Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Vojtěch Horký

Support for NUMA hardware in HelenOS

Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký
Study Program: Software Systems

2011

I would like to thank my supervisor, Martin Děcký, for the countless advices and the time spent on guiding me on the thesis. I am grateful that he introduced HelenOS to me and encouraged me to add basic NUMA support to it.

I would also like to thank other developers of HelenOS, namely Jakub Jermář and Jiří Svoboda, for their assistance on the mailing list and for creating HelenOS.

I would like to thank my family and my closest friends too – for their patience and continuous support.

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

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

Prague, August 2011 Vojtěch Horký

Název práce: Podpora NUMA hardwaru pro HelenOS
Autor: Vojtěch Horký
Katedra (ústav): Matematicko-fyzikální fakulta univerzity Karlovy
Vedoucí práce: Mgr. Martin Děcký
e-mail vedoucího: decky@d3s.mff.cuni.cz

Abstrakt: Cílem této diplomové práce je rozšířit operační systém HelenOS o podporu ccNUMA hardwaru.

Text práce obsahuje stručný úvod do problematiky ccNUMA, přehled vlastností NUMA hardwaru a přehled vlastností HelenOS spojených s touto problematikou (správa paměti, plánování atd.). Práce analyzuje rozhodnutí při návrhu implementace podpory pro NUMA počítače – zavádí reprezentace topologie do datových struktur jádra, zpřístupnění těchto informací do uživatelské prostoru, afinita vláken k procesorům a uzlům, politiky alokace paměti či vyvažování zátěže.

Práce též popisuje prototypovou implementaci podpory ccNUMA v HelenOsu na platformě AMD64 and stručné srovnání s podporou ccNUMA v jiných monolitických i mikrojaderných operačních systémech.
Klíčová slova: HelenOS, NUMA, jádro, operační systémy

Title: Support for NUMA hardware in HelenOS
Author: Vojtěch Horký
Department: Faculty of Mathematics and Physics, Charles University
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: decky@d3s.mff.cuni.cz

Abstract: The goal of this master thesis is to extend HelenOS operating system with the support for ccNUMA hardware.

The text of the thesis contains a brief introduction to ccNUMA hardware, an overview of NUMA features and relevant features of HelenOS (memory management, scheduling, etc.). The thesis analyses various design decisions of the implementation of NUMA support – introducing the hardware topology into the kernel data structures, propagating this information to user space, thread affinity to cores and nodes, memory allocation policies, load balancing, etc.

The thesis also contains a prototype implementation of ccNUMA support in HelenOS for the AMD64 platform and a brief evaluation and comparison with ccNUMA support in other monolithic and microkernel-based operating systems.
Keywords: HelenOS, NUMA, kernel, operating systems

Contents

Chapter 1

Introduction

The endless effort of software and hardware engineers is aimed at improving performance of computing machines. It started by adding secondary task on mainframe computers in the fifties and sixties and ends with cluster computing in the 21st century. When technology reached its speed limits, the processing units were duplicated and tasks were computed in parallel.

On one side of the parallelism efforts are huge symmetric multiprocessors (SMP) with hundreds of processing units. The other side of the spectrum is occupied by distributed systems running on thousands of simpler and cheaper machines. Somewhere in the middle (though closer to the SMP) are NUMA – non uniform memory access – machines that try to tackle problems of huge multiprocessors.

1.1 Goals

This thesis aims to analyse problems related to implementation of NUMA support in operating systems. The thesis describes main advantages and disadvantages of the NUMA architecture and analyses implementation and design decisions authors of NUMA-aware operating systems shall take into account.

To illustrate these decisions, prototype implementation of NUMA support for HelenOS [1] – a portable microkernel-based operating system – would be created.

The prototype implementation shall bring following improvements into HelenOS.

- Detection of ccNUMA hardware on AMD-64 platform.
- Provide API to allow user space programs use NUMA resources explicitly.

- Provide tools for end users allowing them modify (run-time) behaviour of programs without explicit support for NUMA.

1.2 Text organization

Below is an overview of the thesis structure and contents of individual chapters.

Chapter 2 introduces the concept of NUMA hardware, providing motivation for supporting it in HelenOS.

Chapter 3 introduces the HelenOS operating system. The chapter focuses on parts that are most relevant for the topic of this thesis.

Chapter 4 analyses the approaches and problems that need to be considered when extending operating system with support of NUMA hardware.

Chapter 5 describes the design and implementation decisions made during programming of the prototype.

Chapter 6 compares the prototype with other operating systems.

Chapter 7 provides results of several benchmarks run above the prototype implementation and reasoning of them.

Chapter 8 concludes the thesis.

Appendix A summarises benchmark results, B contains instructions for building the prototype implementation. Appendix C describes command line tools available in the prototype and short overview of the API.

Chapter 2

NUMA

This chapter serves as an introduction to NUMA hardware in general. It describes some terms used in the following text and also explains why NUMA was invented and what are the advantages of having explicit support for ccNUMA in an operating system.

The terms NUMA and ccNUMA refer to a computer architecture or a hardware setting of a computer. They are properly defined after short introduction describing problems of symmetric multiprocessors.

2.1 Reasoning behind NUMA

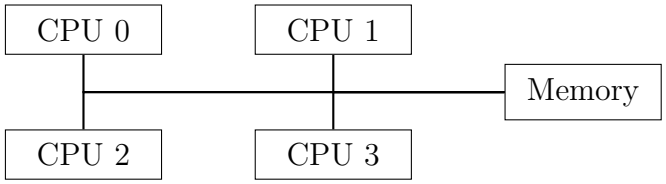
From the start of computer history in the 1940s, there was always effort to improve the performance of the computer machine. At the beginning, higher speed was acquired by improving the hardware technology. When the technology reached its (physical) limits, effort was shifted in other direction: instead of making the components faster, components were duplicated and the effort was invested into making the software capable of parallel processing. NUMA hardware is another approach to this problem.

The reason for NUMA is that even the parallelism has its limits.

Typically, a multiprocessor machine consists of several processors connected to the same memory using a shared bus. Schema of such computer is shown in Figure 2.1.

For a small number of processors, the requests for memory reads or writes could interleave without any performance degradation. That is due to the fact that a full cache line is always loaded – satisfying several sequential reads. Also various caches may decrease volume of the traffic on the bus – let us omit the problem of keeping cache coherency now and focus solely on situation when all processors request data from different parts of the memory.

Figure 2.1: Multiprocessor architecture with shared memory bus



However, when the number of processors is increased, the bus is not able to satisfy all requests and the processors are forced to wait. This effectively limits the maximum number of processors sharing the same memory bus.

The problem of a bus congestion can be postponed by introducing bigger caches but that may introduce problems related to cache coherency – more processors are competing over the same memory area. Another problem with bigger caches is economical because such caches are more expensive than ordinary memory. Other techniques can be used as well to improve the performance – for example prefetching that can be done in times when the traffic on the bus is lower. An extensive comparison of these techniques is described in [2].

Because of these factors, the NUMA architecture was introduced. Although NUMA does not solve all the problems (e. g. maintaining some kind of coherency – either on hardware or software level – is an inseparable problem of any multiprocessor architecture) it provides an effective means for achieving higher performance with reasonable costs.

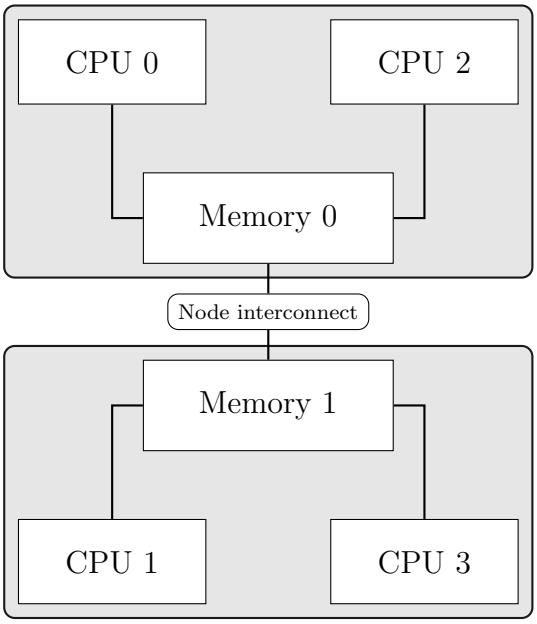
2.2 The NUMA architecture

The non-uniform memory access architecture is a multiprocessor computer architecture where individual processors have different access times to given memory area – i. e. some memory is further away for some processors than for others.

A schema of a simple NUMA machine is shown on Figure 2.2 (more complex schema involving other components can be seen in Figure 4.1). The machine has four processors and two memory banks. All processors can access both memory banks but with different speed or latency. For example, processor 0 can access memory 0 faster than memory 1.

Before moving on in describing the advantages and disadvantages of the NUMA architecture, some terms need to be defined and explained.

Figure 2.2: Schema of NUMA architecture



2.3 Terms

Node

Components with the same access time to a certain part of memory. The conventions for what is a node may vary but typically a single node consists of a memory bank and several processors connected to the same bus.

Optionally, a node may also encapsulate other hardware chips, e. g. a hard drive controller. In such case, the processors from other nodes may not be able to access the controller directly but only with assistance of the local processors. This may or may not be transparent to the software.

Local memory

Memory on the same node. Access to this memory is the fastest and optimally, a processor would access only the local memory.

Remote memory

Memory on a different node. Access to this memory is typically slower than access to the local memory.

NUMA factor
: Ratio between access time to local and remote memory. On PC platform, the ratio is usually normalized to factor 10 (i. e. NUMA factor for local memory is 10).

ccNUMA (cache-coherent NUMA)
: Special case of NUMA architecture where the hardware by itself takes care of cache coherency across all nodes. Optionally the 'non-uniformity' might be transparent to the software. For example, physical addresses may span across all nodes as a single continuous memory area.

UMA (Uniform memory access)
: 'Standard' symmetric multiprocessor architecture where the distance – either speed or latency – from a processor to memory is the same for all addresses and for all processors.

2.4 Topology

Figure 2.2 displays a very simple NUMA machine where each processor can (almost) directly access any memory. However, the machines might be more complex and mutual resource access might be more restricted. The restriction can be applied both on hardware and software level and in many cases makes any strict division of NUMA machines very complicated or even impossible.

For example, when the nodes are attached in a square fashion as can be seen in Figure 2.3, there are three situations how processor from node 0 might access memory on node 3.

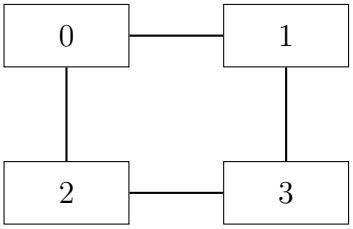
1. Directly – hardware ensures proper redirection of memory access instructions. This denotes a typical ccNUMA machine where the hardware provides high level of transparency.

2. Explicitly – software must issue some special instructions to access the remote memory.

3. The processor might not be able to access the memory at all.

Another option for node connection is in a tree-like fashion. That could be seen as a hardware equivalent of the 'divide and conquer' technique used for many parallel computation tasks.

Figure 2.3: Schema of 4-node machine



2.5 Advantages and disadvantages of the NUMA architecture

This section describes basic advantages and disadvantages of a NUMA architecture in comparison with a 'standard' UMA design.

The main advantage of the NUMA architecture is the possibility of achieving higher degree of parallelism. As stated at the beginning of this chapter, sharing the same bus can lead to bus congestion and performance degradation. With NUMA, this problem is bypassed by lowering the number of processors sharing the same bus and by increasing the number of separate memory banks.

The higher degree of parallelism can be utilized in many applications and itself could be treated as a reason important enough for introducing NUMA. Actually, all other attributes of the NUMA architecture could be viewed as disadvantages. However, these disadvantages can be minimised by using 'smart' software to such extent that the main advantage would prevail. This is discussed in Chapter 4.

First disadvantage of the NUMA architecture is that the operating system needs to recognise it. On PC platform with ccNUMA, the memory splitting is almost transparent and an operating system without any support for NUMA can use all resources. Different platforms and architectures may use different settings and may leave node discovery to the operating system. Actually, it might be sometimes difficult to say what is still NUMA and what is already a cluster and the definitions may vary. This text will focus solely on cache coherent NUMAs.

Other disadvantages arise when the software does not know that the hardware is actually a NUMA one (such situation is possible on PC platform).

Threads can be scheduled on processors belonging to a different node than the memory these threads access. Such problem can lead to worse performance than using simple UMA architecture because the processors might

compete over the same bus and also might require assistance of controllers from the remote node, thus slowing also access for all processors on the remote node.

Another disadvantage is that UMA thread load balancing does not respect incidence of the processors to the NUMA nodes. UMA load balancer then might migrate threads across nodes, thus completely removing effects of shared caches and memory locality.

