatcher. The dispatcher allows other modules to selectively subscribe to receive messages and receive them as calls to their methods, which is convenient and allows for nicely formatted python code.

A separate module is responsible for launching a QEMU virtual machine with command-line parameters determined according to the current HelenOS configuration. The user of the class representing the QEMU instance may hook various channels to virtual devices such as serial ports or I/O channels provided by QEMU itself. This module also contains a class for controlling the virtual machine using the QEMU monitor protocol, which is briefly introduced in section 3.1.

Communication with the agent running inside HelenOS running in the virtual machine is handled by a module for encoding and decoding protocol buffer messages used in the communication protocol between the two parties. The remote control protocol (RCP) used to remotely control the agent is described in the section 5.4. We decided to write our own decoder and encoder because the official python implementation [21] didn't suit our needs.

First of all, that implementation didn't support storing unknown message attributes, which is a feature that is desirable for our runner program because it allows it to embed all messages into its log file even if it does not understand the contents of the messages. This enables those messages to be later analyzed by human and helps to achieve forward-compatibility of the test runner.

Secondly, using a custom implementation allows us to tailor the implementation to the asynchronous core we use. Moreover, it is then straightforward to reuse parts of the code to encode and decode the envelopes of RCP messages.

Last but not least, adding an external dependency to the project adds maintenance burden to the developers, who then need to ensure compatibility with a range of versions available in the wild. Embedding the dependency into the project does not help either, as it enlarges the effort necessary to setup a developement environment and the developers have to still track the upstream code.

We also implemented a proof-of-concept module that gathers log messages from the kernel log buffer using GNU Debugger, which is connected to the virtual machine interface for remote debugging. This module uses the GDB/MI protocol (see also section 3.1.1) to communicate with the debugger. Unfortunately, setting breakpoints in the virtual machine works correctly only in recent versions[1] of QEMU and we were succesful only with the i386 architecture.

Again, we used a custom implementation of the encoder/decoder of the GDB/MI protocol, because python implementations available at the time this code was developed did not fit our needs. Some new python GDB/MI implementations have emerged since then, however.

In order to extract new log messages, the `gdb.py` module sets a breakpoint inside the function that logs the messages. The location of the breakpoint is currently hardcoded inside the module, but this can be changed in the following ways:

- add a special comment that the runner would search for
- split the function in question into two and set a breakpoint to the entry point of the function representing the second part

The program reads the log buffer, which is an array of bytes and parses the log messages itself. This means it has to know the size of native integers stored in the buffer. Currently only 32-bit integers are supported, because setting breakpoints

[1]version 1.2.0 or later should be sufficient

only worked on 32-bit architecture, but as soon as the problem with breakpoints is resolved, it should not be a problem to modify the code to use integer width based on the build configuration.

The gdb module also sets a breakpoint to the location where a kernel panic function resides, so that it knows when such event occurs.

We also investigated the possibility to set arguments of the initial user-space tasks via debugger. These arguments may be set by the boot loader on some architectures, such as ia32 or amd64 where GRUB2 is used. Some architectures do not use a boot loader that allows this, however. Even if such support existed it would be necessary to setup the bootloader to communicate with the launcher program (or leave this responsibility to the kernel).

As it turns out, we are able to set arguments for an initial task in this way by modifying the respective variables in the kernel upon startup. This functionality could be used to change the mode of operation of various system components during testing without rebuilding the boot image.

Using the debugger for obtaining information about the system has several advantages and disadvantages. One of the advantages is that it does not require a special kernel device driver to communicate this information. On the other hand, the setup requires line debugging information to be computed during the build process in order to work[2] and pauses the virtual machine for a short amount of time whenever a breakpoint is hit.

A similar approach would be to embed the code monitoring the state of the virtual machine directly to the virtualizer so the delays are eliminated. While modifying QEMU code should provide better performance, it will require maintenance of patches against upstream QEMU that would have to be tracked and rebased whenever a new version of QEMU is released. An ideal solution would be if QEMU allowed to inject a probe pseudocode that would be interpreted and stable across QEMU versions. Similar functionality is provided by VMWare's vprobe technology[8].

5.2 Kernel space changes

5.2.1 Logging subsystem

HelenOS traditionally used simple character output buffer for its logging needs. This approach makes it hard to copy information from the kernel log to structured user-space logger service. Therefore, we decided to implement a structured log in the kernel as well. However, the old character buffer remained and is used by the debugging output drivers if they are enabled. The log messages are transparently copied to the kernel output buffer.

Kernel log messages are stored sequentially in a cyclic buffer. Once the buffer is full, the oldest messages are discarded as necessary. Since messages are stored

[2]otherwise the debugger would not know the address where to set a breakpoint

Table 5.1: Format of log messages stored in the kernel log buffer

Data type	Width	Field
size_t	native	Length of the message, in bytes
uint32_t	32-bits	Message ID
uint32_t	32-bits	Facility ID
uint32_t	32-bits	Log level
uint8_t[]	variable	Contents of the message

with metadata about their size, originating facility and severity level, it is possible to manipulate them as a whole. A message is either retained in its entirety or dicarded completely. This fact, combined with atomic access to the log structure, guarantees consistency for an agent retrieving the messages.

The kernel log API can be divided into a lower-level interface and a higher-level one. The first allows flexibility while the other provides convenience for basic usage.

While it would be possible to just have a single function that takes a character array to write to the log, this approach is very limiting. Consider a situation where the log message must contain a variable number of parts which are best generated in a loop. Such a message would have to be prepared in an external buffer, which may require allocation of the memory. This allocation may not be possible early in the initialization process or after memory has been exhausted[3]. Therefore, it is necessary to atomically append directly to the cyclic buffer, while allowing the appending part to be done using multiple calls. As one of our design goals was atomic access to the log, it is necessary to surround the block of code accesing the log with a pair of commands locking/unlocking the corresponding mutex. However, as the messages are preceded with a header, it would be necessary to always add a call to write the header just after the call locking the mutex when appending to the log, so we decided to embed the locking calls inside `log_begin`/`log_end` calls, respectively. This also allows the length of the message to be automatically computed after the message has been appended to the buffer. While the structure of the log message allows to store any binary data, it is convenient to store textual content to make debugging easier, so we added `printf`-like function to append formatted text to the log.

