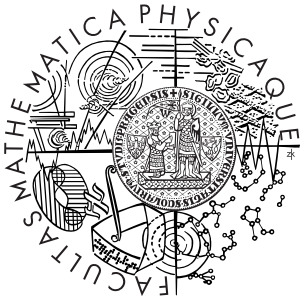
Charles University in Prague
Faculty of Mathematics and Physics

# MASTER THESIS




Stanislav Kozina

## HelenOS monitoring

Department of Distributed and Dependable systems

Supervisor: Mgr. Martin Děcký
Study program: Computer Science, Software systems

2010

I would like to thank Mgr. Martin Děcký, the supervisor of the thesis, for many hours of consultation and for patient clearing up many HelenOS features to me. He also suggested the main topic of the thesis and lead me through whole process of design and creation.

Next big thank goes to Doc. Ing. Petr Tůma Dr. who introduced our group of students to the operating system design fundamentals. Thank to his courses I get interested in operating systems at all.

Last but not least I would like to thank Jakub Jermář as the original creator of the SPARTAN kernel upon HelenOS is built.

Thank you all.

I hereby declare that I have written this thesis myself, on my own and solely using the cited sources. I give permission to loan this document.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 1. 8. 2010

. . . . . .
Stanislav Kozina

# Contents

# List of Figures



# List of Tables

# List of Abbreviations

| | |
|---|---|
| ACK | Amsterdam Compiler Kit |
| APIC | Advanced Programmable Interrupt Controller |
| CD | Compact Disc |
| CPU | Central Processing Unit |
| ELF | Executable and Linkable Format |
| GNU | GNU's Not Unix |
| IA-32 | Intel Architecture, 32-bit |
| IO | Input / Output |
| IP | Instruction Pointer |
| IPC | Inter-Process Communication |
| IPI | Inter-Processor Interrupt |
| IRQ | Interrupt Request |
| MSR | Model Specific Register |
| NMI | Non-Maskable Interrupt |
| NS | Naming Service |
| PC | Personal Computer |
| PIC | Programmable Interrupt Controller |
| PMC | Performance Monitoring Counters |
| RTC | Real Time Clock |
| SMP | Symmetric Multi Processing |

**Title**: HelenOS Monitoring
**Author**: Stanislav Kozina
**Department**: Department of Distributed and Dependable Systems, MFF UK
**Supervisor**: Mgr. Martin Děcký
**Supervisor's e-mail address**: martin.decky@mff.cuni.cz

**Abstract**: The main purpose of an operating system is to supply the user programs with persistent, simple and enough powerful service interface. However for practical use of the system it is often necessary to know the current state of the system, how much resources the system is using and which user programs consume the biggest portion of them.

In the thesis we discuss the possibilities of the operating system monitoring in common. For HelenOS system (which was missing any monitoring facility) such interface was created.

Next to the monitoring services we designed and implemented a simple statistical profiler. Both the user programs and the whole operating system can be profiled. Two userspace applications were created for this purpose.

**Keywords**: HelenOS, monitoring, profiler

**Název práce**: HelenOS Monitoring
**Autor**: Stanislav Kozina
**Katedra (ústav)**: Katedra Softwarového inženýrství, MFF UK
**Vedoucí bakalářské práce**: Mgr. Martin Děcký
**E-mail vedoucího**: martin.decky@mff.cuni.cz

**Abstrakt**: Hlavním úkolem operačního systému je poskytnout uživatelským úlohám stabilní, jednoduché a přitom dostatečně silné rozhraní služeb. Nicméně pro praktické použití systému je často nutné zjistit v jakém stavu se systém opravdu nachází, kolik jeho prostředků je využito a které uživatelské programy spotřebovávají největší část těchto zdrojů.

Práce se zabývá možnostmi monitorování stavu operačního systému obecně. Pro systém HelenOS, kde doposud monitorovací rozhraní zcela chybělo, bylo takové rozhraní nově vytvořeno.

Vedle monitorovacích služeb byl navržen a implementován také jednoduchý statistický profiler. Profilovat lze uživatelské programy stejně jako celý operační systém. Pro tento účel byly vytvořeny dvě uživatelské aplikace.

**Klíčová slova**: HelenOS, monitoring, profiler

# Chapter 1

# Introduction

Through the past few decades computers made a giant step forward. Half century ago computer was just a one-purpose machine which was able to compute the algorithm for which it was constructed. But as the time moved forward they evolved into contemporary machines on which a program can be created and run without even restarting the machine. Also several programs can run in parallel not considering each other. Thousands of machines are interconnected and many programs can exploit the power of all these computers for one purpose.

Modern computer offers plenty of services which are used by user programs to fulfill tasks they were designed for. Thanks to several layers of abstraction the user programs don't need to care a lot about a low level details. To a certain extent they even don't need to care about the machine architecture they are running on.

The most simple (from the computers point of view) user programs use the machine only to perform different math and memory operations. But typical tasks also employ services offered by the operating system such as I/O operations, Inter-Process Communication (*IPC*), task handling, synchronization and others. Simple operating systems offer about tens of such services via the system call interface, widely spread systems has more than few hundreds of these calls.

The operating system must also response to many unexpected situations. These are many different errors, inputs on interfaces from the outer world and even internal device notifications. These events are typically delivered to the operating system in the form of exceptions or interrupts, the system must process all of them. Typical reaction is to correct the machine state

in order to fix the exceptional situation. Non-existing memory mapping is refilled, received data are stored in the memory and the failing user task can even be killed if the error is serious enough. Although these situations are called 'exceptions' they usually happen many hundred times per second.

It is very hard to trace all the operations performed by today's computers. Typical machine can process few billions of instructions per second. It is almost impossible to watch them all. Just for saving every processed instruction we would need at least few other instructions. The space for storing the results would be enormous.

Usually only remarkable operations of the user tasks (and operating system itself) are recorded to gain a general overview about what's going on in the system. All real systems contain similar facility, however, at the beginning of the work on this thesis HelenOS system was lacking any such ability.

## 1.1 Goals

The primary goal of this thesis is to design and implement the system monitoring facility in the HelenOS system. Its output should be exposed through some kind of interface to the user tasks. The solution must be able to monitor all basic system and task operations. To simplify viewing the output also some userspace tools should be made. Next to this work we discuss also options and solutions of monitoring in other operating system.

It is not desirable to port any existing solution from other system to HelenOS as its design differs a lot. Many basic system structures are very different from any other one. The basic HelenOS motto ('To do the things The Right Way') should be bared in mind all the time.

Next to this primary goal we consider possibilities of a task and system profiling. An aim is to design and implement a brand new profiling apparatus in the system kernel. Also some userspace tools should be created for setting up the profiling facility and for listing its results.

## 1.2 Organization of the Thesis

The whole thesis is divided into several sections. There are two weakly coupled parts – one about monitoring features and another about the statistical

profiling. They are both described in separate sections, each is also preceded by one more common introduction chapter.

**Chapter 2** contains introduction to the operating system fundamentals and their solutions in HelenOS system in particular. It should be read by all of those who are not familiar with the HelenOS system.

**Chapter 3** describes the system monitoring in common. Here we define the goal and the range of the monitoring. Several existing tools are discussed and the basic technical possibilities are inspected.

**Chapter 4** shows a detailed description on how was the HelenOS monitoring subsystem implemented. We present the ideas, their benefits and drawbacks and the final solution which was implemented.

**Chapter 5** shows the possibilities of profiling in common. We define the profiling scope and show how it can be accomplished, where the samples can be collected and what are the overall possibilities of implementation. At the end we present several existing profilers.

**Chapter 6** describes the implementation of the statistical profiler on the HelenOS system. Two separate profilers were implemented, one for profiling a selected user task and one for profiling all code running in the machine (including kernel and interrupt handlers).

**Chapter 7** concludes whole thesis and evaluates the asset of the work. Also some future possibilities are described there.

**Appendix A** introduces several userspace tools which were made for viewing all results.

**Appendix B** lists some example monitoring and profiling results.

## 1.3 Content of the CD

The attached CD can be directly used for booting HelenOS system with both the monitoring and profiling features enabled. The system was configured and built for IA-32 system architecture. Simply insert the CD into the drive and set your BIOS to boot from it.

The directories stored in the root directory on the CD have following content:

- **boot** contains HelenOS boot files,
- **src** contains complete system source code,
- **thesis** contains this thesis in PDF format.

## 1.4 Compilation and Running

HelenOS system source code is managed by Bazaar distributed version control system [1]. All currently developed branches are accessible to public on the Launchpad server. For this thesis a new branch was created:

```
lp:~ersin/helenos/measure2
```

For obtaining the source code of the work done for this thesis first install the Bazaar version control system (either from your distribution repository or from project web page) and make a branch of the measure2 branch:

```
bzr branch lp:~ersin/helenos/measure2 ./local-branch-name
```

For getting the cross-compiler required to build the HelenOS sources use the tools stored in the 'tools' subdirectory in the repository. For later building use the command `make`. The build system lets you to configure the system and after the compilation it produces a bootable CD image with the system binary.

The image can be burned on a CD and used for booting on a real machine, or can be inserted to the virtual CD drive of a machine emulator. Unfortunately most of the virtual machine emulators do not emulate all low level hardware features[1]. Booting HelenOS on a real hardware is therefore recommended instead. As some of the features presented in this work are very architecture dependant they were implemented only for a IA-32[2] architecture. Some features are even CPU model specific so all of them are available only on an IA-32 architecture CPU made by Intel vendor newer than Intel Pentium chip. As HelenOS does not enable local APIC controller in single processor configuration it is also necessary to boot it on some multi processor system.

[1] For example a popular virtual machine emulator QEMU does not emulate Performance Monitoring Counter registers which are used in this thesis. Also its implementation of the Real Time Clock chip is not able to generate interrupts at higher frequencies.

[2] Also known as x86 or i386

## 1.5 Directory Structure of the Source Code

The main part of code related to monitoring features is stored in these files:

```
kernel/generic/include/sysinfo/abi.h
kernel/generic/include/sysinfo/stats.h
kernel/generic/src/sysinfo/stats.c
uspace/lib/c/generic/stats.c
uspace/lib/c/include/stats.h
uspace/app/top/*
uspace/app/stats/*
uspace/app/tasks/*
```

The monitoring facility probes were inserted into these files:

```
kernel/generic/src/proc/scheduler.c
kernel/generic/src/proc/thread.c
kernel/generic/src/proc/task.c
kernel/generic/src/interrupt/interrupt.c
kernel/generic/src/syscall/syscall.c
kernel/generic/src/time/clock.c
kernel/generic/src/sync/waitq.c
```

Code related to statistical profiling can be found mainly in these files:

```
kernel/generic/include/profile/sprofile.h
kernel/generic/include/profile/sprofile_system.h
kernel/generic/include/profile/abi.h
kernel/generic/src/profile/sprofile.c
kernel/arch/ia32/include/drivers/rtc.h
kernel/arch/ia32/include/drivers/perf_count.h
kernel/arch/ia32/src/drivers/rtc.c
kernel/arch/ia32/src/drivers/perf_count.c
uspace/lib/c/generic/sprofile.c
uspace/lib/c/include/sprofile.h
uspace/app/profile/*
uspace/app/sprofile/*
```

## 1.6 Conventions

Several conventions are used through the document.

New terms and important names or abbreviations are emphasized by an *italic* font. These words can sometimes also be searched for in the C source code or on the Internet.

Fragments of the code itself are written in a `non-proporcial` font. This is used for both the function names, system call names, message methods and constants.

# Chapter 2

# HelenOS Overview

HelenOS is a relatively young research operating system. Its development started in 2001–2004 when Jakub Jermář wrote from scratch the SPARTAN kernel as a school assignment. It was later expanded (also as a school project) to the HelenOS operating system by the group of authors who mostly develop the system till these days.

The main purpose of the HelenOS system is to develop an operating system in a new and reasonable way. Many original structures and subsystems were created in order to see their behavior in real. Compatibility with existing standards is not emphasized a lot.

## 2.1 Architecture

The main architecture of the HelenOS system is based on micro-kernel paradigm. There is a small (but not minimal) kernel which provides the very basic system calls. A lot of common system services are implemented as several user space programs called *servers*.

Among the most important servers we find these ones:

- The **Naming Service**, which serves as a central point for other tasks for establishing the connections between each other,
- The **Device Mapper** which forwards user messages sent to a device to corresponding device driver,
- Several **File System** servers, which serve requests for files and directories by reading blocks from disks or memory in particular format,

- The **Virtual File System**, which serves requests for files by handing the request over to corresponding file system server,
- And many others.