Other closely related problem to load balancing is selection of initial processor a thread would run on. While UMA architecture scheduler can select the least loaded processor, NUMA aware scheduler might take also other attributes into account, e. g. scheduling on a node that has enough free memory.

The list of advantages and disadvantages of the NUMA architecture is definitely not complete. It is a mere overview of problems that need to be solved when designing a NUMA-aware operating system. These problems will be described in more detail in Chapter 4 together with approaches for their solving.

Chapter 3

HelenOS operating system

This chapter provides basic information about HelenOS operating system with emphasis on parts that are relevant to the scope of this thesis. This includes a description of HelenOS memory management, scheduling and kernel interface to user space.

3.1 System architecture

HelenOS is a microkernel, multiserver operating system running SPARTAN kernel. The design of the kernel is very minimalist as it tries to push as many things as possible from kernel to user space. That means that kernel in HelenOS needs to take care of the following tasks only.

Memory management
: This includes probing for existing memory, tracking of used and free parts of the physical memory and providing a virtual memory abstraction to the tasks (processes).

Task management
: This includes managing existing threads and tasks and their scheduling.

Inter process communication (IPC)
: Communication between pair of user space tasks[1].

Common abstraction level for user space applications
: HelenOS is able to run on several different hardware architectures and the last task of the kernel is to hide these hardware differences to allow

[1] We adhere to use the phrase inter process communication for consistency with terminology used in HelenOS, although the proper name shall be inter task communication due to the microkernel nature of HelenOS.

programming of the user space applications without prior knowledge of the target architecture.

Other functionality of the operating system – that is usually part of monolithic kernel – is provided via user space servers. That includes a file system service, network service, device drivers or a naming service for connecting to other services.

3.2 Kernel

This section will focus on parts of HelenOS kernel that are relevant to the scope of this text.

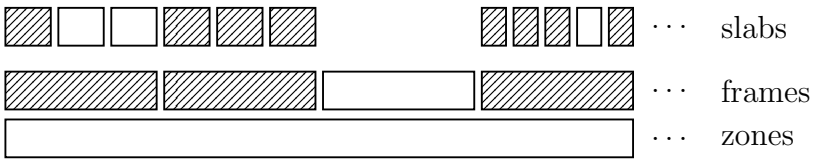
3.2.1 Memory management

To hide architectural differences of memory organization two layers of abstraction exist for memory management.

On the lowest level, memory is organized into *memory zones*, where each zone represents a continuous block of a physical memory. Buddy allocator controls allocations on each zone and also tracks free and used frames. The buddy allocator is able to satisfy requests for allocating 2^n physical frames.

However, working with the physical frames is hidden. User space tasks use a virtual memory abstraction while the kernel uses its own memory allocator – the slab allocator. Figure 3.1 displays hierarchy of memory management abstractions in HelenOS. They are described in more detail in the following sections.

Figure 3.1: Memory management structures hierarchy



Memory zones

A memory zone in HelenOS is the first layer of abstraction above the hardware memory banks. A single zone always represents a continuous block of homogeneous[2] physical memory, aligned to frame size.

[2]The word *homogeneous* refers to any attribute of the physical memory that might differ across the whole memory. This might include access rights (e. g. the ROM with BIOS or a firmware part) or a node boundary in a NUMA system.

Each zone contains information about its position in the physical address space, zone flags (e. g. read/write access), information about each frame in the zone and also service data for the buddy allocator.

Buddy allocator (frame allocator)

The buddy allocator [3] in HelenOS is used to retrieve physical frames in power of two counts. The allocator itself does not need to know anything about the actual memory and merely serves as an allocator that satisfies the requests by returning free frame number.

Kernel memory allocator

To avoid direct access to physical frame structures when the kernel needs to dynamically allocate memory, a special kernel memory allocator exists. HelenOS uses slab allocator that is briefly described in the following paragraphs.

The concept of a slab allocation was first introduced in [4]. HelenOS slab allocator is closely modelled after allocator in Solaris [5] and is described here in more detail because the prototype NUMA implementation introduced several changes to it.

The goal of the slab allocator is to prevent memory fragmentation and to speed up object allocation and deallocation. That is based on two observations. First, memory fragmentation is caused by sequencing allocation requests for different sizes after each other. Second, many objects are initialized and destroyed in a complicated manner but reusing objects is trivial.

The memory fragmentation is remedied by allocating objects of the same size from a memory area called slab. Typically, the slab is backed by several consecutive frames, their number is chosen to minimize overhead of unused memory.

The problem with frequent initialization and destruction of objects is remedied in the magazine layer. A magazine is an array of objects that are fully (or almost fully) initialized but not used. When the kernel needs such object, it is taken from the magazine. If a magazine is empty, the object is allocated from the slab and initialized. Object disposal means putting them back to 'their' magazine. When there are too many objects, they are fully deallocated and returned to the slab. The slab then returns them back to the frame allocator.

To maximize the positive effects of CPU caches, each CPU has its own pair of magazines from where it allocates the objects.

The structure of the whole allocator is best described by Figure 3 in [5].

Virtual memory and address spaces

The kernel is also responsible for creating a virtual memory illusion – separate address spaces for individual tasks. Virtual memory management is hierarchically higher than NUMA layer and is mentioned here only to define terms used in HelenOS.

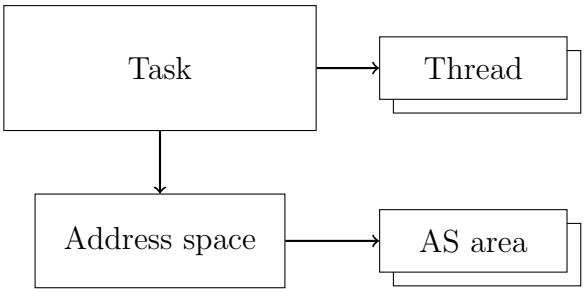
Address space in HelenOS is a wrapper that groups together *address space areas* – regions in the address space that share common configuration. Each address space area consists of one or more pages (backed by physical frames of the same size). An address space area can be shared by several tasks.

The actual contents of the area is obviously stored in the physical memory but HelenOS provides several *memory backends* with different capabilities (e. g. the ELF backend maps contents of an ELF image file into the memory while the anonymous backend is used for arbitrary data). All backends use the buddy allocator to allocate physical frames holding the actual data.

The address space area is the smallest entity user space task can operate with when managing its address space.

Figure 3.2 schematically displays relations between task, threads, address space and address space areas.

Figure 3.2: Task, threads, address space and address space areas



3.2.2 Threads, tasks & scheduling

HelenOS uses explicit tasks that are wrappers grouping together threads as the executing entities and a single address space providing a memory backend. The tasks are also endpoints for inter process communication – it is not possible to send a message to a certain thread but only to a certain task.

The scheduling uses round robin scheduler with several priorities, each thread is scheduled separately and independently regardless of a number of

threads in the containing task. There is a separate instance of the scheduler for each processor.

Load balancing of the thread queues on individual processors is performed by special kernel threads. Each such thread is *wired* (i. e. cannot be migrated) to a single processor and monitors queues on other processors and occasionally 'steals' a thread, migrating it to its own processor.

Unlike Unix systems, new tasks are not created with the `fork()`, `exec()` pair but by a call to a *loader*. Loader is a special program that is cloned by the kernel and that starts a new program. Task wishing to start a new one calls the loader via IPC and new task is started without any binding to the original one. The IPC is wrapped inside `task_spawnvf()` function. As a result, no explicit child – parent relationship is established among tasks. Also, the mentioned system calls that are vital for any Unix system are not present in HelenOS and their implementation would be rather problematic.

3.3 User space

The user space environment in HelenOS consists of client applications and system servers providing services of the operating system. Their description is beyond the scope of this text because support for NUMA was added at the kernel level. The only exception – how HelenOS passes information from kernel to user space – is described in the following section.

3.3.1 Passing information from kernel to user space

Although the aim of a microkernel based operating system is to move as much functionality to user space as possible, there will always be information that only kernel has access to.

The question is how to propagate such information to user space. In HelenOS, these information include list of running tasks, size of free memory or number of available processors. HelenOS uses so called *sysinfo* that provides a hierarchical way to convey information to user space. Each item is referenced by a path string and its value can be either statically assigned or computed dynamically at the time the value is actually read.

Chapter 4

Analysis

The idea of adding support for NUMA hardware into an operating system is quite easy to grasp – detect such hardware configuration and use this information to enhance the system performance. This chapter analyses in more detail what changes shall be done in the operating system to improve performance when running on NUMA hardware. It does not focus on a concrete implementation but rather provides set of problems each implementation must deal with.

The analysis begins with decision of what kind of operating system is being extended and continues with low level problems such as hardware detection onto more high level problems, such as load balancing. The analysis covers what information shall be propagated to user space programs. Discusses what would be handled by the operating system itself and what would be left on the end-user applications. The analysis also briefly touches benchmarking of a concrete implementation and requirements for the software running in user space.

4.1 Intended operating system usage

The first question before implementing NUMA hardware aware operating system is the intended usage of the system. Demands on a system targeted at running complex mathematical (parallel) computations are different than on system used as a database server and are different than requirements for a server running continuous integration tests.

Although the variety of tasks computers are used to solve is enormous, it is possible to select few representative samples and analyse requirements of an operating system for them.

For analysis of requirements for NUMA aware operating system the following representatives were chosen. First, a home computer. Next, a com-

puter running complex mathematical computations and system for multimedia applications. And finally server oriented applications. They include 'general purpose "multi" server' (typical settings for a smaller firm where the single machine serves as a web server, a database and an authentication server) and machines dedicated for a single task – web servers and database servers are typical examples. Somewhat special is also machine used for running large continuous integration tests and also a virtualization software running hundreds of emulated machines.

4.1.1 Home computer

Requirements of a home computer operating system are rather slim. Personal computer is today used typically for three different purposes. First, as a platform to run a web browser or some text processing tools. Next, as a multimedia platform for viewing and even editing videos and music. Last, as an entertainment tool used for playing computer games.

It is important to realise that actual efficiency of the system as a whole is usually not relevant here. The user requests low reaction latency and the focus must be shifted towards more subjective aspects.

Running a web browser or a text processor efficiently does not require NUMA architecture at all. The computer is used by a single person and computers today are far more powerful than it would be needed for running such applications, even on UMA machines. Moreover, most of the time the applications merely wait for some user input and running them on NUMA hardware would only increase the expenses for buying such machine.

Computer games are usually the only pieces of software that stress the home computer to its limits because game producers are trying hard to provide more realistic game, resulting in higher demand of a computer power. However, they are limited by the power of the computers that are currently on the market and could not go beyond their limits. The author is of opinion that computer games would not be targeted at NUMA machines unless their price would be low enough to motivate people invest in them.

The requirements for a computer as a multimedia platform are described later in 4.1.3.

4.1.2 System for complex mathematical computations

Under the term 'complex mathematical computations' are meant computations of mathematical problems that are not difficult in the principle but difficult because of the scope of the problem. For example, solving a system of linear equations is principally very simple but challenge arises when the system has thousands or millions of unknowns. Some of these problems

are easily parallelized and are solved using huge multiprocessors. For such problems NUMA might bring significant boost of performance.

However, for authors of NUMA aware operating system demands of such programs are the easiest to satisfy. The reason is that authors of these applications must understand very well the mathematics behind and are able to specify precisely the requirements on resource allocation – both processor and memory. Also these programs exist for a long time and their behaviour is well known. Usually, these programs explicitly query the system to describe its resources (or are written with fixed configuration for a concrete piece of hardware) and then explicitly divide the task onto the processors.

Extending these programs for NUMA architecture means identification of jobs that would profit by running on the same node. This could be done by the author or by the operating system itself. For the former, the operating system must supply means to get information about processor - node binding and also means to allocate memory from a given node. For the latter, the system must have means for monitoring access patterns.

Access pattern monitoring can be done both in hardware and software. In the hardware, with of help of memory breakpoints or performance counters. In an operating system by monitoring page faults (they mark the first access to a given page). But both of these approaches degrade the performance heavily. And not only by adding some instrumentation code. Hardware breakpoints and page faults almost always involve exception (interrupt) handling which is generally slow.

The question is whether running some automatic monitoring based on techniques described above would actually improve overall performance when the results would have been applied to the scheduling and allocation policies. Certainly not for short-lived tasks – and the operating system, unless being instructed by an operator, has no knowledge of the run length of the task. And even for longer tasks there is no guarantee that the 'guess' by the operating system – no matter how sophisticated – would be accurate.

However, having option to monitor these accesses would be helpful for development purposes. Especially for a software where the relations between individual subtasks are more complex.