So, the low-level API looks as follows:

```
log_begin(facility, level);
log_append(data, len);
log_append(another_data, another_data_len);
log_printf(fmt, ...)
...
log_end();
```

[3]This is the reason why the log buffer is allocated statically

While this API is flexible, it is necessary to make at least three calls to log a message, which is inconvenient when one wants to just log a simple message. Therefore a high-level wrapper named simply `log` combines the three calls into one. It is a `printf`-like function with the following signature:

```
int log(facility, level, fmt, ...)
```

When a user-space client wants to retrieve kernel log messages, it specifies the last message ID it saw and the kernel returns only messages with a greater ID. This concept does not reveal any internal information about how the kernel stores the messages, therefore it is possible to change the implementation in a backwards compatible way between releases. Not only this allows only new messages to be returned, it allows the client to determine whether some messages were missed, because they were discarded by the kernel[4], in this case the ID of the first returned message will be greater than the last ID the client saw. While the counter for message IDs may overflow, it should not cause many problems in practice. Although the current implementation does not check this case, the kernel can easily determine that the counter has overflown, as it may remember the IDs of latest and newest messages stored in the cyclic buffer and return appropriate messages to the client.

This design turned to be beneficial also when reading the kernel log buffer directly via a debugger, as the external test runner can do the same handling of log message IDs and determine which messages are new since it last checked the buffer.

5.2.2 Test runner

While most generic kernel code could be taken out of the kernel and tested entirely in user space (in fact, several library functions, such as handling of strings, generic algorithms and data structures are exactly the same or very similar in kernel as in user space runtime library), we opted not to do so. The main reason for this is that the kernel has a different runtime environment than user space processes. The environments differ in several ways, most notable being implementation of memory management and a variety of different synchronization primitives. Although these share a similar interface, there are subtle semantical differences which are caused by slightly different needs of the implementations. Reproducing the kernel environment accurately in user space therefore seems to be work with diminishing returns.

The kernel tests are statically included into the kernel binary during the build process if debug build with tests is enabled in the project configuration. While simple to implement and convenient for use if the size of the tests is relatively small, this may not work that well when the tests (and all required data) reach a certain size limit depending on the limitations of boot loading implementations on some HelenOS supported platforms. Fortunately, this scenario is currently unlikely[5] and

[4]The current implementation does not check this though

[5]The user space initrd image is currently significantly more likely to cause problems like this as more and more user space code is ported from different projects

this situation may be avoided either by only embedding a subset of tests at once, or loading the kernel tests dynamically. The latter is not much viable, however, as HelenOS kernel is not able to load kernel modules dynamically and this functionality may even never be implemented because it goes against the microkernel principle.

As there may be some platform specific tests that require interaction from the outside world (such as pressing a button on the device, inserting a hotplug capable processor, etc.) for testing certain functionality of the kernel (although this currenly applies only to debug-only kernel drivers), the kernel tester allows each test to be suspended and wait to be resumed by an external stimulus. This is triggered by any of the tester subsystems for controlling test execution, described later in this section. A test may be suspended only at a predesigned named point. Moreover, we defined semantics of the wait such that the test itself must be able to handle a failure of the waiting and terminate prematurely if requested. A test may therefore be resumed in one of the following states:

- the waiting was successful, continue the test
- the external action failed or is not available, fail/skip the test
- a request to abort the test suite occured, abort the test

This behaviour is implemented by running the tester itself in a separate kernel thread, which allows the tests to be expressed naturally using C functions and the wait points to look like a function call. In addition to allowing the test suite to be paused, running the test suite in a separate thread allows us to leverage support for kernel accounting mechanism and determine how many CPU cycles each test consumed.

As the information about CPU usage is recorded, the user may use a test suite for benchmarking purposes. The tester allows to specify how many times each test will be executed. If the test passes all repetitions of the test, it is marked as passed. Otherwise, as soon as an error or another test result condition (such as the test is skipped) is encountered, the test is immediately terminated and no further invocations are made.

One of the subsystems to control the test suite may be the kernel debugging console (if compiled into the kernel). It allows the user to execute various commands to examine and control the state of the kernel for debugging purposes. The subsystem is not supposed to be present in production builds. In the meantime, it allows convenient interaction with the kernel and the tester registers several commands to control its behaviour – execute the test suite, query status, pause/resume and abort the test suite.

While the kernel console is suited to be operated by the kernel developer, there are better ways how to control the test suite from the user space. The tester exports system calls to setup and control the test suite and also hooks itself into several user-space-facing subsystems. Information about all the tests available in the kernel is exported through the sysinfo subsystem, which allows user space processes to query various system properties through a convenient user-space API.

The manipulation of the tester state from user space is done using dedicated system calls, one for preparing the test suite (i.e. selecting which tests are to be run) and another one for querying, pausing, resuming and aborting the test suite. The kernel notifies the user-space test runner when the state changes (such as when the tests are suspended) via asynchronous event mechanism. In the early developement phase, the system call for controling the tester was blocking, which was suited for testing, but using asynchronous notifications not only obviates the need for multiple threads in the user-space tester application, but also allows us to leverage the benefits of the user-space async framework.

Our design specifically includes a global symbol pointing to the most recent test run, so that test results can be located and read from the kernel memory by using a debugger. Combined with setting breakpoints to function in the kernel test runner, it should be possible to observe the state of a test run.

For running just the kernel tests before the user-space is initialized, the kernel checks for a variable specifying the tests that should be executed during startup. The variable is currently not set in the kernel itself, but is left to be changed by a debugger. It can be set in kernel if such need arises, e.g. by setting a multiboot kernel parameter.

5.3 User space changes

5.3.1 Logging subsystem

While HelenOS' orignal user-space logging subsystem was mature, our work still required some adjustments. First of all, the original `logger` server stores all logs into files under `/log` directory. This is suitable if the only consumer of the log messages are the users themselves. On the other hand, if the consumer is a machine, this behaviour is not very optimal, as there are many log files that will require parsing. Therefore, we extended the logging server to allow an external consumer to register a callback connection where all logged messages are reproduced.

Apart from allowing log messages to be consumed, it is necessary to synchronize user-space and kernel-space log messages to achieve full observability. For example, if something fails during user-space initialization, a user (and possibly a machine monitoring the system) may not be able to interact with the user-space components. On the other hand, synchronizing kernel log messages to the user-space log allows to use richer capabilities provided by the user-space, such as getting the log messages via a network interface.

Kernel log messages are synchronized to user-space logger by the `klog` server, which gets notifications from kernel whenever a message is appended to the kernel log. Since all messages in the kernel log are tagged with facility code, the `klog` server can easily determine which messages from the kernel log originated in the user space. Such messages are discarded to prevent infinite loops with a message bouncing between kernel and user-space logs. Messages originating from the kernel are classified according to facility and logged to the appropriate log channel.

The other direction of the synchronization process in not that straightforward, however. While the `klog` server may easily register for notifications of log messages from the user-space logger, some messages may be lost. This is caused by the race between first messages arriving to the logger server and the `klog` server registering for notifications. Since such a loss of log messages is not desirable (they provide valuable information for diagnosing failures), our initial implementation worked around this problem by passing the responsibility of synchronization to the `logger` server itself. This setup has its own disadvantages, though. One of them is that the `logger` must be able to know which are the channels that `klog` uses. Another one is that turning the synchronization on or off requires special handling in the logger.