The kernel was developed with portability in mind since the very first days. Ports to almost all common architectures were already made although some of them were not tested on the real hardware yet.

One of the design goals was to create a modern operating system. HelenOS is fully preemptive multitasking system, supports multiple CPUs and virtual memory. The communication between the tasks (mainly between the user programs and system servers) is done via original IPC implementation based on asynchronous message sending interface. For more information about HelenOS design see HelenOS web page [7].

## 2.2 Threads and Tasks

The basic unit of execution in HelenOS kernel is a *thread*. Thread is the memory structure which represents one flow of execution which can run on one CPU. The CPU is assigned to the thread for short periods of time periodically by the scheduler.

Apart from the CPU context (as the most significant part of the thread structure) thread also contains a name, priority, state information and more internal system items. For every thread there is also a separated kernel stack which is used by the kernel code whenever the thread is interrupted.

Every thread must be a member of one *task* structure which serves as a container for them. Usually every user program is represented by one task but this is not the rule. Few threads can be members of one task which represents all their common attributes. Here we find memory mapping, IPC communication endpoints, security context etc. We see that communication is done between the tasks, not between the threads. When the very last thread of the task terminates, its task structure is also destroyed.

There are few special purpose threads running only in the kernel space. A *kernel* task is created to contain all of them. This task exists all the time the system is running.

## 2.3 Fibrils

Beside the common threads which are managed by the kernel scheduler subsystem (as in any other fully preemptive operating system) HelenOS introduces also user space pseudo-threads called *Fibrils*. This idea is not new at all, for example Microsoft Windows Fibers [17] represents almost the same feature. What makes HelenOS different compared to other systems (where user space threads are not widely used) is the importance of fibrils for even basic operations. All userspace tasks perform the majority of their operations through the *Asynchronous library* which is based on fibrils and exploits their benefits very remarkable.

## 2.4 IPC Communication

IPC communication subsystem is one of the most interesting ones in the whole HelenOS kernel. The communication is based on an analogy of telephone calls made to the telephone answer boxes.

Any thread of the task can connect one of task's *phones* (sender's endpoint) to the *answerbox* of a called task (receiver's endpoint). After the connection is established it can send *calls* (small 'messages') over the line and wait for the answers later. A thread can send several calls before waiting for the answer because the communication is asynchronous. There is a small buffer for sent and received calls. Every call (except kernel IRQ notification calls) must be answered.

Every call contains six fixed size arguments of payload which can be used for passing data. First argument of the message defines the message method. Lower specific values are processed by the system. Higher values can be used by user application freely.

The IPC subsystem is accessed using standard syscall interface. Two specific versions of message passing calls exist. The fast one passes all arguments through the CPU registers, however only a limited number of values can be passed. Slow one stores all arguments on the stack.

Table 2.1 shows an overview of basic system message methods.

### 2.4.1 Call Forwarding

Some calls can even be forwarded from one task to another if the original destination task decides to do so. In this case the message is transmitted to

| Message type | Message description |
|---|---|
| IPC_M_CONNECTION_CLONE | Send a copy of a connection. |
| IPC_M_CONNECT_ME | Notify a receiver about a cloned connection. |
| IPC_M_CONNECT_TO_ME | Register new system service. |
| IPC_M_CONNECT_ME_TO | Make a connection to requested service. |
| IPC_M_PHONE_HANGUP | Terminate the phone connection. |
| IPC_M_SHARE_OUT | Share out sender's memory area. |
| IPC_M_SHARE_IN | Create a sharing of receiver's memory area. |
| IPC_M_DATA_WRITE | Write data to recipient's address space. |
| IPC_M_DATA_READ | Read data from recipient's address space. |
| IPC_M_DEBUG_ALL | Start a debugging session on a recipient. |

Table 2.1: Basic system messages

different task than was originally intended. This mechanism is mainly used by the *Naming Service* (the 'NS') for creating new connections between the tasks.

When tasks are created they get their first phone connected to the Naming Service. This can be used for registering offered services (by servers) or on the other hand for making connections to the servers (by client tasks). The server notifies the Naming service about services it offers, NS answers this message and a new callback connection from NS to the server is created. When any user task asks the NS for a service it forwards the call to the corresponding server through the previously created connection. Thus the new connections from the tasks to the servers are created.

### 2.4.2 Data Copying and Sharing

When larger data are to be passed over the IPC connection two special mechanisms can be used.

First mechanism is used for copying the data between the task address spaces. A task can send a call to the other task to initialize either data reading or writing. This call contains only a size and a pointer to the source or target data respectively. If the target task acknowledges the call the data are copied between the tasks by the kernel.

Similar mechanism can be used also for data sharing simply by using respective message method numbers. Message order is the same, only the

size of the area is filled by the kernel. The memory sharing is created for the address space area specified in the call arguments.

### 2.4.3 Asynchronous Library

Because the whole IPC communication is performed between the tasks a serious problem can arise. If two threads or fibrils of one task send a call consequently and an answer to one of them is received it is not trivial to determine which one of the calls was answered.

To help programmers solve this problem quite sophisticated *Asynchronous library* was introduced. The library itself should be used for all IPC operations as it simplifies the communication quite significantly.

For more information about IPC subsystem design see HelenOS IPC for Dummies [8].

## 2.5 Sysinfo Interface

Sysinfo is the interface which serves for passing various system information from kernel to userspace tasks. All its items are stored in one sysinfo tree. Path to every node (which contains data) is given by a path string with items on the way separated by dots.

At the beginning of the work on this thesis the leaves of the tree could contain only data stored in fixed size integer values. However the implementation was very quickly expanded by Martin Děcký so now there can be many different types in the tree nodes and leaves. The values can also be generated dynamically by calling a function every time the corresponding node's value is requested. Thanks to these modifications the sysinfo interface shapes a powerful tool for passing information from kernel to userspace tasks.

## 2.6 Task Creation

Another very interesting (although not so much related to this work) part of HelenOS design is a process of a new userspace task creation.

In traditional (monolithic) kernels derived from original UNIX system there is a `fork` system call which makes a precise copy of the calling task (only with different task id or 'pid'). One of the new tasks (usually the

'child' one) can later use the `exec` system call to replace its code with the new one loaded from a file system.

This procedure is quite inefficient as the memory areas of the calling task are copied to be immediately replaced by the new ones obtained from the binary file used in the 'exec' system call.

HelenOS has a totally different system call `program_spawn_loader`. The result of the system call is also a newly task, however its code is a copy of the userspace task called *loader*. This loader receives the path to the new file executable to be loaded over IPC communication, loads the binary from the ELF image to its own memory and passes on the execution to the code loaded from the binary.

As this process is quite new in the HelenOS system it is not described in the original design materials. It was created as a supplementary part of the Jiří Svoboda's Master thesis [5] and as such is described there in more detail.

## 2.7 Synchronization Primitives

HelenOS kernel supports several types of synchronization primitives.

*Atomic variables* are supported on all architectures for lockless solutions. *Spinlocks* can be used for active waiting in Multi processor (SMP) configuration. Spinlocks are even used in two different ways – either with interrupts enabled or disabled.

For passive waiting there are only *wait queues* synchronization primitive similar to the counting semaphores. All other primitives are build upon them. Threads block on the wait queue until wakeup operation is performed.

The wait queues are used for implementation of fast user mutex primitive or simply *futex*. Futex operations are quite similar to mutexes except that successful locking can be done entirely in userspace.

Futex is the main synchronization primitive offered to user tasks. Standard libraries implement several other primitives such as *mutexes* and *semaphores* upon them.

## 2.8 More Reading

There are many other sources about HelenOS design which are more focused on introducing HelenOS design features than this thesis.

As the main document we would recommend original HelenOS design [9] which is however obsolete now. But almost any thesis, presentation or paper on HelenOS features can serve as good source for understanding how the system works. The newer ones are better as they present event features added more recently. There is an overall page which lists all existing documentation [10].

Of course the best and the most accurate (although the most difficult to read also) source of information about HelenOS is its source code itself. The central Bazaar development branch was very recently renamed from 'head' to 'mainline'. Currently it is therefore accessible at this address:

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

# Chapter 3

# Operating System Monitoring Overview

Let's have a look on what can even be monitored in the operating system. As was said the main purpose of the operating system is to run all user applications, separate them from each other and to correctly handle all exceptions and interrupts. The programs should be switched fast enough to make an illusion of parallel execution.

## 3.1 Monitoring Basics

Every computer has some hardware resources such as CPUs (possibly more than one), some amount of memory, disk space, network interfaces etc. Usage of every resource can be monitored in a percentage portions of its full capacity. On a CPU we can trace how many cycles were spent doing some useful job (compared to the wasted cycles), memory can be watched for the total size of allocated frames, network interface for total bandwidth usage and so on.

On the other hand there are some requirements on the system which mainly originate from the user space task calls. Tasks want to use the CPU, store their data in the memory and send or receive data through the network infrastructure. The operating system must handle all these requirements and satisfy them with as much resources as it can afford (according to the fact that all resources are limited). The system should pay attention to fairness in distribution of the resources between the tasks, although it cannot be simply told what this requirement exactly means. There are many policies

on how to distribute the resources and none of them is clearly the best. Some of them are better for certain conditions and needs, some of them for others.

As we can see the basic separation of the monitoring is between monitoring of the resources and requirements. Monitoring of the resources is quite simple as every resource is represented by some physical device of the computer and as such can me measured.

The situation with the requirements is more complex as it depends on the actual system design what exactly the requirement means. Typical systems of these days use threads as the main unit of execution, but there are many different services which the threads can require.

There are also layers of the operating system itself which don't fit into any of these groups. For example file system should be definitely considered as the part of the operating system (not necessarily of the kernel). A user thread requires access to some files but in fact it is the file system which reads the raw data and it is actually some low level driver which uses the hard drive. So we should measure the requests for the files (as the user requirements), the actual usage of the hard drive (as usage of the resource) and the time spent by the file system and the driver to handle the request (as the operating system overhead).

## 3.2 Resources Monitoring

Let us have a closer look on what resources and how can be monitored.

### 3.2.1 Processor

Every system has one or more *processors* (CPU) which are used for almost all computations. Every CPU processes instructions of both the user tasks and the kernel and essentially there is almost no difference between them (except the CPU operation mode).

It is very hard to say how long every instruction takes. Modern CPUs make a lots of optimizations thanks to which there is almost no correlation between amount of processed instructions and the CPU frequency. However most CPU models measure also the amount of processed clock cycles which are perfectly regular. Therefore they can be used as the unit of measurement. It is then not important how many cycles processing of every instruction took.

If there are some threads to run then the CPU is processing their instructions. But what if there is nothing to do? For this purpose there might be for example some special 'idle thread' prepared in the system. This idle thread would not do any useful job, it would just cycle forever. The operating system would be aware of it and schedule it only if there is no other thread to run.

In such case the CPU usage can be easily computed. It is the rate of cycles spent by the special idle thread and by all other threads during last short period of time.

However with today's CPUs the situation is slightly different. Idle cycling is just wasting of electricity, the most of modern CPUs therefore introduce a special *halt* instruction which stops CPU execution until it is waken up by some interrupt again. Luckily CPUs usually (at least in the case of common IA-32 architecture [11]) count passed cycles even in this case. If the CPU does not count the cycles while halted another technique must be used. For example the theoretical maximum of processed cycles per second can be computed according to the CPU speed. The CPU usage is then given as a rate of $max - computed$ to computed cycles.

Also some external source of time such as the *Real Time Clock* ('RTC') chip can be used to determine how long the CPU was sleeping. However this solution would be much less accurate than measuring based on cycles counted by the CPU itself.

The final trouble of the CPU monitoring are interrupts. As (one of) the CPU is the central unit of every computer it is also responsible for handling requests from other devices. These requests are delivered to the CPU in the form of interrupts which stop the original flow of instructions and force it to switch to the interrupt handling.

The instruction cycles spent on interrupt handling should be counted separately as they are often not caused by neither the code of the user tasks nor the system itself. If there is a network packet coming from outside world then the network card causes an interrupt and the CPU or the system has no (unless the network card offers such mask) way how to inhibit this behavior.

### 3.2.2 System Load

Although the current CPU utilization is definitely a piece of interesting information, it says nothing about the long time CPU usage. For example a thread which sleeps most of the time does not load the system a lot, but

it can now and then make some short intensive computations. The actual value of CPU usage is not sufficient to determine if there is still some free CPU time or not.