4.1.3 Multimedia applications

Multimedia applications can be divided into two basic groups. In the first group are applications for viewing videos (and listening music) and in the second one are applications for their editing.

Multimedia players usually work in the following pattern – retrieve the encoded multimedia stream from a hard disk (or a network), decode it and display it on the screen. This pattern does not provide any means to improve

the performance by introducing NUMA hardware. For optimal performance, the graphics card and the disk shall be connected to the same node, thus rendering any other node useless. Furthermore, most of the processing is done on the graphics card and primary goal becomes fast transfer of the multimedia data from the storage to the card.

On the other hand, multimedia editing might profit from the NUMA architecture. For example, video recoding can be run in parallel with almost linear speed-up [6]. The bottle-neck then may become access to the storage media – e. g. the hard disk. Using more hard disks with stripped content and having them connected to different nodes might decrease the negative impact.

4.1.4 Dedicated servers

The base requirement for any server is scalability. An ideal server and operating system would double performance when the number of processors and size of the RAM are doubled. Such scalability is very difficult to achieve on UMA system (because of congestion of the shared memory bus) but NUMA system has higher probability of increasing the performance in linear fashion when more nodes are added.

For this, the software itself (i. e. the server) must be able to scale easily. That is usually achieved by creating multithreaded programs where each thread processes its own requests with minimal interaction with other threads. Interaction is needed only when querying shared queue of requests and the queue might be hierarchical to minimise the number of competing threads.

For the operating system, the requirements come in two flavours – memory allocation and thread migration due to load balancing.

Memory allocation

The server threads typically do not interact among themselves and thus it is the best policy to allocate all memory locally. That is the part of the problem with obvious solution. Not so obvious is what shall be done when there is no free memory on the node.

All of three possible responses from the operating system – 'out of memory', allocate on different node or swap to disk – might be correct as well as disastrous to the overall performance. The software might be smart enough to run its own balancing routines and returning 'no memory' could help it to rebalance. While allocating from somewhere else might corrupt its internal records about resource placement. On the other hand, for simpler software allocation elsewhere might be the solution that is 'good enough'.

Thus, the operating system should provide means to specify where memory shall be allocated from when there is no free memory available. The problem is further complicated by the fact that most operating systems actually allocate the memory when the program accesses it for the first time, thus making the option for returning 'no memory' almost unavailable. The operating system might still provide means to specify whether to swap to disk or fallback to other nodes, for example.

Thread migration between processors

Servers might distribute the requests among the threads by its own balancing rules and the best policy for the operating system might be to leave the load balancing alone – i. e. perform no balancing at all. However, the server is typically not the only process running and the operating system must maintain some load balancing checks. Because the emphasis is on the performance of the server software, the operating system might be more conservative and migrate threads only when the differences are extremely high. And migration between nodes can be turned-off completely.

4.1.5 Continuous integration servers

Operating system running software for continuous integration (CI) testing must be able to cope with bursty nature of load when the server spawns a lot of new processes in a quite short period of time. Some might be running for a very short period of time while others might execute for several hours. The variability of demands for such server could be very high.

The following is a quite common scenario for continuous integration testing. It clearly shows that operating system intended as a backend for such job must be able to handle several 'load patterns'.

The first step of CI is usually a checkout from some repository. That could be from a network storage and network cards usually use some kind of DMA for transfers and it is vital that such memory is physically close to the network card and to the processor that will process the downloaded data. And also close to the hard disk drive the files will be stored on.

Next step is compilation. For large projects this phase might include manipulation with thousands of files. In a NUMA system a hard drive might be connected to a single node only and processors from other nodes might access it only with a 'help' of the local processors. In a microkernel system, tasks servicing file system operations thus shall run on such node.

For the compilation, most of the source files has to be loaded into memory. Because the compilation can run in parallel, the initial placement of a new thread must take into account load of the processors and also amount of free memory. Compilation tasks run usually for a short time – compiling (without

linking) a single file is a quite short operation. Thus it might be better to find the node with most free memory first and then search for least loaded processor on that node only.

After compilation usually comes the phase of unit tests. By nature of unit test, this phase has very similar pattern as the compilation one.

After unit tests come tests on higher level (e. g. integration ones). They can run for longer time and the operating system might put more weight on balancing evenly the load on processors than on memory. Because these tasks run for longer time, the operating system may migrate pages together with the tasks.

Last phase is a clean-up which is somewhat similar to the checkout phase because it brings higher load on the disk drives.

From the described scenario it is clear that running CI requires transparency of NUMA hardware towards user space tasks. The programs run during CI typically do not gain better performance by executing on a NUMA machine. Also, the CI server software itself do not 'need' NUMA (after all, it is 'merely' a process launcher). The performance is expected to be improved by more effective parallel execution of the individual jobs.

The overall performance is thus determined by the ability of the operating system to balance the load evenly and effectively. However, the CI server may be able to provide hints about the requirements of the launched processes to help the operating system in resource allocation decisions. For example, a 'short lived' hint could be interpreted by the operating system as an order to minimise task migration among processors and to disable internode migration totally.

4.1.6 Virtualization software

The demands of virtualization software depends on kind of virtualization the software wants to provide. However, except for full virtualization when the virtualized system could access all components of the real machine, the demands on the operating system are the same as for a dedicated server.

It would be highly useful that threads belonging to the same virtual machine would execute on the same node to minimise inter-node communication.

4.2 Hardware detection

The hardware detection is a vital precondition for supporting any special piece of hardware at all. Each architecture has its own way to provide information about available hardware but for some hardware components merely having the drivers is not enough – they have to be available at very early stages of booting and may need to run under special conditions.

The biggest problem of NUMA detection is the need to break following cycle.

1. The information about the topology must be placed into some kernel structures.

2. Unless the configuration is static (i. e. known at compilation time), storing such information may require memory allocation.

3. In order to have working memory allocation, memory subsystem must be initialized.

4. Memory subsystem can be initialized after it knows the memory configuration of the machine.

Obviously, such cycle can be broken (e. g. by reading the topology information twice or by splitting memory initialization into more parts) but it is an issue that must be addressed and that might affect boot process of the operating system significantly.

But the hardware detection does not end by discovering placement of the memory. Typically, it just needs to be detected first. Detection of placement of other resources, e. g. processors or hard disk drives, can be postponed. For example, to have wrong information about binding between processors and nodes can lead to cache misses but thread can be later migrated to the correct one. But not knowing node boundaries can lead to allocation across them and once a kernel object is allocated, it may not be possible to move it. This problem may arise on ccNUMA machine where nodes are transparent and hardware provides illusion of a continuous physical address space.

4.3 Memory management

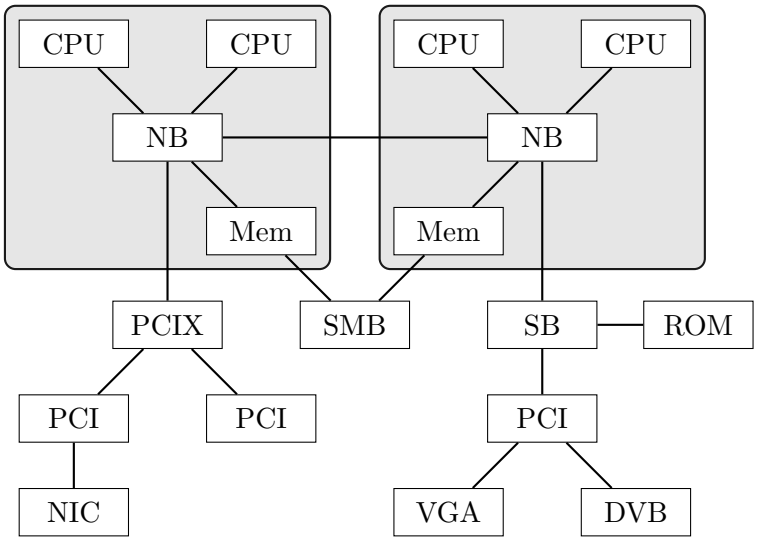
NUMA was introduced to speed up memory access and any operating system with ambitions to use NUMA hardware effectively must ensure that running tasks will use the memory effectively.

No matter whether the system lets user space tasks specify where to allocate memory or it decides it by itself (see discussion in 4.5) it is necessary that it has means how to specify from which node to allocate memory. Such simple requirement can be used in number of different cases and its presence (as opposite to having only operation 'allocate from whatever node where is enough memory') can significantly boost performance. And it is not only the common case to allow task allocate memory that is local for processor the task currently executes on. Other case is for example allocating memory

for DMA transfers. The hardware configuration may place a network card closer to one of the nodes.

Diagram in Figure 4.1 show possible settings for a two node system. The picture was greatly inspired by the 'Melody' machine shipped with the SimNow simulator [7]. It is clearly visible that operating system with ambitions to use the NUMA configuration properly should allow allocation of memory blocks for DMA for network card and for DVB-T card from different (close) nodes to minimise inter-node communication.

Figure 4.1: Example of a NUMA machine on AMD-64 architecture



The schema displays two node ccNUMA machine with two processors on each node. Network card and DVB card are attached via PCI to different nodes.

The frames surrounds components that forms a single node.

In a broader sense, every device connected to node's Northbridge can be considered part of that node.

CPU	Processor
Mem	Memory module
SMB	System management bus hub
NB	Northbridge controller
SB	Southbridge controller
PCIX	PCI-X controller
PCI	PCI bus
NIC	Networking card
DVB	DVB card
VGA	Graphics card
ROM	ROM chip with BIOS

The requirements of the software might be more complex. For example software running a complex computations in parallel. The usual pattern for

most parallel tasks is that the task is divided into workers at the beginning, each worker then run on its own and the task concludes by joining results from individual workers. The joined results are typically copied into single area and post-processed. Allocating this target area on a single node can lead to memory congestion. However, if the operating system offers means to specify that certain memory should be interleaved across more nodes, such congestion could be avoided with minimal efforts on the side of the application programmer.

There are more patterns of parallel computing where operating system intervention can help a lot. An application might mark certain data as read only (without the possibility to revert that flag) and ask operating system to duplicate them on all nodes. Such duplication can be done on the application level as well but shifting such actions to kernel enables other options. The operating system can perform the copying lazily (i. e. during page fault) or may use special instructions for the copying that are available only in privileged mode of the processor.

4.3.1 Allocation in kernel

Almost all kernel objects are allocated dynamically during the life of the operating system. Some of them exist since boot and are never deallocated.

But many of these objects are temporary and exist for a very short period of time. Such short-lived objects are then accessed only from a single thread. Thus it is very probable that the thread would not be migrated and that allocating strictly on a local node would not degrade (if not improve) the performance.

Other objects could be considered as mid-length with respect to their lifespan from allocation to deallocation. For example, an object representing a running thread. Such object would be typically accessed from the context of the represented thread and it makes good sense to allocate this object in the local memory. Although the thread may be migrated, the load balancer might minimise the inter-node migration thus minimising the need to migrate the object.

It is important to state problems involved in migrating a kernel object. First, kernel might use some kind of a slab allocation and migrating a whole page would lead to unnecessary migration of other objects. The obvious remedy is to migrate the single object only. But that means that the address of the object would change and the kernel may have no means to find all references to the original pointer. That renders single object migration unusable. And even migrating whole page with several objects is not without risks. Kernel may use identical mapping, thus effectively addressing with physical address, and such migration would change the address.

The conclusion is that kernel shall allocate objects locally because it would be optimal in most situations and shall not cause serious performance degradation in general.

4.4 Load balancing

The purpose of load balancing is to ensure an even load on all resources of the same kind available in the system in order to get the best performance. In a symmetric multiprocessing, the load balancing is degraded to balancing load on individual processors. These processors are usually identical and thus migration of a process from one processor to another one does not lead to any serious performance loss. Of course, there is the loss of the processor caches but due to shared caches on multicore processors the disadvantages are rather small in comparison with an idling processor.

On a NUMA system, there is the problem of symmetrical load balancing on each node but also the problem of balancing between nodes. And balancing between individual nodes has to take into account both processors and memory, thus making any 'optimal' solution even more complicated.

Load balancing for a UMA machine is typically done by balancing queues of ready threads on each processor. When the queue on one processor is longer than a queue on another one, the thread is taken from the longer queue and appended to the shorter one. Migrated thread is then marked as 'just migrated' to prevent starvation.

On a NUMA machine, the same technique can be used for balancing load within single node. But migrating a process to another node shall take into account how much memory the process allocated on the original node. It is possible that the process allocated only a small amount of memory and the overhead by accessing remote memory would be minimal. But it is also possible that the process is a long running one and allocated huge memory blocks and moving it to another node would cause performance degradation much higher than leaving one node with higher load.