The proper solution is to ensure that a client may obtain the log messages despite connecting at a later time. Just like the kernel maintains a buffer of recent messages, the logger now maintains a backlog of messages as well. This allows the logger clients registering for receiving messages to specify whether they are interested in getting the recently logged log entries. While some messages may still be lost in case of excessive logging early in the user-space initialization process, this should not happen in the usual case. For the case of logging messages to the kernel logger, this is not a problem at all since the kernel discards old messages anyway if its buffer reaches full capacity.

In order to better correlate kernel and user-space messages[6] in the future, both may be annotated by a timestamp obtained from the kernel uptime (the kernel itself does not have a notion of a wall-time, which is not necessarily monotonic anyway).

5.3.2 Task output

Traditionally, if a task's output and error streams were not written to a file or console, they were automatically redirected to the kernel character output buffer by HelenOS C library. A disadvantage of this approach is that the messages printed by various tasks may in principle interleave in the kernel character output buffer, although it rarely happened [7]. This functionality was used for logging purposes and was already superseded by the user-space logging service that existed prior to our work. However, the option to use the character output remained and so there was not enough incentive to migrate the tasks to use the logger service.

We removed support for user-space tasks to directly write to the kernel output

[6]note that the notifications about log messages are asynchronous in both kernel and user-space, therefore the system maintains causal relationship only within either the kernel or user-space log, but not between them

[7]This issue could be seen when we used a debugger for reading information from this buffer, as stopping (pausing) the virual machine apparently caused slight differences in timing that could trigger this behaviour

buffer (log messages are still copied automatically to output) and any output that was not redirected to a file (or console) is discarded.

5.3.3 Test runner

The user-space test application is composed of several components, each providing a separate functionality. A test runner component manages the lifecycle of a test run and uses components providing backends to run the tests. Currently, providers are available to run tests from kernel or user-space. A component for creating and maintaining user-space sandboxes is described separately in 5.3.4. Last but not least, the application needs to communicate its status and test results, so components for user interaction via console and remote control server (see 5.3.5) integration are provided.

One of the first things the `test` application does is that it initializes all test providers and enumerates all tests that are available. If the user did not supply any command-line arguments, the list of tests is simply printed to the program's standard output. Otherwise a test run is initialized using a test specification provided via command-line. Either console or remote control support is initialized next to control and observe the test run. At this point, the test run is started and the program's components begin to execute asynchronously.

As far as the generic test runner component is concerned, a test is described by its name, description, a flag indicating whether it is safe to execute this test in a batch, and the provider that it is associated with. This information can be used to initialize a test run structure, which keeps information about tests that are scheduled to be run, their results and information pertaining to the current state of test run execution.

The list of tests to be executed is created based on a pattern that is specified by the user (or program) that launched the `test` application. The pattern is matched against qualified name of every test that was discovered during the enumeration process and if it matches, the test is added to the list of tests to be executed. The qualified name of the test is composed of the name of the provider and the name of the test separated by colon.

Figure 5.1: Example qualified name of a test

```
directory:tester-spawn-logger
```

For the pattern it is possible to either specify a single qualified test name or match multiple tests. When a single name is specified, only the single test matches, regardless of the setting of its safe flag. In case a pattern that could match multiple tests is specified, only tests that are marked as safe can match.

The exact matching algorithm works as follows:

1. The pattern is a provider name, optionally followed by colon and a test name.
2. Match the provider name as follows:
   - If the provider name ends with an asterisk, strip the asterisk and test whether the result is a prefix of the provider name.
   - Otherwise, match the entire provider name.
3. If the optional test name was not provided, return the match result, but don't allow unsafe tests to match.
4. Otherwise, the pattern is in the form `provider:test` and the same matching strategy is used for test names.
5. Return the result, allow unsafe tests only if neither pattern contained an asterisk at the end.

This matching algoritm allows for cases when a user wants to to execute all available tests, restrict tests only to specific provider, tests of a specific subsystem (if all tests testing the subsystem start with the same prefix) either in a single provider or across all providers (pattern like `*:tester*`). Of course the algorithm could be extended to support a list of patterns so that user can enumerate a list of tests to execute. In fact, it would be probably best to use a library for matching regular expressions, but there is no such library available in HelenOS at the moment.

The test providers supply implementations of operations to enumerate tests, initialize and control execution of the provider's tests. The provider for kernel tests uses system calls to control the kernel test runner described in section 5.2.2.

Userspace tests are stored inside `/tests` directory of the filesystem as executable programs with associated metadata in a separate file. When the tests are enumerated, all files with `.test` file extension in this directory are parsed for test definitions. This allows tests to be added simply by copying the appropriate files into this directory, which can be helpful when developing or installing new programs in the system. Moreover, the tests are not limited to a single programming language.

Each test metadata file may contain test definitions for multiple tests in a simple, yet extensible file format. A definition begins with a line containing the name of a test. This is the name that is visible in the user interface and is also passed as the first command-line argument to the test's binary. The name is followed by a description, again on a single line. A variable number of options may follow, one per line. Multiple test definitions are separated by a single empty line.

The tester uses a special-purpose IPC protocol to communicate with the tests, which allows it to control the test. The protocol is bidirectional and tests report their results and possible waiting points. A test may also indicate that it expects to crash. This inverts the tester logic in case the test crashes. Any logging in the tests is done using the user-space logging framework (which is redirected in the sandbox,

Figure 5.2: Example test metadata file

```
tester-simple
Simple test that always passes

tester-simple-unsafe
Simple test that always passes, but is marked unsafe
unsafe

tester-spawn-logger2
Tester should not spawn logger
logger disable

tester-sandbox-forward
Test checking sandbox forwarding of clipboard
```

but this is transparent to the test, see the next section). Standard I/O streams of the tests are not connected, so their output is discarded. There are several reasons for this. First of all, there is no support for pipes is HelenOS. Secondly, when run in a sandbox, the virtual filesystem instance is different, so this approach will require an additional agent in the sandbox to forward the traffic. Last but not least, using the logger service is advantageous because the log messages are labeled with severity and source information.