In traditional UNIX-like systems there is another value next to the pure CPU utilization computed for this purpose. The value is called *load* and is based on an average length of the scheduler queue through the past short period of time. Usually this value is measured for three different periods – that is 1, 5 and 15 minutes.

Quite interesting part is the load computation itself. Collecting the scheduler queue length samples through the time would be quite complicated (and time consuming). The load is therefore usually computed using a special decay function. The function itself is rather magic, there is a nice article on its implementation in GNU/Linux in Linux Journal [6]. We explore this algorithm in more detail in Section 4.4.2.

### 3.2.3 Memory

Measuring of the memory usage is quite straightforward. There is total amount of physical memory for use by the system. Major part of the memory is usually allocated by the user tasks. To compute the rate between allocated and free frame counts is a simple task.

On real systems free physical memory is usually further used as a buffer for IO operations. Disks are much slower than the main memory, therefore this principle can significantly speed up the whole system. However there are no such buffers used in HelenOS system yet because they are not needed for basic system operation.

Real systems are further able to handle the situations when there is not enough physical memory. Let us describe this mechanism although this is not the case of HelenOS neither. The lack of physical memory is usually handled by copying some of the memory pages out to the disk. This operation is commonly known as *swapping*. The swapped out memory pages are stored either in special disk partition (e.g. in the case of Unix system descendants) or in a file (e.g. Microsoft Windows system family). The great advantage of using a separate swap partition is the fact that data are not fragmented, but this can be achieved even for the swap file if the file system manages to do so.

Monitoring of all these additional memory features should be also implemented once they are introduced in the HelenOS system.

### 3.2.4 Storage

Also the usage of the hard drives and other storage facilities can be measured to get better overview of the resources used by the system.

Briefly the total amount of used and free space on the drive can be measured to get know how much data can be further stored on the drive before it becomes full. Because all drives in today's operating systems are used through some file systems their space usage is typically traced by the file system itself.

Next to the space we can also measure the speed of the data flowing to and from the drive. Every storage facility has some theoretical maximum speed of data transmission however the real speed is usually much lower. Thanks to the physical construction of the classical hard drives the total throughput depends on the order of the requests very significantly[1].

Similar to the CPU usage also the disk bandwidth usage can be measured. But since the total transfer speed much more depends on the order (and block numbers) of the requests this value is not very important and is not widely used.

### 3.2.5 Network Interfaces

Also network interfaces deserve some attention about how much data can be transmitted through them and how much they are really doing so. Usually every interface counts the transmitted packets. If the system knows the theoretical maximum speed of appropriate interface then it can also compute the interface usage.

Network usage is measured by almost all operating systems. For basic network infrastructure facilities such as routers, switches and firewalls this value is even one of the most important ones.

### 3.2.6 Others

Every physical device and I/O port of the computer can be measured somehow. For some of them this is important, but for many others not. Of course we can measure how much data is pouring through the serial port but as the serial ports are disappearing from today's computers nobody cares if there is still some free bandwidth.

[1] Although this is not the case of modern SSE disks whose construction allows real random access.

Basically the operating system can store the size of all data passing to the specific device. If it knows the maximum throughput then it can even compute the percentage usage, but for most of other devices these numbers are not such important as for the previously mentioned ones.

## 3.3 Requirements Monitoring

Let us have a look at which requirements of the user tasks can be monitored and how.

### 3.3.1 Tasks and Threads

In running system there are usually many user tasks containing even more threads. Some of them are running but the most of them are just waiting for some events such as a timer alarm or keystroke hit by a user. The count and state of all tasks and threads is the first thing which should be monitored. Every system should have some simple interface how to detect how many tasks and threads are there running at the moment.

There are few typical states which can be reached by the thread in the operating system. As the thread is born it enters the *Ready* state. System preemptive scheduling periodically switches the thread between the *Ready* and the *Running* state. If the thread sleeps on some wait queue it enters the *Sleeping* state. After it leaves its main function it enters the *Undead* state in which it can be joined (or detached) by its parent thread. Figure 3.1 shows overview of the most typical thread states.

The total count of all tasks can change very quickly because the tasks can created and destroyed very often. For example typical build of any software by common 'make' tool usually invokes the compiler for every single file, therefore new tasks are born many times per second.

### 3.3.2 Processor time

The main resource used by running threads is the CPU, so the CPU time they used should be measured. For better overview also a sum of the time periods used by all task threads should be computed.

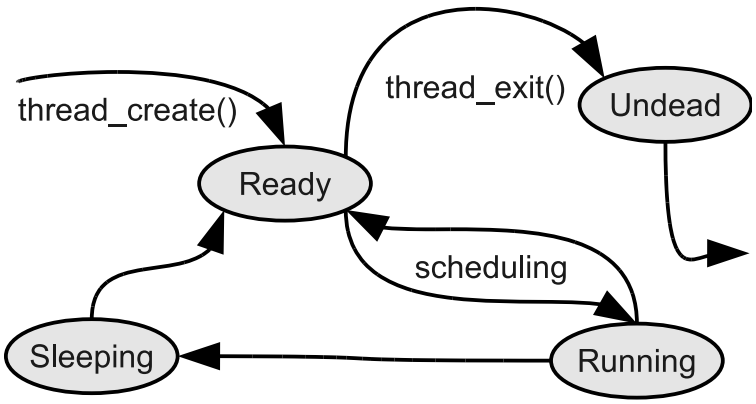



Figure 3.1: Thread states diagram

### 3.3.3 Memory

All tasks in modern system are separated by some king of virtual memory (usually *paging* in these days). Every task must first ask the system to make the mapping before it first accesses the area.

The system must keep the mapping of all areas to their relevant physical addresses for all tasks in order to handle the request for page table refills. The total amount of allocated memory for each task should be monitored for easy detection which task consumes the biggest amount of memory.

### 3.3.4 Syscalls

For standard interaction with outer space the task must ask the system to do so through some system call interface. Overview of all executed syscalls might be quite interesting for tracing task activities.

However this simple solution would work only for traditional systems based on monolithic kernel. In any system based on micro-kernel design the most used syscall by far is the one for sending messages or other respective communication with other tasks. All services in such systems are usually done through some other (server) task. In this case the total amount of syscalls done would not be so interesting as statistics of the sent messages which would show much more about what is going on in the tasks.

### 3.3.5 Others

Monitoring of the running tasks, their amount of allocated memory and interactions with the system is just the most basics. There are many other operations which the task can request from the system and all of them deserve to be monitored.

If the system offers some work with the files (and almost all systems does) we should measure how many file operations is being done by the tasks. The same applies to the network layer, graphics subsystem or any other system-specific feature. Simply said every feature offered by the system to the tasks should be also measured how much it is really used.

## 3.4 Real Monitoring Interface Overview

Let us have a look at how common operating systems expose their monitoring information. Among the most common systems are those of Microsoft Windows family and various descendants of old UNIX system so let's focus on them.

### 3.4.1 Microsoft Windows monitoring

For users there is a tool called "Task manager" which shows all basic system information. There are few tabs offering almost anything what normal user needs to know:

- **Performance** tab shows CPU and memory usage,
- **Networking** tab shows network usage and
- **Processes** tab shows information about all tasks in the system, how much do they use the CPU, how much memory do they consume etc.

In higher (and more expensive) variants of Windows system there is also a more advanced tool called "Performance monitor" which contains many *counters* which can be used for more detailed monitoring. These counters allows users to see how many bytes are send through specified interface, how much write operations are processed by specified hard drive etc.

How can be these data collected? There are two interfaces which allows programmer to access the performance counters. First, he can query registry

values on key `HKEY_PERFORMANCE_DATA`, or he can use special *PDH* functions [18] to obtain the same results.

How the monitoring itself is implemented is hard to say as Microsoft has not made the sources of Windows operating system opened to public.

### 3.4.2 UNIX Monitoring

Traditional UNIX-like systems (since ca. 1984) expose basic process information through a virtual file system called *procfs* and this solution is still widely used. Procfs looks like a standard file system with files and directories. But its directory structure and file contents are generated by the kernel every time the file system entries are accessed.

For every running process in the system there is one directory (named by its process id or *pid*) which contains plenty of information about it. The most important files (as they are found in GNU/Linux system) are listed:

- **fd** lists all opened file descriptors of the process,
- **stat** shows different statistics of the process such as used CPU *jiffies*[2], memory page faults,
- **status** shows status of the process, its total memory usage, user and group ids and many other information.

GNU/Linux system places some more virtual nodes in this directory. All these extra directories have names containing only alphabetic letters. It is therefore easy to distinguish them from the process directories which have names consisting of only numeric digits. These special directories and files contain overall system information. Here are some important items found in the procfs directory tree:

- **cpuinfo** lists static information about the CPU model (or models),
- **loadavg** shows average load of the system,
- **meminfo** lists various physical memory statistics,
- **interrupts** show statistics of time spent by interrupt handling,

[2]Time slices assigned from the system to the threads

- **stat** shows overall various statistics such as the CPU usage, number of running processes and count of served syscalls since boot.

There are also several other ways how to obtain statistics on UNIX systems, but the procfs is by far the most used one. From the others there is e.g. `getrusage()` function which retrieves resources used by particular process, `statvfs()` for file system usage statistics and others.

There are many resources dealing with GNU/Linux monitoring interface but quite good documentation is supplied as a part of kernel sources itself [12]. Search the subdirectory **Documentation**, for procfs overview look at the file `Documentation/filesystems/proc.txt` in particular.

The original aim of procfs design was to replace the ptrace system call. However today the ptrace syscall is used for process debugging and tracing opposite to procfs which is used more for monitoring and gathering statistical information about the processes.

### 3.4.3 Other Monitoring Tools

During more recent years another interesting tools were introduced. In 2003 Sun Microsystems made available DTrace tracing framework [4] for its Solaris operating system. It was by the way made public as the very first part of the OpenSolaris project later. DTrace was designed with the aim for tracing both kernel and userspace tools, but it can be used for monitoring purposes also. Amongst the best features of DTrace is the fact that for all tracing functions it uses strictly dynamic instrumentation. DTrace therefore has no affect on running system if its features are disabled, in fact there is no DTrace related code present in the kernel at all.

DTrace allows its users to attach many kinds of *probes*[3] on almost any place in the kernel. At these points user usually retrieves arguments of the functions, but DTrace can also just count how many times particular point was executed. As the result DTrace can be used for monitoring system state. An example for such probe follows:

```
syscall:::entry              // probe description
/pid == $1/                  // predicate
{
  @[probefunc] = count();    // action
}
```

[3]Small pieces of code that usually just stores the arguments of the function

Although a part of DTrace was unofficially ported also to the GNU/Linux, this system was still missing similar abilities. As a consequence main Linux vendors announced development of SystemTap, another tracing tool based on dynamic instrumentation. SystemTap introduces also many *probes* written in special scripting language, which are however compiled into a kernel module which is later inserted into the kernel itself.

Another accessible tracing tool is Linux Trace Toolkit (New Generation) which is however not so useful for system monitoring.

# Chapter 4

# HelenOS Monitoring Implementation

Let us introduce now the implementation of the HelenOS monitoring features which were implemented as a part of this thesis.

Because the implementation covers basic system structures it does not depend on any particular architecture. This is a great feature because no work is needed on monitoring facilities when porting HelenOS to new architectures.

All results of the monitoring features can be viewed by several tools which were made for this purpose. Their listing can be found in Appendix A.

## 4.1 Monitoring Implementation Overview

The monitoring data are collected on several places in the kernel. As all the information must be valid right at the moment they are requested they are not cached anywhere. All data are captured and handed over right at the moment when the request on them is detected.

At first all data were passed to the userspace using standard syscall handling procedure. But this solution was not very elegant. For example the monitored data set might depend on actual kernel build configuration, the interface should therefore also offer the list of all accessible information.

All monitored data follow a tree structure. A task contains some threads which further contains thread data such as CPU usage and priority. The interface used for obtaining these data should therefore reflect this tree structure. As a result the interface for transferring monitoring data to the

userspace was switched to (already existing) sysinfo interface which can reflect the tree structure of the data. The sysinfo subsystem was originally quite trivial and was missing advanced features for passing structured data. Martin Děcký however expanded its implementation to the current state so it can be used for passing almost any data without troubles.

Because the monitoring features became the part of the sysinfo subsystem its source code is also located there. Search the HelenOS kernel source code at `kernel/generic/src/sysinfo` subdirectory.