An option for inter-node migration is to move the process memory as well. This possibility can improve the performance as well as degrade it. For a young process, the amount of memory is small and moving it would be a quick operation. Moreover, such moving can be done in a lazy manner. On the other hand, a process that is about to terminate would not benefit from memory migration. Such memory migration might involve copying of hundreds of pages of data, thus producing higher load on the inter node bus than accessing the memory remotely.

Principally, the operating system does not have any knowledge what would be the requirements of the running process and the best it can do

is to offer the administrator tools to balance the long running tasks manually. This could involve both thread and page migration.

Another factor influencing the migrating decision is the node distance. When the nodes are attached in a circular fashion (as shown for example in Figure 2.3), moving the memory to any other than immediately neighbour node can be completely out of question. Such moving may involve assistance of the nodes 'on the path', hurting the performance seriously.

The strategy might use NUMA factors to filter out too remote nodes. However, balancing the system locally only can create huge differences between more distant nodes. This difference can hurt the performance much more than an occasional migration between distant nodes only. Consider the simplest scenario: two node machine with high number of processors on each node. The policy to not migrate between nodes at all can lead to perfect load balance on both nodes – first node has running thread on every second processor while the second node has four threads per each processor. But it is unlikely that the computer is used effectively.

The thread load balancer can thus choose from several strategies. Below is a short overview of the basic ones.

No migration at all
: This approach could be very useful for short lived processes because it minimizes extra traffic on the system bus. The operating system must ensure that the initial placement is on the node with the lowest load.

Migration between processors of the same node only
: Possibility for running long term processes accessing memory a lot. Operator intervention might be needed for initial placement.

Inter-node migration
: Useful when accessing remote memory produces smaller overhead than memory copying.

Full inter-node migration – threads and memory
: Useful when it is cheaper to move the memory once to other node than to access a remote memory.

The strategy can be either system wide or each thread might use different setting that the load balancer respects. This setting might be automatic, manual or combination of both. User may manually specify the strategy for special tasks. The operating system may change the strategy automatically by observing the task 'behaviour'. For example, it might make the strategy more conservative if the task runs for a long time of if it allocates a lot of memory.

4.5 Transparency with respect to user space vs. explicit control

Once the operating system knows how to control and use the NUMA hardware by itself, it needs to allow user space tasks take the advantage of it as well. There are two opposite approaches to this problem. First, make the non-uniformity of memory access completely transparent for user space programs and use some heuristics to get 'optimal' usage of available resources. Second, leave all the decisions on user space programs and concentrate purely on executing their demands for using selected resources. Because both approaches to the problem might be useful under certain conditions, general purpose operating system must combine both to be able to satisfy various demands.

Making the non-uniformity transparent to the user space applications is almost a base requirement of any operating system. Even if the target application (e. g. mathematical computation) is built for NUMA machine explicitly, there would be always helper programs that do not need NUMA. Obviously it is possible that a shell would use in the code `numa_alloc_local()` instead of `malloc()` but the time investment for porting these applications would be too high. This hiding of the non-uniformity can be done by a system library but has to be present somehow. Exception is an application that runs on the bare hardware – i. e. the operating system is not needed or is part of the application – but that is a very special case.

Allowing user space programs query the NUMA configuration and explicitly allocate resources is vital for highly parallel computational problems where the author of the program knows in advance the requirements. These programs shall know as much as possible about the actual hardware configuration.

Thus the operating system must at least allow the programs to query the configuration – number of nodes, number of processors and binding between processors and nodes. The operating system then must give means to allocate memory on a given node and specify on which processors (or nodes) a thread may execute. The operating system can provide extra information – that could include placement of other components (hard disk drives, for example) or NUMA factors.

4.6 Inter process communication

Previous sections could be well applied to any operating system. Microkernel based operating systems has one more demand that is crucial for good performance – quick inter process communication.

Obviously, tasks running on the same node would have faster IPC because the transferred data are stored on node that is local to both processors. Grouping communicating tasks on the same node would probably speed up the IPC. Of course, this cannot be achieved system wide because it would effectively move all tasks to a single node. However, for tasks interchanging huge blocks of memory, such grouping might improve performance significantly.

For example, moving task with hard disk driver closer to the node to which is the disk connected sounds like a good idea. The data needs to be transferred to the node anyway but it is questionable whether wiring a certain task to a certain node wouldn't actually hurt the overall performance.

Nevertheless, the operating system must provide means for such actions because they might be worth doing under certain circumstances.

Another option might be to make system servers (such as virtual file system server) multithreaded and distribute the threads on all nodes. The client would then connect to the closest thread to minimize the inter node communication. Downside is higher number of threads and thus possibly longer queues on the processors.

4.7 Benchmarking

The final problem of each implementation of a NUMA aware operating system is to show that the support for such hardware actually improves the performance. There are many areas that could be benchmarked and compared: from overall performance, onto performance gains during run of a concrete program ending with synthetic measurements of small operations typical for given operating system.

As with any benchmarking it is very important to correctly interpret the results. Below is a short list of problems that could lead to misinterpretation and misunderstanding of the results.

1. Caching. Performance of programs working with small chunks of data at a time is more influenced by CPU caches than by anything else.

2. ccNUMA is more like a bigger UMA than workstation cluster. The difference between accessing local and remote memory can be extremely small and might be perceptible only when processing huge amounts of data.

3. NUMA adds extra resource layer. Thus the OS needs to iterate over one more layer and for very short benchmarks this might obscure the effects of local and remote memory.

4.8 Summary

This chapter listed several aspects author of NUMA aware operating system shall take into account to create a powerful system. Initial requirements on the operating system are guided by expected usage of the system that can vary much. However, the requirements can be applied to the following observations that roughly fit any target usage of the system.

Transparency
: Operating system shall provide means to hide the NUMA nature of the hardware for applications that do not want to take advantage of it.

Topology propagation
: Operating system shall provide detailed information about the hardware to allow applications use it in the best possible way.

Resource placement
: Operating system shall give applications possibility to specify where to allocate memory from and where to execute a running thread.

Tools
: Operating system shall provide tools to change placement of already allocated (existing) resources.

Chapter 5

Design and implementation

This chapter describes design decisions made during implementation of the prototype. We will see how requirements analysis from Chapter 4 was used for prototyping support for ccNUMA hardware into HelenOS.

5.1 Overview

The prototype implementation extended HelenOS in the following areas regarding support for NUMA.

- data structures
- hardware detection
- memory management
- affinity masks for resource allocation
- load balancing
- propagation of hardware configuration information to user space programs
- letting user space tasks control NUMA-related settings of their execution
- implementation of *libnuma* (library to utilize NUMA resources)

5.2 Data structures used for storing NUMA configuration

This section describes how information about NUMA configuration (e. g. information about memory sizes on individual nodes or number of processors on the nodes) is stored in the prototype implementation.

5.2.1 Memory

HelenOS uses zones (see Chapter 3) for storing information about available memory. When adding support for NUMA, there were two options: either extend current zone structure (`zone_t`) with node number specifying the node to which the zone belongs or add extra layer above zones.

The first option has the advantage of leaving most of the existing code untouched. The disadvantage of this approach is that for getting information about single NUMA node, all zones' information have to be read.

The second option represents the reality more clearly – each node can consist of several zones. For example, ROM part of the memory has its own zone that would typically belong to one of the nodes.

It was decided that the first option would be sufficient and also easier to implement. Running the prototype revealed that most of the nodes consists of a single zone only which partially justifies the abstraction shortcut. Usually, the only node with several zones is node zero to which belongs memory area with ROM and BIOS firmware.

5.2.2 Processors

Information about existing processors are stored in `cpu_t` structure and using the same reasons as with memory (see above), no extra layer for nodes was created.

Although the CPU numbers (indexes) might not be sequential per each node (actually, it seems that it is far more typical that the numbering goes across all nodes evenly), the disadvantage – for example when looking for 'siblings' – shall not create serious performance degradation that would justify creating more complex data structures.

Another reason for simplicity is that there are not many situations that would motivate a need for extra abstraction layer.

1. Reading the NUMA topology itself is not expected to be a critical part of any program to justify optimisation for.

2. Load-balancing needs to know which processors are closer but this information is not expected to change during the life of the operating

system. Thus it can be computed beforehand and in long term has no or very little performance impact.

5.3 Hardware detection

This section describes how the detection of a NUMA hardware was implemented in the prototype.

The prototype implementation is targeted at PC platform where hardware detection can use ACPI [8] – advanced configuration and power interface.

5.3.1 Reading topology of a NUMA machine

The actual retrieval of topology description from ACPI tables is straightforward and is mentioned here only for completeness.

The basic topology is described in a SRAT table – system (static) resource affinity table [9, p. 124]. Nodes' distances (i. e. the NUMA factor) are then read from a SLIT table – system locality distance information table [9, p. 127]. While the SRAT table is crucial for establishing abstraction of the actual hardware topology, the SLIT table is optional – it refines information retrieved from SRAT.

5.3.2 Creating NUMA aware memory zones

The creation of zones divided on node boundaries during boot phase of HelenOS is one of the most trickiest part of preparing a NUMA aware system.

The problem is that the operating system needs to read the topology (i. e. read the ACPI tables) prior creating memory zones because zone splitting might not be possible later. But that means that the memory management is not yet initialized (e. g. kernel `malloc()` is not yet available) and thus there is no possibility to store the information in some dynamically allocated memory. Also, since page mapping is initialized after zones initialization, some provisional mapping needs to be available because ACPI tables are part of the physical memory.

The early stages of SPARTAN boot are divided into logical subgroups that initialize individual subsystems. However, for proper initialization of the zones, ACPI initialization needed to be injected before memory management initialization (in current mainline it happens after initialization of the page mapping).

For portability issues, it was decided that SRAT table would be parsed into a static structure that would be later read during zone creation. The main reason was to reduce calling of architecture-specific functions in generic code. See Chapter 8 for discussion of possible improvements.

Once zones are created, the whole memory subsystem is initialized and NUMA detection is complete from the memory point of view.

5.3.3 Processor initialization

NUMA-related initialization of processors only assigns node numbers to them and this assigning is part of standard HelenOS procedure for symmetric multiprocessor initialization.

Once this is done, the hardware detection of a NUMA machine is considered complete.

5.4 Memory management

This section describes changes to HelenOS memory management in the prototype implementation. The changes include creating allocation policies for address space areas or improvements of the slab allocator.

5.4.1 Slab allocator

The design of the kernel memory allocator was explained in Chapter 3. The original allocator present in HelenOS expected a single node system. Section 4.3.1 explained that kernel allocator is often used for short-lived objects and memory locality is vital for good performance.

Because of that extra layer was added to the slab allocator. This layer represents individual nodes and allows allocation from local memory. The original allocator gets free memory frames by asking the memory backend for *any* memory. The new version specifies from which memory node the memory shall be. When such request fails, request for any memory is used as a fallback.

For benchmarking reasons, the NUMA aware allocator usage is a compile time option.

5.5 Affinity masks

Current mainline version of HelenOS does not offer any means to specify on which processors a certain task (or a thread) may execute. All threads can execute on all processors and allocate memory from any node. One of the points considered in Chapter 4 was allowing threads to specify which resources (namely processors and memory) they would use.

To achieve this, new attribute was added to each thread and each task. The attribute is a bitmask specifying on which processors the thread may execute. New tasks starts with a 'full' mask (i. e. all bits are set) and a newly created thread inherits the mask of the containing task. The mask held in

`task_t` exists only as a template for new threads and is not actively used because task is not an executing entity.

The template exists because user may want to bind task to certain node which implies that all of its threads shall be scheduled on that node only.

Similar attribute – bitmask – was also added to address spaces and address space areas. For them, it specifies from which nodes can be allocated frames backing the virtual memory. The policy of inheritance and initial value is the same as with threads' processors masks.

Address spaces and their areas has one more attribute that is related to resource allocation. This attribute is called allocation policy and determines behaviour of the page allocator when new frame is being allocated. This policy is described later, in Section 5.5.2.

Last thing that needs to be explained is effect of changing the mask of a container (i. e. task or address space) on the active entity (i. e. thread or address space area).

5.5.1 Behaviour for inherited masks

For tasks and threads, it is possible to specify whether the new container mask shall be copied into members or whether existing threads are left untouched. Actually, instead of direct copying, logical bit conjunction is used. Such approach fulfills two most common requirements – move the whole task to another processor (including all its threads) and spawn groups of threads with the same mask.

With address spaces areas, the situation is a little bit more complicated. Migrating a thread to a different processor is always possible unconditionally (omitting marginal case such as hardware failure of the processor). It may cause worse performance (due to uneven load) but the thread will eventually execute. On the other hand, migrating a whole address space area to a different node may not be possible at all (e. g. due to insufficient space at the target node). Because of that, changing mask of address space does not affect existing address space areas and such effect must be achieved by migrating areas explicitly.