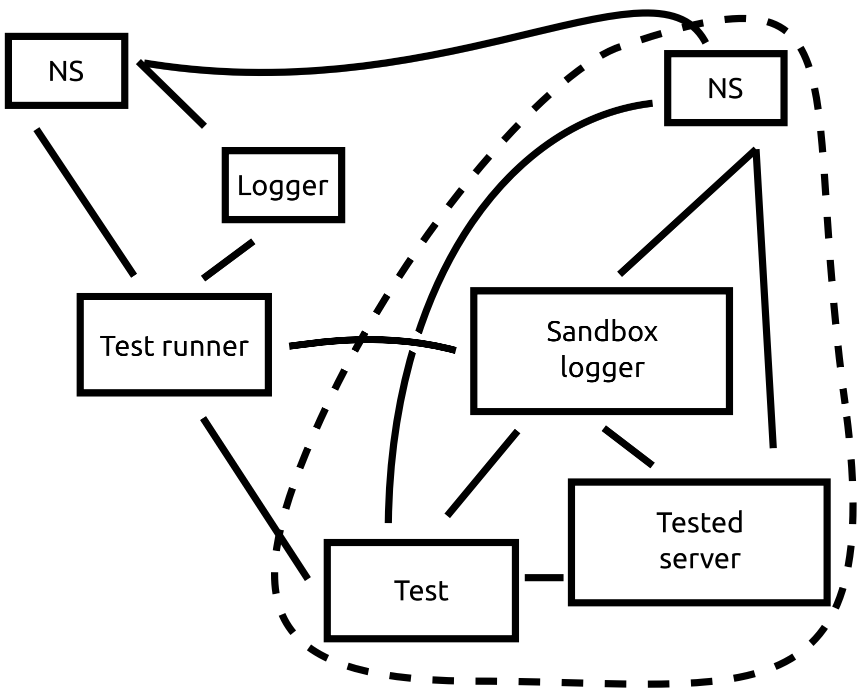
When starting the `test` application, the user can specify one of two modes of operation. When run with `--remote` switch, the tester will connect to the remote control server described in section 5.3.5 and report test results and status information there. If, on the other hand, this flag was not specified, the tester runs using a simple console user interface that presents test results and shows when a test requires external interaction. A human operator can resume the test either with success or failure. Another option is to abort the test run completely.

When tests are configured to be run automatically, the init binary launches both the tester and the remote control server it connects to.

5.3.4 Sandboxing the tests

In order to be able to test various servers providing services, we implemented a model that allows to use a sandbox for the tests. The tester spawns a new instance of naming service and other necessary user-space components such as location service, virtual filesystem server, logger, etc. To allow the userspace tester to spawn a new instance of the naming service and connect new user-space tasks to it, it was necessary to alter the mechanism that is used for spawning new tasks. Before the change, the kernel directed initial phone connections of new task to the naming service, which was the very first task initialized by the kernel. Instead of connecting all tasks to this task, we altered the kernel to connect the inital phone to the task that requested to spawn

Figure 5.3: IPC connections in a simple sandbox setup. The boundary of the sandbox is represented using the dashed line.



a new instance of the loader task. Since the only task that was using this syscall was the naming service, the process of launching new user-space tasks was not dirupted by this change[8].

Currently, the tasks are not isolated in any other way than by IPC connections (or lack thereof). This means that a task may have effect on processes running outside of the sandbox, such as when using special syscalls. Strictly limiting the environment of the tasks requires isolation support in kernel and is out of scope of this thesis. In fact, such a strict isolation would add obstacles, although solvable, even for legitimate cases, for example when a task dump of a crashed task has to be obtained.

Spawning the naming service itself is not sufficient for the sandbox to work correctly, though, as the task that spawned the ns does not have an IPC connection to the naming service available. One possible solution would be that the new naming service would connect to the original naming service[9], but this precludes the use of multiple sandbox instances running in parallel. Fortunately, the process of spawning a new task involves a loader program that loads the binary from the filesystem and then jumps to its entry point. A task that wants to spawn another task has a connection to a unique loader instance that was retrieved from the naming service. This connection is used for all the setup necessary to instruct the loader how to initialize

[8]It even fixes a small design flaw as the kernel is no longer required to hold a special reference for this initial task until the system is shut down

[9]the new naming service has its inital phone connected to the original naming service, just like any other task

the task. By reusing the loader connection, it is possible to communicate with the newly spawned naming service for the sandbox.

The naming service does not use the asynchronous framework for communnication, but relies on low-level IPC calls. However, most other programs use the asynchronous framework, which keeps state for different connections. This means that a loader connection cannot be simply reused when launching such tasks in the sandbox. We solved this by sending a connection request via the loader phone. This creates a new connection that the async framework can handle, because it expects messages of this type.

When a sandbox is created, it does not contain any tasks other than the naming service. This means that when spawning new processes inside the sandbox, a loader program cannot use any of the services it normally expects to exist. This poses a problem as the loader cannot read the binary it is supposed to load from the filesystem. As the program that wants to spawn a new task has a connection to the loader available, we solved this issue by providing a connection to the virtual filesystem server via this connection. The connection to the virtual filesystem server is closed by `loader` just before it loads the new program, so the connection does not leak into the sandbox. Another possible solution to this problem is to shift the responsibility of obtaining the binary from the loader program to the user, so instead of providing a connection to VFS, the user would load the binary itself and copy its contents to the loader. As the current limit for copying data between tasks is about 64kb, the parties would have to use a protocol similar to VFS read. On the other hand, if the user program could provide the binary data itself, spawning a new task would not be limited to programs residing in the filesystem (a test binary may be embedded directly into a tester program for example).

The `loader` also uses services of the logger subsystem. Instead of providing a connection to logger, we chose to simply log messages using kernel syscall if the user-space logging system is not available. This means that if the `logger` service is spawned as the first task in the sandbox, only the first instance of `loader` will log to the kernel [10].

After the logger service is spawned in the sandbox, it does not have a place to log messages coming to it, since it no longer logs messages to kernel log by itself[11] and no filesystem is available inside the sandbox yet. Even if `vfs` was running inside the sandbox, the log messages stored in any file accessible by the logger would get lost as soon as the the sandbox gets discarded. So, after a logger is spawned, the test runner registers itself via an interface we added to the logger server and subscribes to receive all log messages. The messages are then forwarded to the actual system logger, but are namespaced under a log channel beloning to the tester so that messages from the sandbox can be recognized as such.

[10]the one loading the `logger` itself

[11]which is desired behaviour, as this would cause the messages appearing twice or more times depending on the count of logger instances a single message reaches

Another service that is usually spawned in the sandbox is the location server `locsrv`. This server does not currently depend on any other service except the naming server so launching it the same way as other sandbox processes is not a problem. It is still launched after the logger, since a dependency to the logger is likely to be introduced[12].

The tester may be easily extended to spawn additional services to support tests. For example, spawning a virtual filesystem with custom test data is not currently implemented, but requires only a block device server that would accept its data via IPC channel (this is very similar and could be based on the ramdisk server). This would allow the tester to copy a disk image that could be mounted in the virtual filesystem of the sandbox. Spawning services that don't require any data external to the sandbox is a matter of a simple function call.

Sometimes the tests require interaction with services outside of the sandbox. This includes integration tests that could test for example sending network packets or interaction using some other device. While we could opt to run such tests outside of a sandbox, such practice does not allow to enjoy benefits of the sandbox environment. Features such as log redirection or killing all test-spawned processes would not be available. Instead, all tests are run inside a sandbox and the sandbox is more permissive. A test may request that the sandbox naming service forwards requests to a specific service to the system-wide naming service. This functionality can also be disabled by the creator of the sandbox.