The main part of the monitoring code is stored in file `stats.c` located here. This file contains many functions which are registered to the sysinfo interface and which calls them anytime they are requested according to their names in the sysinfo subtree.

On every request the values are pulled from the system, however most of them are also measured during the normal system operation. Table 4.1 shows all sysinfo nodes which contains data covered by this thesis.

| Path | Content |
|---|---|
| `system.uptime` | Current system uptime |
| `system.exceptions` | Interrupt statistics |
| `system.cpus` | Overview of system CPUs usage |
| `system.physmem` | Overview of physical memory usage |
| `system.load` | Current system load values |
| `system.tasks` | Tasks statistics |
| `system.threads` | Threads statistics |

Table 4.1: Sysinfo system nodes and their values

## 4.2 Passing Data from Kernel to User Space

As was already mentioned at the beginning of this section for passing the monitoring results the sysinfo subsystem is used. For retrieving the data themselves special new syscall `SYS_SYSINFO_GET_DATA` was introduced.

This syscall requires userspace to pass four arguments:

- First one contains pointer to the string containing path of the selected sysinfo node,

- Second one contains size of the path string,
- Third one contains pointer to the buffer where the result will be stored,
- Fourth contains maximum size of the data which can be copied to the buffer.

Sysinfo subsystem calls the appropriate stats function which was registered during stats system initialization. Later it copies results of the stats function to the user space buffer specified in the third argument. Data themselves are copied using the standard `copy_to_uspace()` function.

## 4.3 Transition between Kernel and User Space

Crucial point for monitoring thread operations is a correct detection of passing the thread flow between the user and the kernel space. For this the perfect knowledge about all ways how can the user thread pass between both is required. Table 4.2 shows all ways how the kernel code can be entered.

| Name | Function | Cause |
|---|---|---|
| Syscall | syscall_handler | Thread executed syscall instruction |
| Interrupt | exc_dispatch | CPU triggered interrupt for some cause |

Table 4.2: Ways of entering the kernel code

After thread enters the kernel its stack is switched to the special kernel stack. Later even whole CPU context can be switched if scheduler decides not to run this thread any more. In such case another thread is picked up from one of the scheduler queues and its context is loaded on the CPU. All saved (Ready) threads must have entered their states through the scheduler where their context was saved right as they were leaving the scheduler. Restoring their context results therefore also in leaving the scheduler code.

However one more option of switching different thread is possible. If the thread decides to sleep on some wait queue it's context is saved by the wait queue handling code. There it is also restored after it's waiting is done and scheduler decides to run the thread again. This is why it is important to update the code there every time any change is done to the scheduler wake

up code. These two functions must be kept synchronized otherwise some weird situations may occur.

Usually similar behavior would be a sign of a badly designed code. However in the operating system kernel several copies of the very same code are sometimes desirable in order to optimize the execution.

## 4.4 Monitoring in the Kernel Space

Let us have a look on what work was done on monitoring features in the kernel space.

### 4.4.1 Measuring CPU Utilization Statistics

Every CPU today measures a count of processed cycles. This number is accessible in the HelenOS kernel by the `get_cycle()` function. As the CPU computes clock cycles even when halted (by `hlt` instruction on IA-32 architecture) our aim is just to separate which cycles were computed as *busy* and which the CPU was *idle*.

This distinction is done in file `kernel/generic/src/proc/scheduler.c` where the CPU is halted if there is no thread to schedule. In such case the difference in computed cycles since last check is processed and the CPU is marked as idle.

Because the only way how halted CPU can come to life again is through some interrupt handler we check the idle mark of the CPU there. If the CPU was idle, the difference in computed cycles since last check is added to the amount of cycles during which the CPU was idle.

The count of cycles should be also sometimes updated even when the CPU was so busy that it was not going to sleep at all. Therefore this number is updated on every clock time in `kernel/generic/src/time/clock.c`. Otherwise the CPU monitoring would return an obsolete values.

This solution has one great advantage over other systems and that is its precision. Traditionally the CPU usage measurement is based on clock ticks of the system timer – every tick is accounted to some task, system or is recognized as idle. However in such case the interrupt handling overhead is not measured because its granularity is simply finer than the clock tick frequency. It was better solution in days when CPUs were significantly slower but today it is much better to account every processed cycle of the CPU.

### 4.4.2 System Load Computation

As it was already mentioned in Section 3.2.2 traditional UNIX-like systems compute system load value next to the actual CPU usage. The meaning of the load value is not strictly defined, however its value usually somehow reflects the average length of the scheduler queue. The importance of the load value is based on an observation of its values. If the load value is less than the count of active CPUs in the system then there are more likely moments when some of the CPUs are idle. If the load is higher than the CPU count then all CPUs are probably fully used most of the time. This behavior should be preserved even on a micro kernel system.

In HelenOS we do the load computation in a way very similar to the algorithm used on a GNU/Linux system. In the GNU/Linux kernel the load computation is executed every few clock ticks right after the thread was switched from the system clock handler. Otherwise the load computation might get influenced by other system threads. However in HelenOS we do the load computation by a standalone system *kload* thread. This approach is much cleaner as the code computing load is separated from the clock handling routines which has nothing to deal with it.

The actual algorithm of computing load values was slightly simplified. Although the load value should somehow reflect the average count of threads running in the system, its computation is not based on mean value. A decay function is used instead. The common formula is

$$new = old * C + n * (1 - C),$$

where *old* stands for previous load value, $n$ stands for count of currently running threads and $C$ is the specific constant of decay speed. The GNU/Linux system load computation is based on exponential decay function, constant $C$ is therefore defined as $C = e^{-\frac{A}{T}}$. Here $A$ defines period of load recomputation (usually 5 seconds) and $T$ defines a load period (1, 5 or 15 minutes respectively).

As the inverse exponential function has no big influence on the results we simplified the $C$ constant as follows: $C = 1 - \frac{A}{T}$. The comparison of the GNU/Linux load values to our results is shown in Figure 4.1. The constants for the figure were set as $A = 5$, $T = 60$. We set count of running threads $n = 1$ for whole period and the initial value of $old = 0$.

Thanks to the HelenOS micro-kernel nature there is also one more difference on interpreting load values. Many tasks in the system are considered

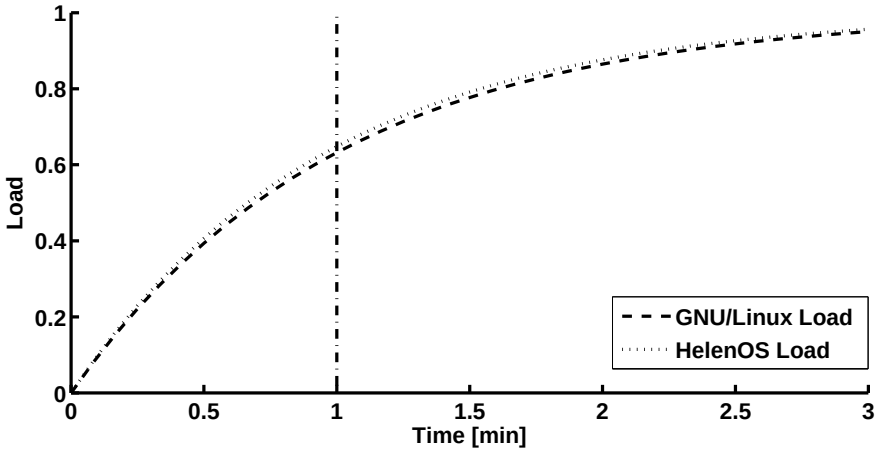



Figure 4.1: Load value comparison in time

as a part of the system itself (such as file systems, drivers and other servers). These tasks are sleeping most of the time and waiting for work to do. Only when some userspace threads are communicating with them they are running quite often. Even worse, they are usually awakened only for a short while which makes the length of the scheduler queue very flexible. Unfortunately it is hard to say what influence will have this fact on practical usage as there are not many real applications on HelenOS yet.

### 4.4.3 Interrupt Handling Time Measurement

Interrupt handling can consume a lot of CPU time if there are some very active devices present in the system (such as Gigabit Ethernet network card). Therefore time spent in each interrupt handler is measured to gain overview on them.

Interrupt measuring is quite simple because all interrupt handling routines must register themselves in one system interrupt handler table. When an interrupt comes the CPU context is switched and `exc_dispatch()` routine in `kernel/generic/src/interrupt/interrupt.c` is called. Here we account interrupted task for last cycles spent, run the interrupt handler and after it we account the handler itself for spent CPU cycles.

If the scheduler is run from a clock interrupt handler the current thread is rescheduled. However the CPU cycles are accounted right before the old

thread is scheduled out. The newly loaded thread stores the current CPU cycle value before leaving the scheduler code. Finally the interrupt handler is charged only for cycles which took the thread to leave the interrupt handler.

This approach is simple and it works quite well. The results can be found in the sysinfo tree under the `system.exceptions` node.

### 4.4.4 Physical Memory Usage

HelenOS keeps track of a physical memory space usage in several *frame zones*. Each zone represents one continuous block of the physical memory, it traces which memory frames in the zone are allocated and which are not.

The usage of physical memory is computed simply as a sum of all frame zones. On request for `system.physmem` sysinfo node the sizes of all memory areas are retrieved and the sum of them is returned.

### 4.4.5 Task Statistics

Most of the task data is just copied from the system structures in the stats request handling function. However few values are more complicated to gain.

#### Task CPU Cycles

Task cycles are gained as a sum of all cycles spent by its threads. The sum is recomputed every time it is requested as cycles processed on behalf of each thread can vary anytime. Before the results are passed to the userspace the amount of cycles is converted into microseconds because this is the unit which users expect.

This approach supposes the frequency of all CPUs constant which must not be true in the future. In HelenOS the frequency is not changed during runtime yet, but all modern processors have abilities to lower their working frequency in order to save energy when the system is not under heavy load. However frequency might be changed only to some fraction of the full speed, so this fraction can be saved on a frequency change request and the cycles can be appropriately adjusted any time they are computed.

Thank to this observation the cycles are converted into milliseconds simply by dividing its value with the CPU frequency. The frequency can be polled from the CPU itself using special instructions or can be measured by computing a dummy cycle for a while. The latter solution is nowadays used by HelenOS system to detect the frequency. In both cases there might be

however some inaccuracy so the results should not be considered as absolutely exact.

#### Virtual Memory Allocated by the Task

For every task the amount of mapped virtual memory is computed. This is simply done by passing through all its address space areas and adding its page counts. Later this should be changed in a way that address space handling code will adjust total size of address space every time it allocates or frees some pages in the address space.

#### IPC Messages Sent and Received by the Task

All IPC messages passing through the task's answerbox are monitored. This is done in the IPC handling code in `kernel/generic/src/ipc/ipc.c` right during processing of the message itself.

Overview of the monitored messages is shown in Table 4.3. Basically processing of every message increments one of these counters by one. The results can be obtained from the sysinfo interface from the `system.tasks` node because these data are part of the task statistics.

| Name | Description |
|---|---|
| `call_sent` | Count of the calls sent |
| `call_recieved` | Count of the calls received |
| `answer_sent` | Count of the answers sent |
| `answer_received` | Count of the answers received |
| `irq_notif_received` | Count of the received IRQ notification calls |
| `forwarded` | Count of the forwarded calls |

Table 4.3: IPC messages monitoring overview

### 4.4.6 Thread Statistics

Gathering statistics of a single thread is quite similar to gathering corresponding task statistics. Most of the fields are just copied from the system structures.

As the spent CPU cycles are measured per thread (not per task) every time the thread is scheduled the current CPU cycle count is saved in the

thread structure. When the scheduler is executed again the difference of the CPU cycles is computed and the result is added to the thread cycles account.

To be more precise the cycles are divided between user and kernel ones. During standard operation the cycles are counted as a user time. In every entry point to the kernel (which is only possible through a syscall or interrupt) the current user cycles are accounted and the thread is switched to the system accounting. When the thread is leaving system code the system cycles are charged at the end of the syscall function or the interrupt handler respectively. There is however one more point where the system cycles must be accounted and that is when the thread is waking up from sleep on some wait queue. A short scheduler wakeup code is executed there doing the some operations as a thread leaving scheduler would do.

## 4.5 Future Improvements

There are many points where this (quite simple) monitoring subsystem can be expanded. We just present some of them:

- There should be a way how to measure all services which are not parts of the kernel itself. Example of these services are be opened files, network sockets or console operations. For these services it is not sufficient to measure only IPC messages and CPU usage, users need to have brief overview of opened files and network streams. A flexible framework would be useful to ease gathering of information from the corresponding servers.