5.5.2 Memory allocation policies

Idea of memory policies was borrowed from Linux [10] and specifies from which nodes are allocated physical frames backing a certain virtual memory area.

Available policies in the prototype implementation are

First touch

Allocate from node where current thread is executing (i. e. to which

node belongs the processor the thread is running on). Because pages are allocated after actual access to them (i. e. as a remedy to page fault), the 'first touch' policy does not imply that all backing frames are from the same node – consider migrated thread.

In Linux, this policy is called 'local'.

Interleaved
Allocate from all nodes in an interleaved manner. For example, when machine has two nodes, all even pages will be backed by memory on node 0 and all odd pages by memory on node 1.

Preferred
Allocate from specified node when possible, otherwise fallback to allocating from any node with sufficient memory.

The policy can be set for each address space area. By default, allocation policy is 'first touch'.

5.6 Load balancing and page migration

The prototype offers several different load balancing strategies that are selectable at compile time. They are based on strategies listed in 4.4.

The strategy can be found under the *CPU load balancing* option in the configuration menu. Following kinds of load-balancers are available.

None
No migration at all. Thread remains on the processor where it was placed during its creation.

Default
Use the original load balancer. This load balancer ignores NUMA nodes and expects a symmetric multiprocessor machine.

On node only
Do not migrate threads to different nodes.

When load balancer is used, it is possible to choose whether automatic page migration shall occur (*Page seizure on thread migration*).

The automatic page migration works in a lazy manner to avoid copying huge amounts of data at once. When the thread is about to be migrated, the load balancer prepares some of the pages for migration. It iterates over address space areas where the policy is 'first touch'. Then it marks pages backed by frames from the original node (i. e. the one from where the thread

is being migrated away) as not present. Pages from shared memory areas are not marked for migration at all.

When the thread accesses the missing mapping, the standard HelenOS page fault handler is executed. The generic page handler then calls a backend-specific page handler for given address space area. This one then decides what to do. If the page is accessed again from the original node, it only removes the 'not present' flag. Accessing from different node results in allocation of a new frame, copying of the data and replacing the page mapping. Then the flag is also dropped.

HelenOS uses *futexes* (fast user space mutexes) for mutual exclusion in user space programs. They are closely modeled after their Linux counterparts [11]. Futex implementation relies on using physical address for their identification in the kernel. Thus the physical address shall not change after first usage of the futex. To prevent migration of pages containing futexes, a special flag was added marking the frame as fixed. This flag is added when futex kernel code is executed and is checked by the backend page fault handler.

Preventing migration of futex-backing pages shall not mean performance degradation. If the task is multithreaded, it would be accessed possibly from more nodes and then no placement can be thought of as 'optimal'. For single threaded, but potentially multi-fibril[1] task, the futex usage do not enter kernel space – due to cooperative nature of fibrils – and thus the frame would not be marked as fixed at all.

5.7 Propagation of NUMA topology to user space

In order to allow tasks use computer resources effectively, they must learn about them. This section describes what approach was used in the prototype implementation to forward information about NUMA topology to user space tasks.

HelenOS has a simple mechanism for passing information from kernel to user space called *sysinfo* (see 3.3.1). This mechanism was used for passing information about NUMA topology from kernel to user space.

Although it would be possible to avoid asking kernel completely – by implementing another detection (as described in 5.3.1) in user space – it was decided that *sysinfo* would be used. The first reason is simplicity. Proper solution would require implementing standalone task that would read these

[1] HelenOS uses concept of fibrils for user space servers. Fibril is, in simple terms, a cooperatively scheduled thread living purely in user space. Fibrils are backed by kernel threads (in $m : n$ relation).

Table 5.1: NUMA-related sysinfo entries

sysinfo path	entry description (C type)
`system.numa.aware`	Whether HelenOS was compiled with NUMA support (`bool`).
`system.numa.max_nodes`	Maximum number of nodes supported by HelenOS (`sysarg_t`).
`system.numa.nodes`	Information about individual nodes (mostly memory sizes) (`stats_numa_node_t[]`).
`system.numa.distances`	Distance table between individual nodes (NUMA factors) (`uint8_t[][]`).
`system.cpus`	Added information about node to which the processor belongs (`stats_cpu_t`).

information and pass them to other tasks via IPC to minimize number of tasks with special privileges. The second reason is that kernel may not be able to reflect the actual hardware configuration properly or that (which is a more likely situation) the user space task would not be able to reflect running kernel configuration (e. g. placement of kernel image to one of the nodes etc.).

Although the idea of creating a special task for querying hardware configuration sounds promising, it is questionable whether such task would ever do such probing of NUMA configuration by itself at all. Such information is needed at *very* early stages of boot and needs to be implemented in the kernel. Thus, it is useless to create another implementation of the same functionality when the topology can be passed via *sysinfo*.

NUMA information is passed in `system.numa` subtree or was added to already existing entries. Table 5.1 summarizes information provided by kernel about NUMA configuration.

5.8 Letting user space tasks control resource placement

Once the kernel has ability to specify where a thread can execute and from which nodes it can allocate memory (see 5.5) it is important to allow control of such placement even from user space.

Although the microkernel nature of HelenOS could lead to implementing a special service controlling resource allocation, more straightforward approach was implemented. All functions for setting masks (both for allowed

processors and allowed memory nodes) and policies are backed by system calls that change directly the underlying kernel structures.

Because HelenOS does not offer a fine grained mechanism for granting permissions, such service would anyway only forward requests from clients to kernel, using the same set of system calls.

The number of system calls that were added is quite high but the author thinks that it is easier to implement and maintain more simple system calls than a single complex one.

The system calls are described in more detail in C.3.

5.9 Prototype implementation of *libnuma*

One of the points in Chapter 4 was that an operating system shall give the user space tasks ability to control resource usage. On NUMA system it means allowing tasks to decide where to allocate memory from and on which processors to execute. Such functions were implemented and are part of HelenOS implementation of base C library.

To show that the set of NUMA related functions is complete, it was decided that already existing library with this purpose would be ported to HelenOS. The library chosen was *libnuma* [12], originally intended for Linux based operating systems [13]. Reason for choosing *libnuma* was that it is an open source library and also because HelenOS is in the guise of a Unix operating system (at least in used terms and naming conventions).

After viewing sources of the original *libnuma*, it was decided that the HelenOS version would be written from scratch and only the API would remain the same where possible. The problem is that Linux *libnuma* reads the configuration by reading the `sysfs` entries while in HelenOS *sysinfo* is used. Also, Linux *libnuma* expects a Linux version of *libc* (e. g. because of the `sched_setaffinity()` function).

The actual implementation was then straightforward. Some of the library functions are merely wrappers of functions from *libc*, others need to convert between different formats expected by *libnuma* and HelenOS version of *libc*.

Although effort was focused on providing fully compatible API, it was not possible with some functions. Reason is that Linux does not distinguish much between threads and processes [14] while in HelenOS, a thread is the executing unit and the task is merely a not-executing wrapper. Thus, original *libnuma* operates purely on processes and expects a fork model of creating new ones. When threads are used, a special converter `gettid()` [15] from thread to process identification must be used.

Thus, several non-portable functions were added (always bearing the `_np` suffix) to bypass these problems while keeping the expected behavior of the 'standard' ones.

The library itself is in `/uspace/lib/numa`. The header file `numa.h` is then in `include/` subdirectory. Most important functions are described in C.2.

5.9.1 Porting `numactl`

Part of the *libnuma* distribution package is also a `numactl` utility. Its purpose is to launch new processes with changed settings of their NUMA policy [16]. According to the manual [17] for *libnuma* it is apparent that *libnuma* shall be used only in special cases.

> For setting a specific policy globally for all memory allocations in a process and its children it is easiest to start it with the numactl(8) utility. For more finegrained policy inside an application this library can be used.

The `numactl` is implemented using *libnuma* functions and the author thought that porting it would be matter of minutes needed to change some include paths and similar trivial problems. But the problem in porting this application was more fundamental. To understand it it is necessary to describe how this utility works.

`numactl` parses the command line arguments that specify the policy for the new process. That includes specification of nodes or CPUs where the process may execute and where from it can allocate memory. The utility can be used also for creating shared memory segments but this feature is not supported in HelenOS and will not be discussed here.

Once the parsing is done, the utility changes the policy for current process (i. e. for itself). After that, the new program is started using the `exec()` system call, thus replacing `numactl` with an image of the started program.

As was mentioned in 3.2.2, HelenOS does not have an `exec()` equivalent that is a key feature for seamless porting of the original `numactl`. A wrapper simulating the `exec()` would be easy to create (it would consist of `task_spawnv()` and `exit()` only) but such implementation would have no possibility to actually pass the NUMA policy to the new task.

The problem has to be solved at the loader level during creation of the new task. Two possible approaches for 'injection' of the NUMA policy to the new task were considered. The first option would mean adding a new method(s) to the loader through which it would be possible to specify the NUMA policy. The second one adds a hook to the `task_spawnvf()` that is called as soon as the structure for the new task is created in the kernel. This hook can then change the policies.

The prototype implementation uses the second approach because that one was easier to implement and is in some ways more versatile. The first one has definitely cleaner concept because the loader can apply the policies at proper times (e. g. apply the CPU mask during creation of the kernel structure instead of rewriting it later) and is also safer. It does not give the user a handle to a task that is not in fully consistent state.

The ported `numactl` does not have all the functionality provided by the original one but can be used without limitations to perform the following tasks.

1. Display hardware configuration.
2. Display current NUMA policy.
3. Start a new task with changed policy.
4. Change NUMA policy of already existing task. This feature is not available in the original utility.

The HelenOS version of the utility is not able to change settings of shared memory and differs in behavior for interleaved memory setting. HelenOS does not use a separate node mask for interleaved policy and thus changes to the 'interleave mask' are translated to the address space mask.

The sources of the utility can be found in `/uspace/app/numactl`.

Command line options are described in C.1.

Chapter 6

Comparison with other operating systems

In this chapter we will compare the prototype implementation with implementations used in other operating systems.

HelenOS is a microkernel and thus the most relevant comparison would be with another microkernel based operating system. However, the focus of the prototype implementation is in the kernel part of the system and there are many aspects that are comparable even with monolithic operating systems. For example, memory and task/process management are usually integral parts of a kernel, no matter whether a microkernel one or a monolithic one.

6.1 MINIX 3

MINIX is probably the best known microkernel operating system and the author feels that it needs to be mentioned although it has no support for NUMA at all. The true is that MINIX is targeted at *'Single-chip, small-RAM, low-power'* [18] and NUMA machines are definitely not of that kind.

6.2 GNU Hurd (Mach)

The GNU Hurd project wants to replace existing Unix kernels with a microkernel one while preserving existing APIs and ABIs. Project website [19] states that

> The GNU Hurd is the GNU project's replacement for the Unix kernel. It is a collection of servers that run on the Mach microkernel to implement file systems, network protocols, file access control, and other features that are implemented by the Unix kernel or similar kernels (such as Linux)

Hurd is based on Mach microkernel that can be briefly described as [20]:

> Mach is a first-generation microkernel.
>
> Mach's basic abstractions include virtual address spaces in the form of tasks, execution contexts in the form of threads, IPC, capabilities in the form of ports, and memory objects, which enable Mach's external pager mechanism.
>
> Controlling tasks, their virtual address space, threads, and other system objects in Mach is implemented by using ports, as opposed to other kernels' system call interface: almost all of the Mach API is implemented by sending messages to ports. Device drivers that reside in kernel space are controlled by ports, too.

The [21] (from 1992) states that

> The virtual memory system is designed for uniprocessors and shared memory multi-processors of a moderate number of processors. It has been ported to non-uniform access memory architectures, although optimal support for these architectures [...] is still being investigated.

However, the author was not able to find any reference to a NUMA architecture in the source codes of Hurd and Mach. That is probably because GNU/Hurd is able to run on IA-32 architecture only. On the other hand, the concept of external pager mechanism [22] allows easy addition of NUMA-aware memory managers.

As K42 (see 6.3), Mach has moved the memory management issues more into user space thus allowing more modular design.

No benchmarking comparison was done with HelenOS.

6.3 K42

K42 is a research microkernel operating system targeted at 64bit architecture. Citing from project website [23],

> The K42 group is developing a new high performance, open source, general-purpose research operating system kernel for cache-coherent multiprocessors. We are targeting next generation servers ranging from small-scale multiprocessors that we expect will become ubiquitous, to very large-scale non-symmetric multiprocessors that are becoming increasingly important in both commercial and technical environments. By designing the system from the start for multiprocessors, we achieve a high degree of spatial and temporal locality in code and data.

reveals that K42 is actually aimed at NUMA technology.