In order to destroy a sandbox and kill any of its tasks that may not have exited yet, it was necessary to modify the naming service. First of all, when a request to kill all tasks (`NS_KILLALL`) arrives at the naming service, `ns` disables spawning of new processes. This is necessary to ensure that it enumerates a finite number of tasks to kill. Once the spawning of new tasks is disabled, the naming service proceeds to kill all tasks that have announced their task ids[13].

While this process worked for ordinary applications, we had to modify how the naming server keeps track of tasks to also work for servers. As the kind reader may remember from section 2.2.1, the naming service keeps track of tasks in two ways: a task is connected via IPC so if it finishes, the IPC connection is automatically hanged up and the task is expected to set its return value.

Original `ns` discarded information about a task as soon as it has been successfully waited for. For servers, this happens early after spawning the task, since the servers set return values during initialization. This means that when a request to kill all tasks arrives, the naming server does not have information about the "child" task id anymore. We solved this problem by modifying the naming server to release information about a task only after both its IPC connection has been terminated and the task has been waited for. This ensures that the naming server has the

[12]The task logged its messages to the kernel character output buffer and this method does no longer work, see 5.3.2 for details

[13]Note that although the task that spawned the sandbox is connected to this naming server instance, it has not announced its task id

appropriate list of tasks to kill when requested, assuming no program hangs up its initial connection by itself[14]. The naming server only kills tasks that it spawned so if sandboxes are created recursively, the tasks created by child sandbox cannot be killed by the parent sandbox as it does not have information about the newly-spawned tasks. This limitation should not cause problems, however, as it should not be necessary to spawn sandboxes recursively during testing.

After the request to kill all tasks spawned by the sandbox, the sandbox creator asks the naming service to stop and it terminates. At this point, no task directly spawned by the sandbox should exist.

5.3.5 Remote control server

Monitoring of the state of the system and communication of this information is handled in a separate server, the remote control server (`remotectl`). The functionality of the server is deliberately separated from the test runner, as its usefulness is not limited to running tests. For example, a user may want to display various information while running a virtual machine or interact with the system in another way, using a GUI tool.

The server communicates with the runner outside of the virtual machine using a binary protocol described in 5.4. There are multiple options how to setup a connection to the remote control server. As the connection setup is abstracted away, it should be easy to add support for more connection options as necessary.

At the beginning, we learnt that the ns8250 serial port driver is not fast enough when reading from serial port at high data rates and hadrware buffer overruns occur (the receive buffer in NS16550 serial port is only 16 bytes long). We modified the driver to use a kernel feature that allows to executed limited program to clear interrupts to read the contents of the buffer and pass it to user-space. This made buffer overruns less common.

While implementing communication via a serial port, we stumbled upon a limitation of the same serial port driver. Unfortunately, the driver does not allow simultaneous reading and writing to the serial port. Once a call to `char_dev_read` is made, any call to `char_dev_write` is blocked and waiting until reading finishes. As the read is blocking, it returns only if new data is fed to the serial port. This greatly limits the usefulness of this communication channel. Moreover, this limitation applies also if multiple serial ports are used because device driver framework uses a single driver to handle multiple instances of a device of the same type. Our code is prepared in a way that when this issue is fixed in the driver, the effort required to make the communication work both ways will be minimal.

Another connection method supported by the server is using a tcp connection. This method supports simultaneous reading and writing, but its availability is limited only to platforms, where HelenOS networking stack is supported (some virtualizers

[14]No program in HelenOS currently does so

lack emulation for network devices). Moreover, the networking stack is a larger codebase than simple serial port, so there is more code that might fail and cause a communication error. On the other hand a network stack is more likely to be used by HelenOS developers and new users alike than serial port, so it may be more thoroughly tested.

Upon startup, the remote control server registers itself at the logger server for notifications about all new log messages. Once a log message is received by a callback fibril handling IPC messages from the logger, the log message is encoded according to the wire protocol and the binary message is enqueued for transmission.

The server cannot do the same callback registration for controlling tester, though. This is because the test runner may not be necessarily running when the remote control server starts-up[15]. There are several options how to handle this case. One of the options is that the test runner registers itself in a category at the location service. While this would allow the user to display a list of all services that can be controlled and monitored by the remote control server, there is no common interface that could be used for this purpose currently. At present, there are only a few services it is worth monitoring so we chose a simpler approach for the initial implementation. The remote control server acts as a singleton service registered via `ns` and the tester estabilishes a direct connection there if it was run with special command-line arguments. The test runner itself may be launched by the user themselves or directly by the remote control server, the latter will require only minor modifications to the code.

Monitoring capabilities of the remote control server may be extended in the future. Good candidates for reporting additional information include task dumps in structured form so that a user may be directed to the source of the crash. Task dumps are automatically executed when a task crashes, so the application that gather information about task state, `taskdump`, may be modified to send this information also via the remote control server.

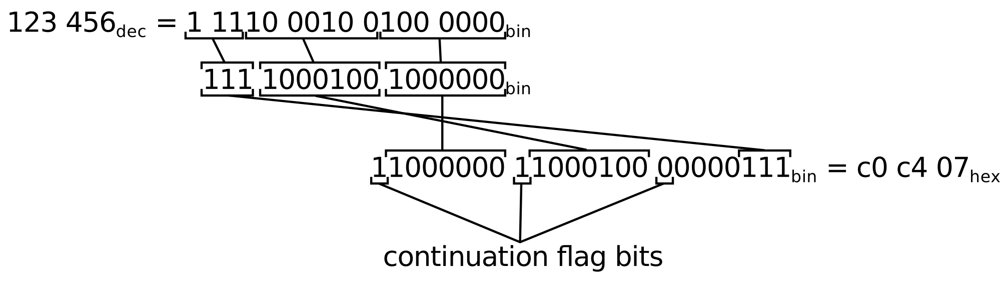
Another possibly interesting information to the remote party may be gathered by periodically snapshotting monitoring information about tasks and other system components that could be obtained by HelenOS monitoring interface[19]. This may include current processor load, memory statistics, etc.

5.4 Remote control protocol

The protocol between user-space remote control server and external runner program is based upon Protocol Buffers format[20]. We chose this format because it is relatively simple, allows for further protocol extensions, supports binary data and there is a lot of libraries available for various programming languages that allow to work with this format[21].