- When paging or caching features are introduces in HelenOS system sizes of such memory areas should be also measured.

- When user accounts are introduced in the system all monitoring structures should be adjusted for them. Almost all data described here would have some relation with the user who was responsible for them.

- The CPU time might be divided also between user and system time. Going even further the time spent in the kernel might be divided between its subsystems. Overview of how many time was spent by handling memory operations, IPC message passing or task creation would be definitely useful.

- The userspace monitoring tools can be improved significantly. 'Top' utility should show also some more raw data or statistics per selected threads. It should also offer some way for sorting and filtering the showed data.

As monitoring facility is quite essential part of every real operating system these features should be maintained together with other system development. Work described here is not done, it is just a foundation of a subsystem which needs ongoing sustainment in order to keep track with the whole system.

# Chapter 5

# Profiling Overview

The goal of every profiling is a better understanding of the operations performed by the program. This is achieved by employing some sort of dynamic inspection of the program while it is running. As such it is opposite to the static analysis which has the similar goal but inspects just the program code (or binary) without running it instead.

There are several different profiling techniques which can be used. However all of them share the basic idea of collecting some samples describing the running program. The implementation itself shares some techniques with debugging tools as they both need to trace the program states. The difference is that a debugger usually serves for monitoring smaller areas of code (where we expect the bug to be hidden) in contrast to profilers which rather monitor whole application as one piece and build overall statistics of the most exposed code.

Let us have a closer look on how different types of computer programs can be profiled.

## 5.1 Profiling User Programs

Profiling of the user programs is the simplest. Basically we can add some special piece of code to every place where we want to store some runtime information. These data can be later used (after necessary processing) to build an overview of called functions and other operations performed by the program.

### 5.1.1 Profiling Based on Source Code Modifications

Usually a programmer wants to know how many times the program functions were executed and how long did the program spend by running each one of them.

A trivial solution is to insert some profiling code into the source code by hand. We cannot insert a profiling code before every line of code because it would pollute the code a lot. Despite this fact this solution is used quite often because it does not need any special tools. If we want to know how many times one particular function is called there is not a simpler solution than to insert an incrementing counter into the function.

The most of compilers offer an option to automate this process to make a programmer's life easier. The compiler inserts a call to a profiling function before every call of the user function. The profiling function stores the current time and name of the user function. After the profiled user functions ends the compiler runs another profiling function. This function gets the current time again and subtract the first time. The result gives the time spent by the user function. The pure difference contains also the times spent by all nested functions. By subtracting also the time spent by the inner functions we get the time spent by executing the original user function itself.

Also the overhead of the profiling facility can be subtracted. This can be done by storing the time at the beginning and at the end of every profiling function. The difference between the former and the latter gives the time spent by the profiling function itself. Using this principle we get the total time spent by executing each user function. If we sort them we get the most exposed functions in the whole program.

### 5.1.2 Profiling without Source Code Modifications

Even if we don't posses the source code (or we are just not able or allowed to recompile it) the profiling is still possible. As the function calls are present also in the binary files these calls can be replaced the same way as it was mentioned in previous section. Going even further this replacement (or *instrumentation*) of the calls in the binary can be done while the profiled program is running (if the operating system let us to do so). If we attach the code of the profiling functions to the binary image in the memory we can call our profiling function before all original function calls.

User applications are usually not allowed to change the code of other

applications. Instrumentation must therefore be done at some lower layer, either in the operating system, in the language interpreter or even in the virtual machine the code is running on.

## 5.2 Profiling Operating System Code

How can we profile the operating system itself? There is no lower layer which can instrument the system with calls to the profiling functions. Although theoretically also the hardware can trace which functions were called this is usually not utilized because the hardware should not depend on currently running code.

The first possible way of profiling operating system is to recompile all code and insert calls to the profiling functions at places we want them. But replacing whole operating system is usually quite painful operation and leaving the calls at their places all the time would slower the system significantly without any benefits.

A better solution is to insert profiling calls to the system dynamically by using instrumentation techniques similar to the ones we have just described for profiling of the user tasks. This solution is more difficult to implement however it offers the best results. Only a selected part of the system can be profiled and if the profiling is disabled there is not any part of its code at all. This means that disabled profiling has no influence on the system performance.

## 5.3 Statistical Profiling

The idea of a *statistical profiling* offers quite a different profiling solution. It does not follow a flow of the operations and the calls of the functions done by the program. Instead it is built around an external source of regular clock events. At each clock signal the current value of *Instruction Pointer* (IP) register is saved. Instruction pointer register always contains address of currently executed instruction.

Binary images of all programs are loaded in the virtual memory according to the area maps stored in their binary file headers[1]. Therefore the program code addresses are the same every time the program is run. By collecting the

[1]Unless some address space loading randomization security technique is employed. In such case the randomization would must have been deciphered.

IP register values regularly it is possible to identify the areas of the program code which were running.

If the binary file contains also the table with addresses of the function names (the 'symbol table') it is further possible to translate the addresses to the symbols. Therefore even the names of the interrupted functions can be gained.

There are some technical details on how to get the value of Instruction Pointer register as it is immediately changed after the CPU enters the interrupt handler. Basically the CPU stores somewhere the context prior to entering the interrupt handler. From there the interrupted Instruction Pointer value can be restored.

The only output of statistical profiling is the list of the functions which were running when the clock event occur. From this list we can only gain counts of how many times every function was encountered. But if the measuring events are regular then the function counts will reasonably represent the time periods spent by executing each function. Thus the numbers will show which functions were exposed the most.

The source of clock events should be as much independent on the system as possible because its regularity is the main requirement for the assumption of the correlation between the sample counts and the time spent in each function. Attaching special physical clock device to the computer every time the profiler is executed would be a strong requirement. Therefore several different internal computer clocks are usually used instead. They might slightly influence the normal system operation, but this is not considered as a significant problem.

### 5.3.1 Event Sources for Statistical Profiling

There are several clock event sources which can be used for statistical profiling. All of them are an integral part of the most of recent computers so no special device is needed.

For the best results we need the event frequency as high as all running threads will be at least once interrupted. Otherwise we might not get any samples for some threads. A time slice assigned to the threads is usually in an order of tens milliseconds. However many threads are not using whole granted slice because they get blocked on some other operation first. Therefore the frequency of the event source should be in order of the kilohertz at least.

### System Clock Handler

As the easiest solution a common *system clock* handler can be used. The advantage is that every modern operating system must already utilize some sort of clock events which are used mainly for preemptive thread switching. It is a simple task to add a call of a profiling routine into the system clock handler.

The biggest drawback of using the system clock handler is its executing frequency which is usually less than hundred hertz. This speed is enough for making a fiction of applications running in parallel for human being.

The system clock events on IA-32 architecture are generated either by APIC controller or by original Intel 8253/8254 compatible timer. Today these chips are made by several other vendors. The timer chip offers three separate pins on which it can generate a periodical signal. However only the first one is connected to the system interrupt controller as IRQ 0. This one is used as a system timer if APIC controller is not configured.

The Intel 8253 timer can be setup for higher frequency. But the APIC also generates the clock interrupts as IRQ 0, so the Intel 8253 interrupts are masked on IO APIC in such case. As a result Intel 8253 events cannot be enabled together with the signals generated by APIC.

### Real Time Clock Integrated Circuit

A different source of events can be supplied by a *Real Time Clock* chip. Every architecture has similar device which can be used for keeping track of time even when the operating system is not running. This device can be usually also instructed to generate a periodical clock signal of a specified frequency.

RTC chip can be therefore used for the statistical profiling. If this chip is reconfigured the hardware clock might loose the track of real time. But this problem can still be fixed after profiling ends. The system handler is used to keep track of time while the system is running, therefore it is also possible to restore the correct time in the RTC chip.

### CPU Performance Monitoring Counters Facility

The last but probably the best solution for the statistical profiling is the CPU *Performance Monitoring Counters*. Majority of modern CPUs (in case of IA-32 architecture since times of Intel Pentium chip) contains specific registers

which can be used for counting of different operations or their durations as they are performed by the CPU. It is also possible to set a limit on the counter register value. When the counter reaches this limit the CPU triggers an interrupt to notify the operating system about this situation.

There are two main advantages on using the Performance Monitoring Counter registers. The first advantage is that they are not needed for a common system operation and as such they are not used in the operating system. The second one is the fact that the interrupts emitted by the CPU itself are usually not maskable[2]. Thanks to this fact all operating system code including its kernel can be interrupted and thus profiled. Also there is no delay on delivery of this interrupts as they can be triggered anytime. But a special attention must be paid when writing corresponding interrupt handlers, because these handlers must be for example lock-less.

## 5.4 Overview of Existing Profilers

Let's have a look on implementations of several existing profilers.

### 5.4.1 The GNU Profiler, gprof

The GNU profiler is the most common representative of a profiler attached as a part of the compiler itself. If the user program is compiled with the GNU C Compiler (gcc) with the -pg option enabled, every function call in the final binary is preceded by the call of `mcount()` function which stores the current IP register value. Also the system is instructed (usually by `profil()` system call) to run a statistical profiler.

After the program ends the profiling results are written into an output file. This file can be later analyzed by a separated program *gprof*.

Very similar profilers are also available for other languages. But the majority of modern languages is running on a virtual machine, the profiling facility is therefore a part of the underlying virtual machine itself. Java Virtual Machine Profiler Interface (JVMPI) is an example of such one.

[2]Term 'not maskable' is usually used for interrupts which cannot be masked by standard interrupt masking technique (using EFLAGS register on IA-32 architecture). However they can still be masked using some other specific ways.

### 5.4.2 OProfile

OProfile is a statistical profiler included in the GNU/Linux operating system [13]. As it is bundled with the system it can profile all user applications as well as the operating system itself.

All three different types of clock signal sources are supported by OProfile. If the system CPU contains hardware performance counter registers than the CPU can be used for emitting periodical clock signals. The most of today's CPU models are supported.

If the performance monitoring counters are not available (either by the CPU model or OProfile missing feature) OProfile falls back to the common system clock handler (on GNU/Linux kernel version 2.6) or to Real Time Clock chip (on kernel version 2.4).

All the system code is profiled because the clock signal source is not aware of currently running thread. Only samples belonging to the profiled task can be preserved while others are discarded. The user is responsible for setting the frequency sufficient to interrupt the requested thread.

### 5.4.3 Pin Tool

Pin tool offers interesting profiling solution if the source code of the profiled application is not available. It uses dynamic instrumentation techniques for detecting an actions performed by the monitored application.

It can be used for observing functions called by the application, so it can also be used for profiling purposes.

### 5.4.4 DTrace

As was mentioned earlier, DTrace is a very powerful tool for monitoring many operations done by both user applications as well as by the operating system itself. The monitoring is set up by activating some of the available probes, which are fired when the respective situation occurs.

One of the actions which can be done on firing a probe is just to "count an event". If the probe is instrumented to fire on every function call, we get a brief profiling overview.

### 5.4.5 Minix Profiling Implementation

For profiling the Minix operating system a brand new profiler was introduced [2]. In fact there are two separated profilers - a statistical and a call one.

Statistical profiler is based on events emitted by the Real Time Clock (RTC) chip. The results are directly written from kernel interrupt handler into the memory of the profiler. After the profiling ends the data from memory are written to a file which is later analyzed by the prepared PERL script.

The call profiler is based on a feature of ACK compiler (which is used for Minix compilation by default). It can call a special supplied function on every begin and end of a function. This special function is therefore used for storing the current call pathes and call times, which are used for later building of an overview of the whole profiling call tree.

# Chapter 6

# Statistical Profiler Implementation on HelenOS

Let us show how the statistical profiler was implemented for the HelenOS operating system.

## 6.1 Basic Architecture

There are three basic parts of the profiler, see Figure 6.1 for a brief overview. First, the userspace control utility allows user to start and stop the profiler and to receive the data samples from the kernel. These samples can be also parsed and printed. Second, in the kernel there is the core profiler facility which stores the Instruction Pointer samples on the clock signals in its buffer. The third important part is the clock signal source itself.

When the buffer of samples is getting full all samples are passed to the userspace tool for later processing. This tool can search for the Instruction Pointer values in the symbol tables are produce overview of the highly exposed functions.

## 6.2 Communication between Kernel and User Space

One of the important questions was how pass data between the userspace control application and the kernel structures.