K42 is designed with high modularity in mind resulting in a very small kernel. HelenOS kernel is a small one and offers only the basic functionality (as mentioned in 3.1) but K42 goes several steps further and reduces kernel 'duties' even more. For example, thread scheduling and page fault handling is moved into user space [24]. The modularity allows replacing individual components very easily and the author thinks that adding support for ccNUMA machines for AMD-64 architecture would be rather simple. Currently, only PowerPC and MIPS are supported as NUMA platforms.

Memory management in K42 [25] is based on regions and file caching manager (FCM) . The FCM can be compared to the memory backends in HelenOS but K42 also offers a disk file backend. Adding such backend to HelenOS would not be a trivial job because HelenOS so far does not have any means to serve page fault from user space. Here the K42 offers easier implementation because all necessary routines are already in user space.

K42 wants to preserve Linux API and ABI and thus is more limited in implementation details than HelenOS that has more benevolent approach. Also K42 was designed for 64-bit architectures only while HelenOS is running on 32-bits as well.

HelenOS is being developed as a hobby kernel by enthusiasts and its pride is in clean design and high portability. K42 was developed as high-performance solution and the author believes that K42 would probably run better in benchmarks. The author does not have access to 64-bit NUMA machine supported by K42 to actually measure the difference (for example, the Levenshtein benchmark mentioned in 7.7 would be easily portable due to usage of Linux ABI & API).

6.4 Linux

Linux is a monolithic kernel supporting many different architectures and different kinds of deployment. One of them are large servers and Linux has many optimisations to provide as high performance as possible. These optimisations are usually part of architecture or even product specific code and HelenOS could not compete with Linux in terms of efficiency or range of supported hardware. However, Linux can be used for comparison as it can show which direction the development of HelenOS can proceed (or which approach is bad).

Support for NUMA technology exists in Linux since 2004 and is being constantly refined in current 2.6/3.x series [26].

From user point of view is NUMA support in HelenOS very similar to Linux. The most important tool for controlling NUMA – `numactl` – was

reimplemented in HelenOS and also the NUMA policy setting library – *libnuma* – is available in HelenOS. Actually, the needs of the mentioned library partially guided the development of several features in the prototype.

Support for CPU masks (*cpusets* in Linux) and memory area policies are almost the same in both systems. Both systems use dedicated system calls to change CPU masks or allocation policies. Linux checks permissions in kernel, HelenOS does not use any permission checking at all. In the future, HelenOS might be extended with some access control mechanism and system calls might be replaced with IPC (see 8.2).

Linux originally used slab allocator (as HelenOS) but today SLUB allocator [27] is used. SLUB uses different organization of the caches and can reduce their number by combining objects of similar sizes into a single cache. It also provides better performance. HelenOS might eventually use different allocator but for Linux efficient kernel allocator is more important than for microkernel HelenOS.

Chapter 7

Benchmarking

This chapter provides concrete results of benchmarks run on the prototype implementation and analysis of their results.

7.1 Benchmarks in HelenOS

Although HelenOS kernel could be considered complete (or at least stable enough to build applications above it), the user space part is still very spartan, especially in number of existing libraries. Practically it makes library or application porting very time consuming due to lack of the basic libraries. Even the base system library – *libc* – is not complete and moreover does not want to be POSIX compliant. This complicates porting even more.

Unfortunately, this means that it is very difficult to measure performance of real-world applications and compare it to other operating systems.

As a matter of fact, the author of the prototype implementation tried to port several user space memory allocators to HelenOS without much success. Either they depended on special functions that were not implemented in HelenOS so far or expected certain thread model.

The most limiting factor was time. HelenOS itself is mature enough to allow porting of many libraries but the time investment would be too high.

Because of these problems, most of the benchmarks are focused on comparing HelenOS with and without support for NUMA hardware. Because NUMA support is an optional kernel and user space feature, selectable at compile time, the comparison will be between same versions of HelenOS and thus shall give valid results. Also, as larger libraries for real problems are not available, most of the benchmarks described later are very synthetic ones focusing on a small part of the system.

7.2 Measured parts of the system

This section describes what features of the system were measured in the described benchmarks and why.

Chapter 4 described what parts of the system must be changed to support NUMA hardware. The performance is then affected the most by changes to memory allocators and load balancing entities.

HelenOS is a microkernel operating system and it makes sense to measure everything in user space. Actually, release build of HelenOS would offer no other choice. However, in debug build HelenOS provides a kernel console and thus it was possible to do some measuring in kernel directly. Nevertheless, except for synthetic benchmark of kernel slab allocator, all benchmarks were run from user space where they could be programmed more easily and with greater flexibility.

Measured parts of HelenOS included

- kernel slab allocator
- IPC between tasks
- wall clock time of a task spawning process
- computation of Levenshtein editing distance

Following sections describes each of the parts in more detail, together with analysis of the results. Tables in these sections show only the average values, complete results are part of Appendix A. The benchmarks were run on different configurations of HelenOS – e. g. with and without NUMA support or with different thread load balancer.

7.3 Biases

The benchmarks were designed to be as objective as possible. Also it was expected that the measuring would happen many times to provide precise results. Unfortunately, actual results showed several problems that the author was not able to resolve.

First, the benchmarking was run on a single machine (see A.1) because the author does not have access to any other ccNUMA machine. The development of the prototype was done in QEMU [28] emulator that is able to simulate NUMA. But QEMU emulator does not simulate different access times to memory and is completely useless for benchmarking purposes.

The second problem is even more severe. The actual results shown big deviations from their average. For example, the 'Levenshtein benchmark' (7.7)

running with 1 MB and 100 KB files took between 39.5 s and 43.7 s – the difference is about 10 % of the actual run time. That is too much to allow reliable conclusions, especially when the differences on GNU/Linux were less than 1 %.

The reason for this difference is not known to the author. Possible explanation could be that Linux is far more optimised – source codes of HelenOS do not contain extra code to boost performance while Linux ones are 'polluted' with architecture specific hacks to utilise power of the machine as much as possible. Another reason might be in a very simple approach towards power saving in HelenOS. In HelenOS, idle processor is halted while Linux uses frequency scaling to provide more fluent behavior. That means that the huge differences could be explained as scheduling anomalies.

Despite these problems actual results are presented. The conclusions shall not be taken as completely accurate, though.

The results use simple mean average and the complete tables in Appendix A display this average and the standard deviation only. Because is was unlikely that the differences would balance themselves out after performing hundreds of measurements, only about ten runs of each benchmark were executed. The actual results can be found on the attached CD, see B.4. Notice that for many benchmarks the number is actually lower because the author manually removed results that were totally out of bounds. For example when the measured time was twice as long as the average of the remaining values and thus would completely disrupt the computation. This selective removal was needed for both Linux and HelenOS.

7.4 Kernel slab allocator

The slab allocator used in HelenOS is described in Chapter 3 and changes to it are described in Chapter 5. There were done two benchmarks above the slab allocator. Both were run directly in kernel, from the kernel console.

The first benchmarks is concerned only with the result of the allocation and could be considered as a proof that access to a memory on the remote node is slower than access to a memory on the local node.

The second benchmark does the exact opposite. It completely ignores the result of the allocation and measures only the time needed to perform the allocation.

The benchmarks start with a 'calibration' phase when approximate number of loops is computed. The 'action part' of the benchmark is run repeatedly until the run time length exceeds given minimum. The counted number of loops is then used for all parts of the benchmark. This simplifies measur-

ing and development – it is possible to develop in an emulator and run on a real machine where the differences are in order of magnitude.

7.4.1 Measuring access time

The test spawns as many threads as there is processors and each thread is then wired to a single CPU. Before the measuring, each thread allocates several objects and their addresses are inserted into a global array. Once the objects are allocated, each thread selects some objects from the global array and will access them during the measured period of time.

The object selection is the decisive factor (we assume that the selection is symmetrical for all threads). If the thread takes back its own objects, they shall be on the local node and access to them shall be very quick. If the thread selects one object from every other thread, it will be accessing both local and remote objects and the time shall be longer.

For the measuring, all threads used symmetric settings. Assume the number of processors (and threads) is n, each thread will create n objects and store them into the global array, starting at $n \cdot i$ index (where i is the CPU index). Taking back own objects is trivial, taking single object from each other thread is also very simple (thread i takes every i-th object from n-long block).

Expected result from this test is that the NUMA-aware kernel shall have very similar access times on all CPUs in any symmetric object selection (assuming the hardware itself is symmetric and the system is not low on memory). The original kernel should show differences as some allocations that should be node-local were actually done on a remote node.

This test is available from kernel console as `test numaslab1`, results are in A.2.

7.4.2 Measuring allocation speed

Unlike the previous test, this test focuses solely on the speed of the allocator without accessing the allocated memory. The purpose of the test is to determine whether it is worth to use the NUMA-aware version of slab allocator.

Obviously, the NUMA version would be slower as there is extra layer in the allocator. However, if the slow down would be very small, it would be worth using the new version as the time lost would be outweighed by faster access to the memory.

This test is available from kernel console as `test slab3`, results are in A.2.

Table 7.1: Deviations in object access speed

Configuration	Local (512K)[a] [‰]	Remote (512K)[b] [‰]
Normal slab allocator	7.8	17.6
NUMA-aware slab allocator	0.7	9.9

[a]Object allocated by the same thread
[b]Object allocated by thread executing on different node

7.4.3 Conclusion for kernel slab allocator benchmark

As can be seen in tables in A.2, the NUMA-aware slab allocator leads to more evenly balanced access times from all processors (see standard deviation values) or summary Table 7.1. The test runs with several block sizes but it is apparent that the test is useless for very small blocks. The processor cache size on the benchmarking machine plays the major role in (repetitive) access speed. Thus only results for blocks of several kilobytes shall be taken seriously. The small difference even for bigger blocks only confirms caching effect.

As was expected, the modified slab is slower.

The question whether to prefer higher access speed or faster allocation routines is quite difficult to answer and more long-term benchmarks would probably be needed. The author thinks that HelenOS would rather profit from faster allocator than from faster access. The reason is that kernel objects are not accessed that often and that the NUMA factors are rather small. However, in monolithic systems, the situation might be reversed – objects representing, for example, open file nodes, would be typically accessed many times. In microkernel systems, such objects exist in user space where different allocators could be used.

7.5 IPC speed

Speed of IPC was measured using already existing applications in HelenOS. The `tester` application provides a simple IPC test that pings the naming service for several seconds.

This test was extended a little bit by migrating the pinging thread to all processors. Placement of the naming service was changed using `numactl`. For larger data blocks, the 'compilation' benchmark (7.6) can be used better.

It is worth mentioning that this benchmark revealed a problem in the implementation related to CPU masks and thread migration. The problem was a combination of starvation (the scheduler respected the CPU mask and migrated the thread to different processor prior running it) and a migration

Table 7.2: IPC speed benchmark summary

Configuration	Any[a] [rt/s]	Local[b] [rt/s]	Remote[c] [rt/s]
No NUMA support[d]	35717	–	–
NUMA support	32842	35052	25843

[a]Naming service can execute on any CPU
[b]Naming service was restricted to CPUs on the same node
[c]Naming service could not execute on the same node
[d]Without NUMA support, the `numactl` is not usable and there is currently no other way to change the affinity, thus results are not available.

to a halted processor (empty scheduler queues make a perfect choice for overloaded processor but also causes the idle processor to halt).

The problem was solved as could be seen the in Table 7.2. The original results has differences in order of magnitude, mostly because of the halted processor where the thread had to wait until next clock interrupt.

The benchmark contains a calibration part that first measures how many pings are possible within several seconds (rounded to thousands) and the actual measuring happens using this number. The results shown in the table are then normalized to round trips per second.

The results show that there is no clear relation between IPC speed and node positioning. Actual times varied a lot and the author thinks that the negative effect of halting and waking-up idle processor was not totally removed. Thus, the speed is more influenced by timer setting than by node distance. The IPC is measured more realistically as a side product in the simulated compilation benchmark, see next section.

7.6 Simulated compilation

Chapter 4 mentioned that one of typical usages of a server is running a continuous integration testing. The actual pattern differ depending on kind of software that is being tested but usually it involves a compilation stage that is somewhat similar for all projects. Thus, a benchmark simulating a compilation stage was created.

The benchmark is only a simulation because HelenOS does not contain a compiler to allow running a real compilation. To bypass this principal problem, a run of GCC was recorded by monitoring function calls and the record was used to create a new C program without any branching that performs roughly the same operations as the original compilation. Because not all li-

braries needed by GCC are available in HelenOS, only certain functions from the original run were used.