[15]remeber that the `logger` is one of the init binaries, so it is running as soon as basic user-space environment is initialized

Figure 5.4: Encoding a number to varint format



123 456dec = 1 1110 0010 0100 0000bin
111 1000100 1000000bin
11000000 11000100 00000111bin = c0 c4 07hex
continuation flag bits

Protocol Buffers encode a message as a list of key-value pairs (or fields). The keys are unsigned positive integers and must remain immutable for the entire lifetime of the concrete protocol implementation to maintain backwards compatibilty. Each message is tagged with the wire format its payload is stored in. This allows parsers to identify the length of the payload. However, the specific format of the payload is not embedded into the message and the parser must have prior knowledge of data type for any fields it parses. This definition allows unknown messages to be safely ignored.

Our implementation supports only a subset of the wire formats of the protocol, as described in table 5.2

Table 5.2: Subset of Protocol Buffers wire formats we implemented

Wire format number	Description
0	Variable-length integer
1	64-bit fixed-length integer
2	Length-delimited payload
5	32-bit fixed-length integer

Of these four wire formats, we actually only use the types 0 and 2 in the remote control protocol.

A variable length integer is encoded as a series of bytes, where each byte has the most significant bit set if there are more bytes following which need to be consumed. Each byte has therefore 7 bits available for the payload. The integer is divided into 7-bit groups and those groups are stored least-significant group first. A small integer therefore uses a small number of bytes of storage, with the sizes ranging from a single byte for numbers less than 128 to a maximum of 10 bytes when a full range of 64-bit integers has to be stored. The encoding process is illustrated in figure 5.4.

A length-delimited payload is a concatenation of payload length, encoded as a variable-length integer and the payload itself. This allows the payload to be composed of arbitrary binary data.

Table 5.3: Remote control protocol top-level message types

ID	Description
1	Hello message
2	Log event message
3	Test status change message
4	Test run status change message

We use only a few data types in the remote control protocol. For numbers, we use unsigned integers, which are stored as variable integers without any transformation before encoding[16]. A string is stored in UTF-8 encoding as length delimited payload.

Each field is stored as a variable integer header, composed of the field ID shifted to the left by 3 bits and 3-bit specifier of the wire format, followed by field data as mandated by the wire format.

Being stored as a list of key-value pairs, messages may be merged by simple concatenation (if the two messages contain the same key, the latter is used). This is a desirable property in some cases, but we need to send and receive individual messages. If we sent the messages through a channel that does not delimit individual messages, such as serial port, we would not be able to distinguish, which key/value pairs belong to which message. Therefore, we wrap each message in a simple envelope that contains its length, message type and the payload (encoded in the same way as a length-delimited field).

5.4.1 Hello message

The hello message is used to introduce the remote control listener implementation and currently only contains information about whether the remote control client is connected to the user-space or kernel implementation.

Having a message of this type allows the client to know when it can issue commands to the remote party, for example to initiate a test run.

Using the protocol buffer language[22], the contents of this message may be defined as follows:

```
message HelloMessage {
  enum ImplType {
    UNKNOWN = 0;
    USPACE_REMOTECTL = 1;
    KERNEL = 2;
  }

```

[16]Protocol buffers support zig-zag encoding which is more efficient for storing negative integer numbers. The specification also instructs how to encode floating point numbers

```
  optional ImplType type = 1 [default = UNKNOWN];
}
```

5.4.2 Log event message

The log event message is emitted by the remote control server whenever a new message appears in the system log. It currently contains information about the source, severity and the log message. A timestamp may be added in the future.

```
message LogEventMessage {
  enum LogLevel {
    /* Fatal error, program is not able to recover. */
    FATAL = 0;

    /* Serious error, program can still recover. */
    ERROR = 1;

    /* Easily recoverable problem. */
    WARN = 2;

    /* Information message (printed by default). */
    NOTE = 3;

    /* Debugging purpose message. */
    DEBUG = 4;

    /* More detailed debugging message. */
    DEBUG2 = 5;
  }
  optional LogLevel level = 1 [default = NOTE];
  optional string log_name = 2;
  required string message_text = 3;
}
```

5.4.3 Test ID submessage

This message is embedded in other messages and represents a test identifier.

```
message TestIDMessage {
  required string name = 1;
  optional string provider = 2;
}
```

5.4.4 Test status change message

This message is sent when a test finishes.

```
message TestStatusChangedMessage {
  enum TestStatus {
    /* The test has not been executed */
    NOT_RUN = 0;

    PASSED = 1;
    FAILED = 2;
    SKIPPED = 3;
    ABORTED = 4;
  }
  optional TestStatus status = 1 [default = NOT_RUN];
  required TestIDMessage test_id = 2;
  optional string result_text = 3;
}
```

5.4.5 Test run status change message

This message is sent whenever the test run changes state, such as when a new test is launched, the test run completes, etc.

```
message TestStatusChangedMessage {
  enum TestRunStatus {
    /* The test run has not been executed */
    NONE = 0;

    RUNNING = 2;
    WAITING = 5;
    ABORTED = 9;
    FINISHED = 10;
  }
  optional TestRunStatus status = 1 [default = NONE];
  required uint32 total_tests = 2;
  required uint32 done_tests = 3;
  optional TestIDMessage current_test_id = 4;
  optional string waitpoint = 5;
}
```

Chapter 6

Comparison with other solutions

In this chapter we compare our framework with other solutions with respect to several properties.

6.1 Test life-cycle management

In our framework, the main component that manages the life-cycle of tests is the user-space test runner. It prepares a list of tests, executes them, and reports the intermediate state and final results to the component running outside of the virtual machine. The same program can be run directly from HelenOS, even when HelenOS is running on real hardware.

NetBSD's Anita[13] and Automated Testing Framework (ATF) tools used in conjuction also allow to run tests in a virtual machine or on a real hardware[14]. Anita is a tool to (optionally) download and install NetBSD in a virtual machine. Once the system is installed, Anita executes ATF, which runs full test suite and stores the results on the hard-drive. Anita then extracts test results from the virtual hard-drive.

On the other hand, OS-autoinst[11], a tool for running automated tests of operating systems using QEMU or VirtualBox, executes and determines test results outside of the virtual machine. OS-autoinst tests wait until an expected event occurs (such as an image appearing on the screen) and automate the virtual machine (e.g. by sending keystrokes using virtual keyboard).

Genode[15] has a tool named `autopilot` that works in a similar way. This tool uses QEMU and uses similar ”wait until an expected event occurs” approach.

6.2 Test isolation

OS-autoinst and Genode's autopilot execute tests in separate virtual machine instances. This provides good isolation of individual tests, but incurs overhead of booting the system every time. This is not a problem when hardware assisted virtualization is used, as it is relatively fast, but may amount for a significant portion of time when not used.

Like Anita, our framework executes tests in a single instance of a virtual machine. Running the whole test suite in a single instance is more appropriate for HelenOS where the tests will be run in a virtual machine with different architecture than the host is running, which means that hardware-assisted virtualization cannot be used to speed-up the execution.

Each test executed using ATF is a separate process (that may launch other processes as well). ATF executes test in an empty temporary directory that is cleaned by the framework after use.