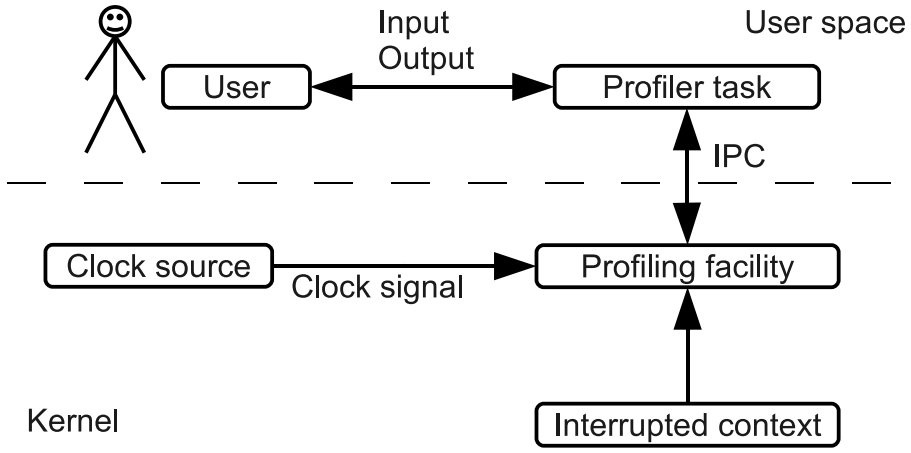



Figure 6.1: Profiler implementation overview

At first implementation of the data passing was based on standard syscall scheme. Several new syscalls were introduced allowing the applications to start, stop and receive all profiling data from the kernel.

But there were many troubles with this solution. The most important one was the statically allocated sample buffer in the kernel. Not even it was consuming much space in the kernel itself, also its overflowing meant losing all future samples. Dynamic allocation of the buffer would not solve the issue as it would just prolong the time when the buffer would overflow. Of course any buffer in the kernel cannot grow beyond all limits.

Simply speaking the samples must be passed from the kernel to userspace continuously because they should not be stored in the kernel.

To accomplish this goal the solution originally designed for userspace debugging support was employed [5]. The work on userspace debuggers introduced special kernel answer box (the `kbox`) in every task which can be used for passing kernel messages related to the task. The messages sent to this box are used for passing commands from the debugger to control the debugged application.

The connection to the selected task kernel box is created by a syscall `SYS_IPC_CONNECT_KBOX`. Further communication is done using the very same way as a standard IPC message passing - even the same functions are used. The connection ends when one of the parties hangs up the connection phone.

Thank to this solution all data can be passed using the standard IPC

messages continuously and they are not stored in the kernel. Next if the messages are later passed through the network between different hosts the profiler (as well as the debugger) can easily run on different machine than the profiled (debugged) application.

Following the solution of debugging support several new kernel box messages were introduced. They are defined in `kernel/generic/include/profile/abi.h`. The overview of them is listed in the Table 6.1.

| Message type | Message description |
|---|---|
| `SPROFILE_M_TASK_BEGIN` | Establish a new task profiling session. |
| `SPROFILE_M_SYSTEM_BEGIN` | Establish a new system profiling session. |
| `SPROFILE_M_END` | End current session. |
| `SPROFILE_M_PAUSE` | Pause or resume current session. |
| `SPROFILE_M_DATA_READ` | Wait for profiling data. |

Table 6.1: Profiling related kernel box messages

### 6.2.1 Profiling Message Format

Common profiling message structure follows the content of the standard IPC messages. Table 6.2 shows the brief overview of the message payload.

| Method | Arg1 | Arg2 | Arg3 | Arg4 | Arg5 |
|---|---|---|---|---|---|
| `IPC_M_SPROFILE_ALL` | Profile command | a1 | a2 | a3 | a4 |

Table 6.2: General profile message structure

The message method must be the same for all profiling messages in order to distinguish them from other system messages sent to the kernel answerbox. First message argument identifies the message command, the rest of the arguments depend on the profiling command specified.

The reply message structure follows the standard message format even more. First argument contains the return value, other arguments are specific to the reply message type.

## 6.3 Profiling Operation Modes

In fact two slightly different profiling modes were implemented. Profiler can use the messages to the kernel answer box to start profiling of either a whole system or just of a task which owns the used kernel answerbox.

Both profiling modes can also be paused. In such case no new samples are generated until the profiling is resumed again.

### 6.3.1 Profiling Specific Task

Message `SPROFILE_M_TASK_BEGIN` sent to the task can be used to start profiling of this selected task. In this case the samples are stored only if one of the task threads was interrupted when the clock signal came.

If the interrupt came when the thread was running in the kernel space only the counter of kernel samples is incremented. This counter serves for a brief overview of how much often the kernel code was running.

The structure of the samples is very simple. It contains only the interrupted IP register value:

```
/** Sample for task profiling. */
typedef struct {
    uintptr_t pc; /**< IP register value. */
} sprofile_task_entry_t;
```

### 6.3.2 Profiling the Whole System

In this mode the samples are stored on every clock signal. The samples belonging to the profiler itself are the only exception. These samples are dropped because profiler should influence the results as few as possible.

In contrast to the task profiling, this mode stores also the samples belonging to the kernel space. These samples are also important to gain an overview of all system operations.

The samples must definitely contain also an id of the interrupted task in order to distinguish the running tasks from each other. But there is one more question on the system sample structure.

#### Symbol Table Location

For each sample we need to identify also the corresponding symbol table. For a task profiler this is simple – the symbol table can be loaded from its

binary file. But for system profiler we need the binary file for every task running in the system. Even worse we need the symbol table also for the kernel itself.

How to identify the symbol table? The simplest solution is to add the name of the task to every profiling sample. The corresponding binary file can later be found according to the task name. The best aspect of this solution is its simplicity. The history of all tasks which were running is kept in the samples themselves.

The main problem is that it is uncertain that the real path of the binary from which the task originated can be constructed simply from the task name. Task name is basically only an information tag for the user. The real path to the binary is known only to the particular task loader, which can set the task name to any arbitrary string. Alternative solution should be therefore based on storing the path to the binary files elsewhere for every task which was running. In such case the samples would contain only the id of the corresponding task.

Where to store the path to binary file which was loaded? The Naming Service serves as the only userspace central point which is aware of all tasks. Next to the connecting the clients to the server it is also used for passing the return values from terminated tasks to their parents. So it can also serve as a central repository for other task related information. Every user task registers to the NS during its loader stage, every server registers all services it offers. All tasks can therefore register their binaries to the NS. The profiler can retrieve the path for every task from NS while it is going through the samples.

The worst situation comes with the init tasks and the kernel. These are loaded by the architecture boot loader and as such their binaries don't even need to be present in the file system. In such case the symbol tables must be supplied by the user or the init tasks (and the kernel) would not be profiled at all. Let us ignore the init tasks further in this section.

The profiler can theoretically retrieve all relevant data from the NS while the profiling session is running. But some tasks can be created and terminated without being noticed by the profiler. The NS should therefore keep the task data even after the task terminates until the profiler requests them. For this service some special operation mode of the NS should be created.

The pitfall in the solution relies in the danger of memory leaks. If the profiler does not tell the NS to drop the path to the binary the NS would store the data forever. The NS can keep an eye on the profiler if it is still

running but this solution is not very elegant.

After serious consideration we implemented the former simple solution. The latter is much more complicated and has some drawbacks though. The whole problem is moreover only weakly connected with the thesis itself because it covers only the way how to obtain the symbol tables for system-wide profiling.

Therefore the system sample structure contains both the task id and its name. The corresponding header file contains this code:

```
/** Sample for system profiling */
typedef struct {
    task_id_t taskid;                 /**< Id of the task */
    char taskname[TASK_NAME_BUFLEN]; /**< Name of the task */
    uintptr_t pc;                     /**< IP register value */
} sprofile_system_entry_t;
```

### Choosing the Kernel Answerbox

One another important question had to be solved. For profiling selected task we sent all profiling messages to the kernel answer box of this task, but which answer box can be used for profiling whole system? Basically we need one which will not disappear while the profiling session is running.

There are about three answerboxes which fulfill this requirement:

- First we might introduce special global answer box for communication with the system. This would be needless complication as other answerboxes already exist in the system.
- Next we can use the answerbox of the kernel task. This task will not terminate while the system is running.
- Last we can connect to the profiler itself making a connection loop. The profiler answerbox will exist right as long as the profiler itself.

The last solution was implemented as it keeps both sides of the connection in the userspace. Also no special message handling in the kernel is necessary. It may look like we cannot use another profiler for profiling system profiler, but even this case (if requested) can be solved by running two profilers and connect their phones from one to another.

## 6.4 Work Done in Kernel Space

All files relevant to the profiling structures are stored in subdirectory **profile** in the kernel generic sources. Here we find the main file `sprofile.c`.

The syscall handler of `SYS_IPC_CONNECT_KBOX` call starts special kernel thread which waits for the IPC kbox messages. When the message is received the kbox thread checks the type of the message and passes it to the corresponding handling routine (either debugging or profiling subsystem). Whole kernel part of profiling support thus runs in the context of the kbox thread.

The kernel box thread takes care about all communication with the user space profiler. On its request it starts and stops the profiling itself and also passes the data from kernel sample buffer to the userspace profile tool.

Samples are collected by the `sprofile()` function in the kernel which is periodically called while the profiling is running. There are several clock sources which are described later in this section.

### 6.4.1 Real Time Clock Driver Implementation

On the IA-32 architecture the *Real Time Clock* (RTC) driver can be used as a source of profiling events. For this purpose a limited driver of the clock was implemented.

The driver is stored in `kernel/arch/ia32/src/driver/rtc.c` file. It contains the most basic functions which can start and stop the interrupts emitted by Real Time Clock chip as well as the handler which is registered for handling these interrupts. The handler reinitializes the RTC chip and calls the `sprofile()` routine.

The operations needed for setting the RTC circuit are done according to Dallas Semiconductor DS12887 chip data sheet [3]. The values needed to start and stop the chip operations are written to the clock chip port using the standard system port output functions.

The RTC chip can be set to emit interrupts in one of predefined frequencies only, that is from 2Hz to 8kHz in increasing powers of 2. RTC driver thus cannot be set to an arbitrary sampling frequency.

### 6.4.2 Performance Monitoring Counters

Also driver for internal CPU Performance Monitoring Counters (PMC) was implemented. Because these counters vary a lot from one architecture to

another we support them only on IA-32 architecture. Unfortunately even implementation of performance monitoring features is quite different on specific CPU models. Thus only CPUs produced by Intel vendor are supported. The CPU must further be one of Intel P6 family (Pentium Pro, Pentium II, ...) or newer because earlier models are missing Performance Monitoring features at all (or has these quite different in case of Intel Pentium model). The used counter is nowadays considered as a part of an *Architectural Performance Monitoring* features and as such should be present in all future models of IA-32 architecture processors made by Intel vendor. See Chapter 30 of Intel IA-32 Architecture Software Developer's manual [11] for more information. Nice example of basic usage of PMC registers for counting CPU cycles can be found in [15].

### Setting the Appropriate Registers

Although majority of CPU models offer usually many performance monitoring counters, only one of them is used for profiling. All performance monitoring features are set by writing into different Model Specific Registers (MSR). Usually there is one for selecting the event to monitor (the *Performance Event Select* register) and another one for counting itself (the *Performance Monitoring Counter* register).

The Performance Monitoring Counter register is firstly set to some predefined value (explained later). Then the monitored event and few other options are set into the Performance Event Select register. After this operation the CPU increases the counter by one on each event of the selected type. On counter overflow the CPU emits the Non-Maskable interrupt[1] (NMI) which is used for profiling.

There are several monitoring events which can be used for statistical profiling. Basically we need one which is continuously occurring not regarding on what the CPU is exactly doing. These events vary from one CPU model to another[2], but basically following options can be counted:

- Unhalted core cycles event
- Instruction retired event

[1] Number 2 on IA-32 architecture

[2] Especially newer models with Hyper-Threading and Virtual Machine extensions introduced many new performance monitoring events depending on virtual cores

The latter event does not occur regularly because processing of every instruction took different amount of cycles on today's CPUs. Therefore the first mentioned event was chosen, which has also the same event code on most CPU models. As we can see the cycles are not counted while the CPU is halted which leads to the fact that also profiling interrupts are not generated while the CPU is halted. But as the CPU is halted only when there is no thread to run (and also no code to profile) this is not a problem.

The interrupt is emitted on the counter overflow. The counter is therefore preset to some high value in order to cause its overflow quite often. From the CPU clock frequency the right value can be computed and the counter can be preset to emit the interrupt in the requested profiling time period.