It is very important that these functions are vital for program performance and their performance is (or might be) affected by running on a NUMA machine. Obvious candidates are functions allocating and freeing memory (e. g. group of `malloc()`, `calloc()` and `free()`). Other candidates could be any functions that might involve any inter process communication. In an operating system with monolithic kernel, there are not many functions like that. For example, GCC running above Linux kernel does not communicate through IPC at all. But in a microkernel operating system, number of such functions can be very high.

However, compilation is 'merely' a reading from a file, parsing this input, converting it somehow and writing output to the file. Of course, the conversion is a very sophisticated process but does not involve any IPC. Thus, another group of functions worth recording are any functions involving operations on files (e. g. `open()` or `read()`).

Because compilation is not an interactive process, there are no other services the compilation task would communicate with.

The original idea was to record compilation of HelenOS in GNU/Linux. As a matter of fact, the author tried several methods to record the run – overwriting relevant functions using LD_PRELOAD, using `strace` and `ltrace` or using *SystemTap* – but none of them printed all the information needed or conversion of the log into a C program was too complicated. Finally, *dtrace* was used on OpenSolaris and compilation of several sources of GNU make utility was recorded.

The output from *dtrace* was translated into C program that was then compiled for HelenOS. The *dtrace* script also followed forked processes – this situation was translated to a thread creation. To simulate file accesses, random files were created in HelenOS disk image and actual filenames were translated to these ones (new, smaller, files were used to save up space). This leads to proper simulation of communication with file server in HelenOS.

The actual simulation consists of a special task – `hmake` – controlling launch of other ones. It can be thought of as a `make` command. This program has a statically prepared tree of dependencies, where each dependency is a launch of some other program.

The dependencies simulate the following scenario.

- Components of type A needs to be generated first and their generation can run in parallel.
- Components of type B and C requires A and also can run in parallel.

Table 7.3: Summary of simulated make benchmark

The table shows results when each component has 20 instances (thus 100 tasks were launched, some of them multithreaded).

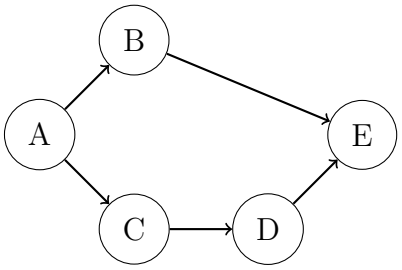
Load balancer	UMA[a] [s]	NUMA[b] [s]
Default	33.6	31.6
Node only	–	46.8

[a]Without NUMA support
[b]With NUMA support, including NUMA-aware slab allocator

- Components of type D requires C.
- Components of type E requires B and D.

The dependencies are shown in a graphical form in Figure 7.1. The vertices descriptions on the right side refer to commands issued during compilation of GNU make. Those who know how to compile the `make` utility from sources shall not have any problems identifying the actual commands. Notice that the graph does not follow the real scenario fully.

Figure 7.1: Graph of simulated compilation dependencies



A `gcc -c glob.c ...`
B `gcc -c implicit.c ...`
C `gcc -c fnmatch.c ...`
D `ar cru ...`
E `gcc -o make ...`

Generating each item means calling the simulated compilation. Number of components of each type can be specified at run-time. At run-time it is also possible to specify maximum number of concurrent jobs.

Table 7.3 summarizes the results. More detailed results are available in A.2.

The author expected that running this benchmark without any load balancer would provide better results than with some load balancer present. But the measuring showed that turning-off or restricting the load balancer prolonged the run by almost one half. That is probably due to the naive

implementation of placement of a newly created thread. The current implementation simply searches all processors and selects the least loaded one – processor with lowest number of ready threads. However, for short tasks that number is very volatile and as can be seen is useless as a pointer for least loaded processor. Even the 'node only' balancer failed to compare with the default one, probably for similar reasons.

The benchmark marks the run on NUMA-aware HelenOS as slightly faster. The difference is not in order of magnitude but shows that allocating local memory (instead of any) can have positive impact on the overall performance.

7.7 Computing Levenshtein distance

Levenshtein editing distance is [29]

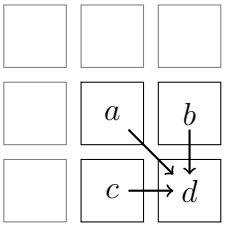
> [...] a metric for measuring the amount of difference between two sequences (i.e. an edit distance). The term edit distance is often used to refer specifically to Levenshtein distance.
>
> The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.
>
> [...]
>
> Computing the Levenshtein distance is based on the observation that if we reserve a matrix to hold the Levenshtein distances between all prefixes of the first string and all prefixes of the second, then we can compute the values in the matrix by flood filling the matrix, and thus find the distance between the two full strings as the last value computed.

But contrary to [29], even the simplest implementation can be easily parallelized with good results. The prototype implementation used for benchmarking scaled almost linearly with number of parallel sub-tasks used.

The data dependency in the matrix is the same for each cell, except for cells in first row and column – their values are statically assigned. Before computing a cell, cells that are above and to the left of it needs to be already computed (see Figure 7.2).

For parallel computation, there are several approaches. The first one is based on the fact that cells in a diagonal do not have any dependencies and can be computed in arbitrary order. The big disadvantage of this method is complex implementation that is error prone due to nontrivial index computing.

Figure 7.2: Data dependencies in matrix used for computing Levenshtein editing distance



$$d := \begin{cases} a & \text{same symbol} \\ \min(a, b, c) + 1 & \text{otherwise} \end{cases}$$

The second approach is based on a simple observation. In order to be able to (i. e. have data ready) compute second half of a certain row, it is necessary to have computed all previous rows and the first half of that row. With two threads running in parallel, each could compute its half of the matrix and the only limitation is that the thread computing the right half of the matrix must be delayed by one row. Obviously, this can be done recursively up to the number of processors available to allow maximum usage of the computing power. Figure 7.3 schematically summarizes the computation for a very small matrix with two worker threads.

The advantage of this approach is its simpler implementation. In this version of the algorithm, it is clear that each thread would work most of the time on local data – the row that is currently being computed. And a task-shared data boundary column. This observation allows to prepare a program that explicitly asks for local memory and that uses interleaved memory for shared data.

The program was implemented in three versions that differ in memory allocation requests.

malloc
: For all memory requests was used standard `malloc()`.

mmap
: All memory allocations were done using `mmap()` resembling POSIX interface. The purpose was to use pages not controlled by the standard `malloc`/`free` allocator.

Figure 7.3: Parallel computation of Levenshtein distance

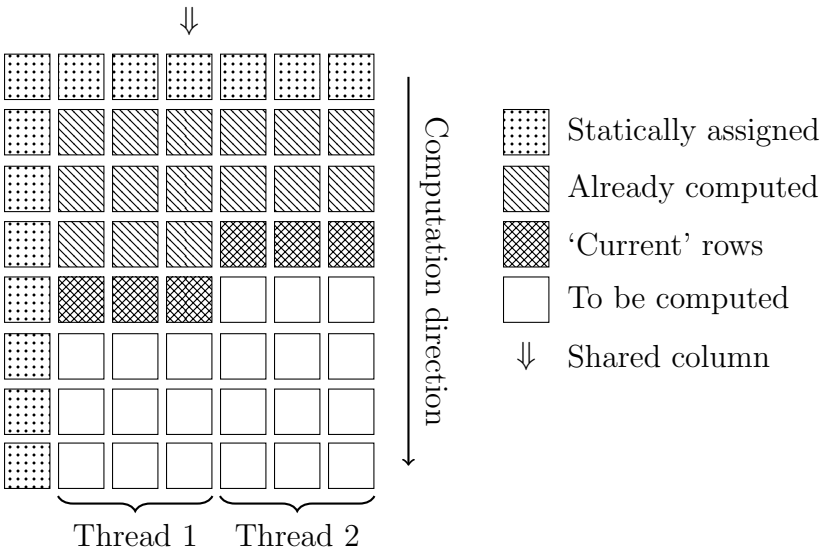


Table 7.4: Summary of Levenshtein benchmark (100k and 1M files)

Operating system	malloc [s]	mmap [s]	numa [s]
Linux 2.6.35 (Fedora 14)	37.4	37.6	39.7
HelenOS, NUMA	40.8	44.1	47.5
HelenOS, no NUMA	41.5	46.2	–

numa
: Explicit requests for local and interleaved memory were issued through *libnuma* API.

The prototype implementation was compiled for Linux and for HelenOS and measured with the same data on both Linux and HelenOS.

The program speed was not compared when run on a UMA machine because the author does not have access to a UMA machine with characteristics similar to the NUMA machine the program was run on.

Table 7.4 summarizes the results. More detailed results are available in A.2.

The table clearly shows that Linux is faster than HelenOS in this benchmark. The difference of more than 10% cannot be set aside as error in measuring. On the other hand, that is to be expected. Linux is being optimised for high performance by thousands of developers while HelenOS is

more targeted at academic environment and is developed by a small group of volunteers. It is necessary to point out that HelenOS with NUMA support was slightly faster than without it but still lagging behind Linux a lot.

What is surprising is that the simplest version of the program, using `malloc()` is the fastest. One would expect that using functions from *libnuma* would improve the performance significantly. But the opposite is true. The first reason might be wrong understanding of the problem itself – for example allocating the shared columns with interleaved policy. They are accessed twice (once for reading, once for writing) and such optimisation is probably not needed at all. Second reason is probably increased complexity of kernel structures. Each allocation creates new address space area, increasing number of objects to iterate when modifying address space (for example checking for overlaps when adding address space area).

Chapter 8

Conclusion

This chapter provides summary of achieved goals related with this thesis. It also describes areas of possible future work to improve existing implementation.

The aim of this text was to analyse requirements for operating systems that are supposed to run effectively on NUMA machines. Part of the effort was also invested into prototype implementation of NUMA support into HelenOS operating system.

8.1 Achievements, contribution to HelenOS

The thesis fulfilled all goals outlined in 1.1 and refined in 4.8.

The analysis covered wide spectrum of decisions any author shall take into account when adding support for NUMA hardware. Some of them were mentioned only briefly – for example, analysis of effective load balancing can be thought of as a standalone topic where load balancing on NUMA architecture is only a subtopic. But all the most important problems were listed and described in enough detail to allow implementation of the prototype.

The prototype dealt with the three most important problems related with NUMA support. The following list summarizes them.

1. Detect NUMA hardware.
2. Make NUMA hardware transparent by default.
3. Provide API for explicit work with NUMA.

The prototype is able to detect ccNUMA hardware on AMD-64 platform and store information about it in the kernel structures. The non uniformity of the machine is hidden by default from user space applications and no changes to them are needed. The base system library – *libc* – was extended

with functions for querying NUMA topology and with functions for explicit placement of used resources.

The prototype implementation goes beyond these basic requirements – see the following list.

1. NUMA-aware kernel memory allocator.
2. Simple, yet functional, support for different load balancing.
3. Implementation of existing API for working with NUMA machines – bringing *libnuma* to HelenOS.
4. Reimplementation of `numactl` – bringing well known tool from Linux operating system to HelenOS.
5. Implementation of several synthetic benchmarks.

Benchmark results do not offer plain answer whether the prototype implementation has better performance than HelenOS without any support for NUMA hardware. However, that was partially expected – it is a prototype implementation that was run on a single real machine with set of rather synthetic benchmarks. But it provides a base for further development that could include other architectures or support for not cache-coherent NUMAs.

The text analyses problems such extending could bring to the authors and offers possibilities for further improvements. HelenOS is an active project and improvements of the prototype and its inclusion into the mainline are not precluded.

8.2 Future work – prototype improvements

The prototype implementation can be improved in several ways that are beyond scope of this text or were not possible with version of HelenOS the prototype was built above.

1. Adding support for other platforms than Intel/AMD. This may include changes to the initialization order of the memory subsystem.
2. Improve the user space allocator (see discussion in 7.1), not only for NUMA-specific situations.
3. Introduce permission checking when changing allocation policies or affinity masks of another task. The permission checking could be done by introducing a new task that would be the only one having kernel capability to change the settings. Other tasks would contact it through IPC and it would verify the permissions with some security service.

4. Allow changing thread balancing policy at runtime or make it a thread attribute.

5. Port some parallel processing framework (such as OpenMP or TBB) to HelenOS.

6. In parallel with development of the prototype was added base support for memory reservations. Memory reservations shall ensure that task cannot ask for more memory than is currently available. The NUMA support shall extend reserves to be node-properties rather than system ones.

7. Literally days before submission, Portable C Compiler [30] was partially ported to HelenOS as a part of Google Summer Of Code project. Its presence would render the simulated compilation benchmark useless. It would be interesting to measure compilation of HelenOS inside HelenOS once PCC would have that capability.

Appendix A

Benchmark results

All benchmarks were run on the same two-node NUMA machine equipped with eight processors. Full hardware configuration is described below. See Section 7.3 for discussion about credence of the results. The oscillation of individual results invalidates any deeper conclusions and together with low number of measures makes other results than mean average and standard deviation useless.