Since NetBSD uses a monolithic kernel, they have a special framework to run portions of the kernel in user-space during testing. HelenOS is a microkernel based OS so the majority of the code resides in user-space and such framework is not necessary.

MINIX 3 has tests in different formats and places, mainly because it uses a lot of other POSIX-compatible programs, which come with their own tests. Some MINIX 3 specific tests restart a service they are testing before and after test execution, otherwise the tests are isolated by being run in a separate process.

In HelenOS, we decided to take the isolation of user-space tests even further – each test is executed inside a sandbox that contains a fresh environment composed of services necessary to execute the test (although tests may be written so that they can access services outside of the sandbox, if necessary).

6.3 Information gathered during tests

Most frameworks store diagnostic messages or other information that is output during tests. OS-autoinst runs outside of the virtual machine, so it mostly logs information about its actions and events (when it detected something on screen, serial output, or when it sent key presses etc.). It also saves images of the screen and creates a video of the test run.

NetBSD ATF stores output of tests (standard output and error streams), test results, and also includes stack traces, if a test happens to crash.

Our framework stores log messages from kernel and user-space logs as well as test log messages (these go to special log channel) interleaved so that events happening at the same time are close in the log. Events such as when test is started, stopped or is waiting are also stored.

Conclusion

In this thesis we created a test framework for running automated tests tailored to HelenOS needs. This framework should replace the basic code that is already present in HelenOS so that more sophisticated tests can be executed and evaluated. This framework allows to add new user-space tests easily as well as extend the framework itself at different levels.

We also described HelenOS subsystems related to the design and implementation of the framework as well as the original code to execute tests that our framework replaces.

Since the topic of creating an automated testing framework for an operating system such as HelenOS is broad, we included an analysis of what should be the focus of our work.

We also elaborated on the design decisions and implementation details of the framework and compared our implementation to other solutions.

Achievements

We achieved most of the goals set out in the section 4.4.

The framework we implemented allows to execute a QEMU virtual machine running an instance of HelenOS. The parameters to launch the emulator are automatically determined based on current HelenOS build configuration. When configured to execute tests, the system runs all kernel and user-space tests it discovers.

Tests may also be executed directly from command-line within HelenOS, in which case the test batch is controlled by the user via console, which also shows test results. Similar capabilities are available in the kernel if it was built with support for the debugging console.

Log messages from the system as well as individual executed tests are transmitted outside of the virtual machine live as they are emitted, where they are processed and stored in a log file.

Support for system tests that require external action to happen was added. The tests may wait at a named point to be either resumed with success or an error condition. Unfortunately, due to unexpected limitiations of some drivers (and a lack of time to fix the limitations), it is currently not possible to respond to those events from the test runner residing outside of the virtual machine. The support may be nevertheless excersized using semi-automatic mode where the tests wait for a human operator to complete the action and resume the tests.

Userspace tests are isolated in a sandboxed environment that allows to safely test basic services of HelenOS userspace. Without this sandbox, testing services such as the naming service or virtual filesystem might cause disruptions to the test framework or other HelenOS subsystems.

It is possible to use use the test runner in GDB mode that allows to read log messages from the kernel using the debugger, at least with ia32 HelenOS builds. Extracting results of tests is not implemented, again due to time reasons, but the kernel test runner stores its data structures so that adding this support should be straightforward when using the facilities already used to extract log messages. Support for GDB in QEMU does not work as advertised on other architectures we tried. When this is issue is fixed, only small changes will be required in our test runner.

Contributions

During developement of the framework we have introduced support for logging structured messages in the kernel to match existing user-space implementation. The two logs may now be transparently synchronized.

Improvements in various parts of code were made along the way, some directly related to our work, some not.

We have also discovered several bugs in various HelenOS components and fixed them. Most of those fixes were also contributed directly to HelenOS mainline.

Future possibilities

Once the framework is merged into HelenOS mainline, the obvious way to proceed is to actually start using the framework and add test cases beyond the few ones for that were included for demonstration. Old tests should be ported to the framework and new tests leveraging all of the features of the framework should be written.

After more tests are available, it makes sense to run the test suite periodically as part of continuous integration process. This shall provide early notifications of regressions, which may manifest on architectures other than that the developer is using.

When the necessary drivers are added, the framework should be able to communicate on more platforms. To extend the reach to platforms not supported by QEMU, support for other simulators can be added.

The remote control server could be extended to send more information from the system and manipulate various subsytems. This can be used in a tool that could provide support and convenience for developers. For example, when a task crashes, the tool could allow the developer to directly open the reported source code line in a text editor of choice on the host system.

If an interface definition language is introduced into HelenOS, the framework could leverage the information provided by such definitions to automatically generate test cases that inject faults into IPC communication of the various HelenOS components.

Last but not least adding support for computing code coverage could be explored. This should help identify which code paths are not covered by tests and generate further test cases to excersize them.

Bibliography

[1] *HelenOS project homepage* [online]. n.d. Updated 2013-04-09 [cit. 2013-04-13] Available online: `http://www.helenos.org`

[2] BELLARD, Fabrice, et al. *QEMU* [computer program]. Available online: `http://www.qemu.org/`

[3] *QEMU Emulator User Documentation* [online]. n.d. [2013-05-01] Available online: `http://qemu.weilnetz.de/qemu-doc.html`

[4] *QEMU Monitor Protocol* [online]. 2010. Updated 2012-05-21 [cit. 2013-04-30]. Available online: `http://wiki.qemu.org/QMP`

[5] *Debugging with GDB* [online]. Free Software Foundation, Inc., ©2013 [cit. 2013-04-10]. Tenth Edition. The GDB/MI Interface. Available online: `http://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html#GDB_002fMI`

[6] *Oracle VM VirtualBox* [computer program]. Oracle Corporation. Available online: `http://www.virtualbox.org`

[7] *Oracle VM VirtualBox User Manual* [online]. Oracle Corporation, ©2013 [cit. 2013-05-01]. Available online: `http://www.virtualbox.org/manual/`

[8] *VProbes Programming Reference* [online]. VMWare, Inc., 2011 [cit. 2013-03-20]. Available online: `http://www.vmware.com/pdf/ws8_f4_vprobes_reference.pdf`

[9] HOLUB, Viliam and Martin DĚCKÝ and Tomáš MARTINEC. *MSIM* [computer program]. Available online: `http://d3s.mff.cuni.cz/~holub/sw/msim/`

[10] HOLUB, Viliam and Martin DĚCKÝ. *MSIM Version 1.3.8.3 Reference Manual* [online]. 2010 [cit. 2013-05-01]. Available online: `http://d3s.mff.cuni.cz/~holub/sw/msim/reference.html`