#### Setting the Interrupt Controller

The interrupts themselves are emitted on the local part of the *APIC* (Advanced Programmable Interrupt Controller). This controller must be enabled and set up to receive the NMI interrupts on the CPU before the profiling starts. Unfortunately one serious drawback arises from this fact - HelenOS system does not use the APIC controller on IA-32 architecture in single CPU configuration. It uses the standard PIC (Intel 8259 chip or its descendant) controller instead. As the result the Performance Monitoring counters on HelenOS cannot trigger an interrupt in single processor configuration. If there are more than one CPU then everything works fine.

The HelenOS should be later modified to use APIC even in single CPU configuration. However this work is beyond the scope of this thesis.

### 6.4.3 Setting the Clock in SMP Environment

Setting the clock event source requires some more special handling if there are more than one processor present in the system. For profiling we need to receive the interrupt on all system CPUs, however this is not the done automatically.

#### RTC Clock Chip

The RTC chip generates the interrupts without any attention on how many CPUs are there in the system. The interrupt is passed to the IO part of the APIC controller which decides to which CPU the interrupt will be delivered.

We need the interrupt to be immediately delivered on all CPUs in the system in order to profile all code running in the system. Therefore we need some special hardware which will spread the interrupt on all CPUs. Fortunately every SMP system contains such device called *IPI*. This device can broadcast simple[3] piece of information between the CPUs. The CPU which receives the RTC interrupt triggers another interrupt by broadcasting over the IPI bus. The `sprofile()` function is called in the handler of this IPI interrupt.

### PMC Registers

Setting the PMC registers requires also special handling on Multi Processor systems. These registers are specific to every CPU core, so their setting must be done on all cores. If we set these counters for example in the handler of profile start message, then they will be set only on the CPU on which the current kernel answer box thread was running in the moment when the message was received.

As a result we have to employ some technique for running a code on all CPUs. There are following options for this purpose:

- Global clock start flag
- CPU bind kernel threads
- Signal broadcast over IPI between CPUs

The former solution is based on the global flag which is periodically checked[4] on all CPUs. If we set this flag from the message handler, all CPU sooner or later observes this fact and sets their PMC registers appropriately. The benefit is that we don't need any special constructs, the drawback is that the flag must be periodically checked even when the profiling subsystem is not used. It can also take a long time till all CPUs notice the flag.

The second option is based on starting several special kernel threads from the start message handler. Every thread can be bound to a different processor. After these threads are scheduled they simply set the PMC registers and end. This solution is very cumbrous because a creation of a new thread is quite time consuming operation. Also it might take a long time till the

[3] Usually just one byte long

[4] For example in the system clock handler

threads are scheduled to run thus the clock start might be delayed from the message handler.

Last but probably the best solution is to employ the IPI bus again to broadcast the signal to set the PMC registers between the CPUs. There is no bigger overhead and the PMC counters are set almost immediately after the start message is received.

### 6.4.4 Starting the Profiler

Receiving every type of message is served by special routine. Starting and stopping the profiler is done by few independent operations:

- At First the task (or system) must be marked as being profiled. This mark is checked in profiler clock handler to decide which samples are going to be stored.
- Further some kind of clock signal source must be enabled. This operation might need some architecture-specific actions as different types of computers have also different types of supporting hardware. Basically the Performance Counter in the CPU, the Real Time Clock or the common system clock handler can be used as a source of profiling events.

For keeping track of how many profilers are there running in the system one global atomic variable is used. The relevant hardware source of profiling events is enabled if this number is higher than zero; if it falls to zero the hardware device is stopped.

### 6.4.5 Stopping the Profiler

Stopping the profiler is just the opposite operation to starting. The hardware used for emitting the signals is stopped and the task or system mark is removed so no samples are gathered anymore.

### 6.4.6 Pausing the Profiler

Pausing the profiler is quite simple operation. Only the state mark is changed to its paused variant according to the profiling type being paused. This state means no more samples are stored, but the profiling session can be easily restored by just returning the state mark to its original value.

### 6.4.7 Storing Profiling Samples

The profiling samples are gathered in the profiling routine which is executed as an action taken on profiling clock events. Usually it is the main part of the corresponding interrupt handler.

The Instruction Pointer register value is gained from the interrupted task state, which is saved to the `cpu_t` structure in advance in the system interrupt handler.

The profiling sample is built in the handler and inserted in the small kernel sample buffer. The task name needed for the system profiling is taken from the global TASK variable. If the kernel was interrupted an empty string is used as the task name. It is later used by the userspace profiler to identify the kernel samples.

### 6.4.8 Passing the Data to User Space Profiler

For reading the profiling samples the userspace profiler must first send the `SPROFILE_M_DATA_READ` message to the profiled task kernel answerbox. Sent call is not answered immediately. Kernel rather stores the call in the task structure and answers it later when it has enough profiling samples to be sent. It is also answered also when the profiled task ends to pass the very last samples.

If all threads of the profiled task are sleeping no new samples are generated. But in this case the profiler will not receive any message and the user might get worried whether the profiling session is still alive. The call is therefore sometimes answered from the system clock handler even if there are no new samples.

For the data reading itself we use the standard `IPC_M_DATA_READ` message method. Thank to it the kernel routine answers the message with the source address of the samples and the size of them to be sent to the profiler. The standard system message handling procedure copies the data themselves.

## 6.5 User Space Task Profiler Implementation

For profiling other tasks the user space profiler called **profile** was introduced. The profiler was designed as an interactive application. It continuously

receives the profiling data from the kernel and stores them in its buffer while waiting for user commands which are also immediately served.

Some of its common routines are implemented as a part of the standard system library (*libc*) as is common for the operating system libraries. These involve the profiling message handling in particular. However as these routines are especially important for the profiler we will refer to them here as they were part of the profiler itself.

### 6.5.1 Task Profiler Initialization

Task profiler offers two ways for establishing the profiling session with the profiled task:

- To start profiling an already running task just run profiler with the `-t` option followed by the selected task's id. Profiler establishes the connection to the task kernel answerbox and begins waiting for profiling samples from the kernel.
- For starting new task from scratch with attaching the profiler from the very beginning just run the profiler with the full path to the application binary. The profiler will load it, create the profiling connection and finally run it.

As the profiler knows the path to the application binary (either from the command line or from the sysinfo interface) it can locate it and load symbol table from the ELF image.

This operation might not succeed either because of ELF image was created without the symbol table or ELF image is missing at all. Therefore all symbol tables are also added to the file system by the HelenOS build system in a plain text files produced by standard Unix tool 'nm'. If loading the symbol tables from ELF image fails the attempt to load the symbol tables also from the text symbol files is done.

The symbol table in a plain text file produced by the 'nm' tool typically looks like the following example:

```
00001074 T __entry
000010c0 t profiling_abort
00001150 t cev_fibril
00001200 t data_fibril
```

```
000013f0 T main
00001ca0 t comparator
```

First column here specifies the value of the symbol, second the symbol type and the last the name of the symbol. As the symbol type is not important for profiling this column is discarded and all symbols are loaded to the symbol table. For more information see the manual page for the 'nm' tool.

If both symbol tables are not found the profiler declines to run. Without symbol table the profiler samples cannot be translated to the function names and therefore the profiling has no sense at all.

### 6.5.2 Profiler Standard Operation

After starting the profiling session and parsing the symbol tables few new fibrils are created to simplify the whole work:

- The *data* fibril which periodically sends the `IPC_M_DATA_READ` message to the kernel and concatenates the incoming chunks into one userspace buffer.
- User commands received from the console in the form of console events are monitored by the *cev* fibril.
- The original thread code waits on conditional variable for signal from these fibrils. When the signal is emitted it serves the request.

The key pressed by the user is detected and stored in the global variable by the cev fibril. Then it wakes up the main thread, which processes this command.

There were also some dangerous pitfalls on using fibrils. For example fibril stack is created as a part of a new fibril. But this stack has (nowadays) fixed size, if it overflows it can easily corrupt other structures on the heap. Because all fibril operations are done strictly in the userspace, kernel has no way how to detect such corruption. Corresponding task is not killed as is usual on thread stack overflow.

### 6.5.3 Symbol Parsing and Printing

After command 'p' is detected all samples received so far are assigned to the corresponding symbols in the symbol table and the whole table is printed.

The crucial point is that there can be very high count of samples. To gain enough samples during every system clock tick the profiling timer should be run in frequency of kilohertz at least. If every sample has just 4 bytes (as one standard 32-bit platform register) we get a data flow of several tens of kilobytes per second. After few minutes of profiling we have several hundred thousands of samples. Because all these samples must be parsed and assigned to the symbols the parsing can be quite time consuming.

Having a speed as a first demand in mind the first solution for sample parsing was based on making a bitmap of all possible sample values. The bitmap items contained counts of so far found samples of value equal to the item index in the map. The parsing of the samples was as fast as possible as every sample was read exactly once and the corresponding field value was just incremented by one.

The solution was even implemented however its drawbacks were very serious. The main problem is that we have to fit all possible Instruction Pointer register values in the given (limited) memory space. The IP register can point to anywhere in the virtual memory, thus the possible values can be any from the values allowed by the architecture byte range.

Some sort of hashing can be employed in order to fit more register values into the (much smaller) table. The most easy hash function would just drop high-order bits from the sample value as most of the user code begins at the lower address and is not larger than few megabytes.

However this operation would introduce some magic barrier in the processing which is definitely unwanted. That's why another solution was implemented for assigning samples to the binary symbols.

Symbols in the symbol table are now sorted according to their values and every sample is later parsed and assigned to the symbol using binary search. This solution is definitely slower than the bitmap as for every sample we need several accesses to the symbol table. However as the typical binary has a few thousands of symbols at most the binary search tree has about ten or eleven layers. The solution was used for parsing of one hundred thousand samples and the parsing still took less than one second.

## 6.6 System Profiler Implementation

The system profiler (`sprofile`) works in a very similar way as the task profiler. Also most of the code of the system and task profilers is very similar.

But because their usage is relatively different two separated applications were made through.

The first difference between the both is that the system profiler connects to its own kernel answerbox every time it is run. It therefore does not need any task id or pathname, in fact there are no options at all. After the connection is established the system profiler sends the `SPROFILE_M_SYSTEM_BEGIN` message to the kbox instead of the task method variant. The standard operation procedure is the same except the assignment of the samples to the symbols because there are more than one symbol table.

### 6.6.1 Assigning the Samples to Task Symbols

Task profiler loads the symbol table before it starts the actual profiling. Without the table the profiler fails because the profiling is worthless. But the system profiler cannot do so because it even does not know which tasks will be really running during the profiling session. It can obtain the list of currently running tasks from the sysinfo interface, but it is more than likely that other tasks will be loaded. This is probably the real cause why the user executed the profiler itself.

The kernel copies the name of the task to every sample which can be used to identify the symbol table. This is quite time consuming operation and it also remarkably enlarges the size of the data copied during answering the profiling data read call. However it is necessary because copying just the task id would not suffice to identify the task binary.

When parsing the samples the profiler first checks the task id. The corresponding symbol table from the cache is used if there is one already present. Otherwise the symbol table is loaded in the way just as it is done in the task profiler. Then it assigns the sample to the symbol table. The symbol table cache is represented by a simple linked list because usually there are no more than ten running tasks in the system.

## 6.7 Interpreting the Results

After the profiler is asked to print the results all symbols are sorted according to the count of their samples. Functions with most samples are echoed the first. These functions should consume also the most part of the application time, but this must not be necessary the rule. If there are several threads running in the task, these threads might have run concurrently in SMP

environment. Few threads can run on one CPU simultaneously whereas another thread can seize the whole CPU for itself. Therefore the latter can spent more CPU computing time and so produce more samples.

Example application and its profiling output is given in Appendix B.4. As we can see there are some unknown samples recorded if PMC interrupts were used as an event source. These samples do not belong to the profiled task, they should not be present in the output at all. They are caused by recording some random samples which is possible only due to the fact that interrupts caused by PMC are not maskable (NMI). The NMI interrupts hits some place in the kernel (where normal interrupts are prohibited) where the IP value is incorrectly copied. Unfortunately it is very hard to identify this place only from the incorrect IP value. Fortunately much less than one percent of the samples has such bad values. Therefore the profiling is still perfectly usable.

Standard interrupts (caused eg. by RTC chip) do not record similar damaged samples.

The example also shows that samples recorded during task profiling perfectly correspond to the supposed time spent in the functions. Unfortunately there are not yet any more reasonable applications running on HelenOS system which would deserve practical profiling.

Usually 'tetris' application is used for testing various HelenOS features. However for profiling this is not a good tester candidate because it's main thread is sleeping most of the time. It is either waiting for an input from a user or a clock signal to move game blocks.

## 6.8 Future Improvements

How can be the profiling interface improved? The implemented solution offers just basic functions which suffice for profiling both user tasks and the whole system. Ideas for improvements are the following:

- Profiling now recognizes a task as a base unit. More precise distinction based on threads can be made. Simple adding the thread id to the profiling sample structure is sufficient for later assignment of the samples to the individual threads.

- The CPU Performance counters are used in a very simple way which is enough for emitting the profiling events. The implementation should

be ported to different architectures at first, but also major expansion of the solution is advisable. The counters can serve for monitoring many different events such as counts of swapped pages, memory accesses and interrupt counts. As the measuring is done by the CPU itself it is by far the most precise solution. Some smart framework which would allow definition of all performance counters on other architectures would significantly simplify the goal.

- Even different things than the pure CPU usage can be profiled. For example the memory usage or network traffic statistics for specific task might be useful.

As we can see there are several ways how this work can be expanded. However the aim of this thesis was to build the basis for the task profiling which was accomplished. As with any other work there will always be ways how to make it better.

# Chapter 7

# Conclusion

## 7.1 Achievements

Among the main achievements of the thesis we consider the introduction of the original monitoring facility to the HelenOS system. Before the work on the thesis started there was no way how to obtain even the most basic data such as a list of running tasks. We created the whole monitoring subsystem together with corresponding userspace tools. This facility can monitor several other kernel subsystems. Also the measuring of other statistic information like the memory and the CPU usage per thread and per task was added. The HelenOS operating system gained tools needed for its later evolution towards the fully working operating system.

Next to the monitoring facility itself we developed original system and task profilers. In these days there is hardly any practical usage of these profilers because there are almost no applications which can be run on HelenOS and which are reasonable to profile. The application with the biggest practical usage by far is the 'tetris' game, which however does not load the system very extensively and as such it does not produce many profiling samples. For testing purposes some dummy load applications were made though. However the profilers will have to wait to bring their benefits until more user applications can run on HelenOS system.

We can declare that all goals of the thesis listed in the Section 1.1 were accomplished. All work was done in a tight cooperation with main HelenOS contributors so the monitoring interface was already accepted to the development mainline. An overview of all newly created userspace tools is given at Appendix A. Some example outputs are shown in Appendix B.

## 7.2 Future Evolution

Several ideas how to improve both the monitoring interface and the profiler support were already presented at the end of relevant chapters. It should be noted that to keep the income of this thesis persistent the monitoring interface must be kept up to date every time a new feature is presented to the HelenOS system.

In the short future a development of new user space tasks can be expected. Recently a file system and a networking support were introduced and both brought new server tasks to the system. A new framework for monitoring these 'system servers' should be made for better a understanding of the operations performed by the user tasks. This framework can either work in a way of receiving statistical messages from the servers or it can follow all IPC communication between the tasks and the servers to build an overview of the operations on its own.

There is also a space for expanding the support for the statistical profiling. The task binaries for system profiling can be stored in the central store. The Real Time clock driver should be introduced on all other architectures supported by the HelenOS system. Also the Performance Monitoring Counter support should be expanded to support other CPU models. It should be further extended to monitor all statistical information measured by the counters themselves. As almost any CPU type has a different set of the Counter registers some framework which would simplify the definition of the counters for another CPU models might be useful.

# Bibliography


[1] *Bazaar Version Control System*, http://bazaar.canonical.com/en/

[2] Meurs R.:
*Building Performance Measuring Tools for the Minix 3 Operating System*, Master thesis, VU Amsterdam, 2006

[3] *Dallas Semmiconductor DS12887 datasheet*,
datasheets.maxim-ic.com/en/ds/DS12885-DS12C887A.pdf

[4] Cantrill B. M., Shapiro M. W. and Leventhal A. H.:
*Dynamic Instrumentation of Production Systems*,
USENIX Annual Technical Conference, 2004

[5] Svoboda, J.:
*Dynamic Linker and Debugging/Tracing Interface for HelenOS*,
Master thesis, Charles University in Prague, 2008

[6] Walker R.: *Examining Load Avarage*, Linux Journal, 2006

[7] *HelenOS Project webpage*, www.helenos.org

[8] *HelenOS IPC for Dummies*, http://trac.helenos.org/trac.fcgi/wiki/IPC

[9] *HelenOS Design*, http://www.helenos.org/doc/design.pdf

[10] *HelenOS Documentation*, http://www.helenos.org/documentation

[11] Intel Corporation:
*Intel 64 and IA-32 Architectures Software Developer's Manual*,
http://www.intel.com/products/processor/manuals/

[12] *Linux Kernel Archives*, http://www.kernel.org/

[13] *Oprofile Statistic Profiler*, http://oprofile.sourceforge.net/news/

[14] *Oprofile Documentation*, http://oprofile.sourceforge.net/docs/

[15] Bandyopadhyay S.: *A Study on Performance Monitoring Counters in x86-Architecture*, Indian Statistical Institute

[16] Tanenbaum, A. S., Woodhull A. S.:
*Operating Systems: Design and implementation*,
Prentice Hall, 2006

[17] *Windows Fibers*,
http://msdn.microsoft.com/en-us/library/ms682661(VS.85).aspx

[18] *Windows Performance Counters Functions*,
http://msdn.microsoft.com/en-us/library/aa373078.aspx

# Appendix A

# Userspace tools overview

Following Appendix lists all tools which were created for user interaction with both monitoring and profiling interfaces. Also some example outputs are listed. All monitoring tools follow customs common to all UNIX system descendants. However thanks to the original environment of the HelenOS system they are still quite different.

## A.1 Tasks utility

The 'tasks' utility can be used for listing of all tasks running in the system. For every task its count of threads, amount of allocated virtual memory and its current user and system time are listed. Tasks tool can also print system load and the CPU usage.

Tasks utility is very similar to standard 'ps' utility found in all descendants of the original UNIX system.

```
Usage: tasks [OPTIONS]
Options:
    -t,--task task_id  List threads of the given task
    -a,--all           List all threads
    -l,--load          Print system load
    -c,--cpu           List CPUs
    -h,--help          Print this usage information
```

Without any options all tasks are listed

## A.2 Stats utility

'Stats' command is the simplest one. It shows just the system uptime and load. Its output is very similar to the output of common utility 'uptime'.

```
Usage: stats
```

## A.3 Top utility

Top is the most powerful utility from all mentioned here. It has abilities of all other utilities together and even more.

```
Usage: top
```

'Top' is run without any options. Every second it retrieves a new monitoring dataset. The difference between last two datasets is used to compute the percentage statistics. Therefore it takes a second to run the utility and retrieve the initial two datasets.

As is common the upper part of the output shows the statistics of the whole system. Here we find system time and date, uptime, sizes of free and allocated memory and CPU utilization overview.

All data in 'top' are updated every second, last two datasets are kept and all numbers are recomputed using differences between the two. There are few commands which can be used to switch 'top' to different modes showing different statistics:

- Command `i` shows IPC statistics of all messages sent between the tasks.
- Command `e` shows statistics of times spent by handling every single interrupt type.
- Command `t` switches top back to the initial state where task statistics are showed.

## A.4 Profile utility

The 'profile' tool can be used either on an already running task or it can start a new task from scratch. If a new task is requested full path to the task binary must be supplied.

```
Usage: profile [taskname|-t task_id]
Options:
    -t task_id  Connect to the already running task task_id
```

After the profiler has started several commands can be used:

- Command 'q' ends the profiler as well as the whole profiling session.
- Command 'a' pauses the current profiling.
- Command 'e' ends the profiling session without quitting the profiler.
- Command 'p' parses the samples received since the last printing and prints the profiling tables with counts of samples for each function.

Because the parsing and printing of the profiling samples can be very time consuming it should be used only when profiling itself is not running. Otherwise same samples might get lost.

## A.5 Sprofile utility

Sprofile utility is very similar to the profile tool. It differs only in its implementation as it profiles all system code. There are no options because there is no task to be selected.

```
Usage: sprofile
```

# Appendix B

# Example outputs

Following sections show example outputs from all tools which were created for the thesis.

## B.1 Tasks

Only first few lines of tasks utility are displayed.

```
/ # tasks
     ID  Threads       Mem      uTime      sTime Name
      1        5         0      0:  0      0:  0 kernel
      2        1    102400      0:513      1:142 init:ns
      4        1    131072      0:  6      0:  3 init:devmap
```

## B.2 Stats

```
/ # stats
00:01:34, up 0 days, 0 hours, 1 minutes, 34 seconds,
          load average: 0.85 0.18 0.03
```

## B.3 Top

Figure B.1 shows example output of running 'top' utility. The picture was taken on a system with one extensively running thread.

```
top - 00:02:27 up 0 days, 00:02:27, load average: 3.63 1.62 0.61
tasks: 31 total
threads: 35 total, 1 running, 1 ready, 33 sleeping, 0 lingering, 0 other
cpu0 (2206 MHz): busy ms: 32143, idle ms: 54384, idle: 100.00%, busy:  0.00%
cpu1 (2245 MHz): busy ms: 16581, idle ms: 69932, idle:  0.00%, busy: 100.00%
memory: 67059k total, 360448  unavail, 7962k used, 51908k free
```

| ID | Threads | Mem | %Mem | %uCycles | %kCycles | Name |
|---|---|---|---|---|---|---|
| 1 | 5 | 0 | 0.00% | 0.00% | 0.00% | kernel |
| 2 | 1 | 106496 | 0.74% | 25.58% | 41.12% | init:ns |
| 4 | 1 | 135168 | 0.93% | 0.00% | 0.00% | init:devmap |
| 5 | 1 | 5525k | 38.42% | 0.00% | 0.00% | init:rd |
| 6 | 1 | 626688 | 4.35% | 0.77% | 0.29% | init:vfs |
| 7 | 1 | 364544 | 2.53% | 0.00% | 0.00% | init:fat |
| 8 | 1 | 380928 | 2.64% | 0.00% | 0.00% | tmpfs |
| 9 | 1 | 372736 | 2.59% | 0.77% | 0.59% | devfs |
| 10 | 1 | 208896 | 1.45% | 0.00% | 0.00% | taskmon |
| 11 | 1 | 339968 | 2.36% | 0.00% | 0.00% | i8042 |
| 12 | 1 | 233472 | 1.62% | 0.00% | 0.00% | char_ms |
| 13 | 1 | 331776 | 2.30% | 6.58% | 0.00% | fb |
| 14 | 1 | 241664 | 1.68% | 0.00% | 0.00% | kbd |
| 15 | 1 | 905216 | 6.29% | 0.38% | 0.29% | console |
| 16 | 1 | 208896 | 1.45% | 0.00% | 0.00% | clip |
| 17 | 1 | 278528 | 1.93% | 0.00% | 0.00% | getterm |
| 18 | 1 | 278528 | 1.93% | 0.00% | 0.00% | getterm |

Figure B.1: Example output from 'top' utility

## B.4 Profile

Following snippet shows a source code of a task which was profiled using profile tool. The task was compiled without any optimizations[1] to keep calls to all functions.

```
void a(void) {
        uint64_t i;
        for (i = 0; i < MANY; ++i)
                ;
}

void b(void) {
        uint64_t i;
        for (i = 0; i < MANY; ++i)
                ;
}
```

[1] Option -O0 of the common gcc compiler

```
void main(int argc, char *argv[])
{
        a();
        a();
        b();
        return 0;
}
```

The results from the profiler are shown below. The profiling frequency was set to 2048Hz and the Performance Monitoring Counters were used as the source of the clock events. If Real Time Clock chip had been used as the source instead, the _heap symbol samples would not be recorded. This symbol was recorded by accident caused by usage of NMI interrupts.

```
/ # profile -t 34
Task profiler.
Loaded symbol table from ELF file app/tester
Profiling task 34 'tester' started
Commands: [Q]uit, p[A]use profiling, [E]nd profiling, [P]rint data
If you want the most accurate profiling data, do NOT press
'P' while profiling is running.
-------------------
Printing data:
Samples:    27845 total, 27495 user,      350 kernel
   Address  Samples Symbol
---------- -------- ------>
    0x10a8    13936 a
    0x112a     6761 f
    0x10e9     6758 b
    0xe000       40 _heap
```