[11] WIEDERMANN, Bernhard M. and Dominik HEIDLER. *OS-autoinst* [computer program]. Available online: `http://www.os-autoinst.org/`

[12] FOWLER, Martin. *Continuous Integration* [online]. 2006 [cit. 2013-04-30]. Available online: `http://martinfowler.com/articles/continuousIntegration.html`

[13] GUSTAFFSON, Andreas. *Anita* [computer program]. Available online: `http://www.gson.org/netbsd/anita/`

[14] HUSEMANN, Martin. Testing NetBSD Automagically. *In: 10th European BSD Conference* [online]. 2011. Available online: `http://2011.eurobsdcon.org/papers/husemann/Testing_NetBSD_automagically.pdf`

[15] *Genode Operating System Framework.* [computer program] Available online: `http://genode.org`

[16] HERDER, Jorrit N. e al. MINIX 3: a highly reliable, self-repairing operating system. *In: ACM SIGOPS Operating Systems Review.* 2006, Volume 40, Issue 3, Pages 80-89

[17] TROCHTOVÁ, Lenka. *Rozhraní pro ovladače zařízení v HelenOS.* 2010. Also available online: `http://www.helenos.org/doc/theses/lt-thesis.pdf`

[18] *IPC for Dummies* [online]. 2009 [cit. 2013-05-03]. Available online: `http://trac.helenos.org/wiki/IPC`

[19] KOZINA, Stanistav. *HelenOS Monitoring.* 2010. Also available online: `http://www.helenos.org/doc/theses/sk-thesis.pdf`

[20] *Encoding - Protocol Buffers* [online]. n.d. Updated 2012-04-02 [cit. 2013-04-13]. Available online: `https://developers.google.com/protocol-buffers/docs/encoding`

[21] *Third-Party Add-ons for Protocol Buffers* [online] n.d. Updated 2013-03-19 [cit. 2013-04-13]. Available online: `http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns`

[22] *Language Guide - Protocol Buffers* [online]. n.d. Updated 2013-03-05 [cit. 2013-04-14]. Available online: `https://developers.google.com/protocol-buffers/docs/proto`

[23] *Port HelenOS to an existing 32-bit MIPS board / computer.* [online] 2012. Updated 2012-12-06 [cit. 2013-05-01]. Available online: `http://trac.helenos.org/ticket/417`

[24] HOUSE, D. E. and NEWMAN, W. F.. Testing Large Software Products. *In: ACM SIGSOFT Software Engineering Notes.* 1989, Volume 14, Issue 2, Pages 71-77.

[25] MARINESCU, PAUL D. and CANDEA, George. Efficient Testing of Recovery Code Using Fault Injection. *In: ACM Transactions on Computer Systems.* 2011, Volume 29, Issue 4, Article No. 11.

[26] MARTIGNONI, Lorenzo et al. Testing system virtual machines. *In: Proceedings of the 19th international symposium on Software testing and analysis*. 2010, Pages 171-182.

[27] YANG, Qian and LI, J. Jenny and WEISS, David. A survey of coverage based testing tools. *In Proceedings of the 2006 international workshop on Automation of software test*. 2006, Pages 99-103.

[28] BORDIN, Matteo et al. Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework. *[In ERTS (Embedded Real Time Software and Systems conference)]*. 2010. Also available online: `http://www.adacore.com/uploads/technical-papers/couverture_ertss2010.pdf`

Appendix A

Source code

This appendix contains instructions how to compile and run the source code of modified HelenOS containing our changes. The source code is stored inside `source.tar.gz` file in the root of the CD-ROM attached to this thesis. The source code is stored in a bazaar repository and may also be obtained by branching our repository:

```
bzr branch http://ho.st.dcs.fmph.uniba.sk/~mato/bzr/helenos-testing
```

Structure of the source code

HelenOS source distribution is divided to subprojects. Main projects that produce output that goes into a bootable image are abi, boot, kernel and uspace.

Table A.1: Top level directories of HelenOS source code distribution

Directory	Description
abi	Headers defining interface between kernel and userspace
boot	Boot loaders
contrib	Various contributed files
defaults	HelenOS configuration profiles
kernel	HelenOS kernel (SPARTAN)
tools	Tools to be executed in the developer's environment
uspace	User-space programs, configuration, etc.

The files that contain our changes to HelenOS are in different places all over the source hierarchy, but the main parts of the framework can be found in:

- Part of the framework that runs outside of a VM
  - tools/run.py
  - tools/runner/
- Kernel test runner

  - kernel/generic/src/tester/
  - kernel/include/test/

- User-space test runner and test library
  - uspace/app/test/
  - uspace/lib/test/

- Sample tests
  - uspace/test/

- Remote control server
  - uspace/srv/remotectl

- Logging subsystems related changes
  - kernel/generic/src/log/
  - kernel/generic/include/log.h
  - uspace/srv/logger/sink.c
  - uspace/srv/klog/

Compiling and running

This section briefly explains how the system can be built and executed. For further information about building HelenOS, please refer to the HelenOS User's Guide:

`http://trac.helenos.org/wiki/UsersGuide/CompilingFromSource`

In order to build HelenOS from source, it is necessary to install a supported compiler toolchain in a GNU/Linux system. A script to automatically download and install the necessary tools is a standard part of HelenOS source code distribution. The script is located in `tools/toolchain.sh`.

For the following text, let's assume that a shell variable `HELENOS_ROOT` is defined and points to the root directory of the source code distribution.

To build a toolchain for the `ia32` architecture, execute from a temporary directory:

```
$HELENOS_ROOT/tools/toolchain.sh ia32
```

With the toolchain installed, proceed to build the system:

```
cd $HELENOS_ROOT
make
```

A configuration screen should appear allowing to set various configuration parameters. Please select *Load preconfigured defaults* and then *ia32*.

The next step is to configure some options:

- *Support for NS16550 controller (kernel console)* should stay disabled
- Enable *Line debugging information*
- Enable *Run test suite at startup*

After pressing *Done*, the build process starts. When the build finishes, you may then proceed to run `./tools/run.py` to launch a virtual machine. Tests will be executed, their results collected, and the virtual machine stopped. When the process finishes, an HTML report should be generated in `test-results.html`.

The `run.py` script expects QEMU to be in system path. Optionally, you may set `QEMU_DIR` environment variable to the location of the QEMU's `bin` directory and the script will run QEMU from there.

With `--gdb` option, the runner uses GDB to extract kernel logs from the running virtual machine. Note that a recent version of QEMU (at least version 1.2) is required for this feature to work correctly.

In case you want to try the user-space runner, you can build the system without running the tests automatically (which closes the VM when done):

```
make distclean
make
```

Just don't select *Run test suite at startup* during configuration.

When a terminal window appears in HelenOS, the you can list all available test by typing

```
test
```

and execute all tests by running

```
test *
```

or just user-space tests:

```
test directory:*
```