A.1 Benchmarking machine

Machine identification is copied from GNU/Linux. NUMA identification learned with `numactl --hardware` is in Table A.1. Information about processors obtained from `/proc/cpuinfo` is in Table A.2 (all processors are of the same kind).

A.2 Actual results

Actual results of individual benchmarks can be seen in the following tables. The tables always shows mean averages together with standard deviation (number in braces). Original data are on attached CD (see B.4). See discussion in 7.3 for number of performed measurements.

Following list explains used configurations.

Table A.1: Machine used for benchmarking – node information

	Node 0	Node 1
CPU binding	0, 2, 4, 6	1, 3, 5, 7
Memory size	8190 MB	8192 MB
Distance to neighbour	20	20

Table A.2: Machine used for benchmarking – processor information

Vendor	AuthenticAMD
Model	Quad-Core AMD Opteron(tm) Processor 2356
Speed	2294.424 MHz
Cache size	512 KB
TLB size	1024 4 K pages

UMA
HelenOS without NUMA support and with default load balancer.

NUMA
HelenOS with NUMA support, with default load balancer.

NUMA, node LB
HelenOS with NUMA support, with node-only load balancer.

NUMA, no LB
HelenOS with NUMA support, without any load balancer.

NUMA, slab
HelenOS with NUMA support including NUMA-aware slab (kernel) allocator, with default load balancer.

Linux 2.6
Linux with 2.6.35 kernel, Fedora 14 distribution (with NUMA support).

Following tables contains results for individual benchmarks.

Kernel slab allocator
Tables A.3 and A.4.

IPC speed
Table A.5.

Levenshtein editing distance
Table A.6.

Simulated compilation
Table A.7.

Table A.3: Slab allocator speed (no object access)

Settings	NUMA	NUMA, slab	UMA
Batched	6717.17	5268.54	6609.91
Random	7514.12	6023.39	7529.94

(all results are in op/s)

Table A.4: Slab allocator – object access (read/write) speed

Settings	NUMA	NUMA, slab
16B buffer (× 1000), 'local'	4218.06 (1246.5)	5498.00 (765.6)
8KB buffer, 'local'	17279.83 (32.4)	17285.66 (30.7)
512KB buffer, 'local'	128.16 (1.0)	129.99 (0.1)
8KB buffer, 'remote'	16645.12 (48.2)	17274.04 (40.2)
512KB buffer, 'remote'	130.48 (2.3)	131.33 (1.3)

(all results are in access/s)

Table A.5: IPC speed

Settings	NUMA	UMA
Ping, any node	32842.50 (10044.6)	35717.98 (14704.0)
Ping, local node	35052.83 (11474.9)	–
Ping, remote node	25843.64 (416.9)	–
Buffer 4096B, any node	23245.74 (7855.9)	26993.01 (10600.7)
Buffer 4096B, local node	24395.56 (8341.2)	–
Buffer 4096B, remote node	19117.23 (697.6)	–

(all results are in rt/s)

Table A.6: Levenshtein editing distance

Settings	Linux 2.6	NUMA	UMA
1M and 100k, malloc	37413.88	40814.00	41503.33
	(70.1)	(866.7)	(1319.0)
1M and 100k, mmap	37553.66	44165.71	46226.66
	(53.3)	(3860.3)	(3133.2)
1M and 100k, libnuma	39667.55	47536.25	–
	(929.9)	(4092.2)	
100k and 100k, malloc	3661.44	6680.00	6933.75
	(4.2)	(1285.8)	(1075.1)
100k and 100k, mmap	3668.22	6902.00	6340.00
	(7.8)	(1335.1)	(1393.1)
100k and 100k, libnuma	3799.33	7661.00	–
	(78.9)	(1257.1)	

(all results are in ms)

Table A.7: Simulated compilation

Settings	NUMA	NUMA, node LB	NUMA, no LB	UMA
`hmake -j 8, -n 20`	31677.14	46813.33	47028.33	33615.00
	(995.9)	(272.7)	(314.1)	(1616.7)
`hmake -j 32 -n 20`	33545.71	46703.33	46876.66	33111.66
	(1302.6)	(43.1)	(40.3)	(1235.1)

(all results are in ms)

Appendix B

Contents of the CD, building the prototype

The attached CD contains sources of the prototype implementation, sources of the benchmark programs and scripts to compile them. The compilation is described later in this section. Below is a list of directories on the CD with their content.

prototype/
: Source codes of HelenOS with the prototype implementation of NUMA support and ISO images with built HelenOS for AMD-64. This directory also contains scripts for installing cross-compiler for actual building of HelenOS.

leven/
: Sources of the program for computing Levenshtein editing distance.

simake/
: Sources for simulated compilation.

bench/
: Raw benchmark results.

Following sections contains instructions for building. They assume that they are executed on Unix-like operating system such as GNU/Linux. For other operating systems, the actual commands may differ.

B.1 Building HelenOS

HelenOS cannot be built using the normal compiler shipped with the operating system but rather with a special cross-compiler. The cross-compiler can

be built automatically using the `toolchain.sh` script from `prototype/tools` directory. Parameter is target architecture, `amd64` is needed for the prototype. The cross-compiler is installed into `/usr/local/cross/amd64` and the script typically needs to be run with superuser privileges.

Once the cross-compiler is installed, one can proceed to building HelenOS. First it is necessary to configure HelenOS. The configuration allows user to select target architecture and other features. The configuration menu is launched by executing

```
make config
```

from `prototype/` directory.

The menu contains a lot of options but the correct configuration can be selected by loading so called 'Preconfigured defaults'. Choosing 'amd64' is the right choice for the prototype. Several lines from the top are positioned items relevant for NUMA build. The first is actual support for NUMA ('NUMA support') followed by more fine-grained options such as 'Kernel NUMA-aware slab allocator'. Near the top of the menu is also possible to select load balancer ('CPU load balancing'). Once the user is satisfied with the selection, he or she can confirm it by selecting 'Done'.

The actual compilation is started by running GNU make:

```
make
```

The compilation may take a while and a successful one is terminated by creation of `image.iso` file – a bootable ISO image with HelenOS operating system.

This ISO can be passed directly to QEMU emulator or burned onto a CD.

If a user wants to insert other files to the ISO that are not part of HelenOS build, these files has to be copied to `uspace/dist` directory prior compiling (or HelenOS has to be recompiled by issuing `make` again).

The sources are also available on-line at Launchpad repository

lp:~vojtech-horky/helenos/numa

The repository has also an on-line browser at

https://code.launchpad.net/~vojtech-horky/helenos/numa

B.2 Running HelenOS with QEMU

This section describes how to use QEMU to emulate a NUMA machine and how to boot the prototype in it. The obvious prerequisite is to have QEMU installed. KVM support is not needed.

Starting QEMU without NUMA emulation is very easy:

```
qemu-system-x86_64 -cdrom image.iso
```

Typically a user might want to specify more details about the machine. Memory size is controlled via `-m` option, number of processors via `-smp` option. Following command launches 8 processor machine with 2048 MB of memory (still a UMA machine):

```
qemu-system-x86_64 \
    -cdrom image.iso \
    -m 2048 -smp 8
```

NUMA is emulated by using `-numa` option on the command line when starting QEMU. The parameter is used to specify memory size and CPU binding. Simplest machine with two nodes (each 64 MB memory) and two CPUs can be started with command

```
qemu-system-x86_64 \
    -cdrom image.iso \
    -m 128 -smp 2 \
    -numa node,mem=64,cpus=0 \
    -numa node,mem=64,cpus=1
```

Larger NUMA machine with four nodes where each node has 512 MB memory and 4 processors can be started with the following command. It is necessary to mention that emulating such machine will exercise the host machine a lot.

```
qemu-system-x86_64 \
    -cdrom image.iso \
    -m 2048 -smp 16 \
    -numa node,mem=512,cpus=0-3 \
    -numa node,mem=512,cpus=4-7 \
    -numa node,mem=512,cpus=8-11 \
    -numa node,mem=512,cpus=12-15
```

B.3 Building other applications

Both `leven/` and `simake/` directories contains `Makefile`s for compiling of both programs. The default target builds all required versions at once.

The program for computing Levenshtein editing distance is shipped with sample data that were used for the benchmarking.

The simulated compilation is shipped as a trace from Solaris `dtrace` together with PERL script for conversion into a C code. Notice that the generated C program is large and can compile for unusually long time.

The dtrace scripts are in the `simake/` directory as well.

B.4 Benchmark results

The CD also contains source data for benchmarks presented in this text. The `bench/` directory contains several plain text files where each file belongs to certain HelenOS configuration.

The file contents is line oriented, each line starts with benchmark name and is followed by space-separated list of measured times.

Appendix C

Prototype implementation – tools & API

This chapter gives a brief overview of API and tools that are part of the prototype implementation. First, it describes `numactl` and gives a few examples of its usage. Follows a list of the most important functions from the *libnuma* library. Last section lists system calls that were added in the prototype implementation.

C.1 Using `numactl`

The `numactl` is a tool to change NUMA policies of existing tasks or to launch new tasks with explicitly set NUMA policies. GNU/Linux offers program of the same name and the HelenOS version of it tries to mimic the behavior as much as possible (see 5.9.1).

The tool is a command-line program controlled purely by switches. Below is a description of them. Several arguments accepts a mask (either of CPUs or nodes). The mask can be entered either as a list of individual members (i. e. numbers) (`0,2,5`) or as an interval (`1-4` or `!0-1` for inversion) or as a combination of both – e. g. `0,4-5,8`.

`--help`
: Displays short help and exits.

`--hardware`
: Displays hardware configuration of the machine and exits.

`--task`
: Specifies id of the task the new policies would affect.

  This option does not have equivalent in the GNU/Linux version.

`--show`
: Shows current NUMA policy (i. e. policies of the running `numactl` instance) or policy of a given task (when `--task` specified).

`--cpunodebind MASK`
: Binds the task to CPUs on given nodes only.

`--physcpunodebind MASK`
: Binds the task to given CPUs only.

`--membind MASK`
: Allows allocation from given nodes only.

`--bind MASK`
: Alias for `--cpunodebind` and `--membind`.

  This option is not available in the GNU/Linux version.

`--interleaved MASK`
: Sets allocation policy to interleaved on given nodes.

`--local`
: Sets allocation policy as local (first touch).

`--preferred NODE`
: Sets allocation to preferred from given node (single one).

When the task (`--task`) is not specified and none of `--help`, `--show` or `--hardware` options are present, any remaining arguments are treated as a path and arguments to a program that shall be launched with specified NUMA policies. See next section for examples.

C.1.1 Example usage

Learn information about the machine:

```
/ # numactl --hardware
Available nodes: 4
Node 0 CPUs: 0 1 2 3
Node 0 memory: 524236KiB (495272KiB free)
Node 1 CPUs: 4 5 6 7
Node 1 memory: 524288KiB (512660KiB free)
Node 2 CPUs: 8 9 10 11
Node 2 memory: 524288KiB (519164KiB free)
Node 3 CPUs: 12 13 14 15
```

```
Node 3 memory: 524276KiB (519152KiB free)
Distance table:
   |   0   1   2   3
---+----------------
 0 |  10  16  16  16
 1 |  16  10  16  16
 2 |  16  16  10  16
 3 |  16  16  16  10
```

Run the `tester` application (memory allocation test selected) on node 0 only. The output of `tester` is not shown.

```
/ # numactl --bind 0 tester malloc1
```

Display current policy. The first command displays the default policy because it is run inside shell that does not change policy of launched tasks. The second command launches another instance of `numactl` to actually display the new policy. Notice the explicit `--` to separate arguments of the launched task. The third example shows a rather useless policy where memory is preferably allocated from node 1 while threads shall execute on node 2.

```
/ # numactl --show
Allocation policy: first touch.
Task CPU mask: ################.
/ # numactl --bind 0 -- /app/numactl --show
Allocation policy: first touch.
Task CPU mask: ####------------.
/ # numactl \
    --preferred 1 --membind 1 --cpunodebind 2 \
    -- /app/numactl --show
Allocation policy: preferred (1).
Memory bind mask: -#--.
Task CPU mask: --------####----.
```

C.2 *libnuma* API

The API is described in more detail in the sources in form of Doxygen comments. Below is a short list of the most important functions only.

`numa_available`
: Initializes the library, checks that the OS is running on NUMA machine.

  This function must be called prior calls to any other function in *libnuma*.

`numa_max_node`
: Tells number of nodes in the system.

`numa_allocate_cpumask`
: Allocate bitmask for setting CPU affinity. The bitmask can be changed with group of `numa_bitmask_*` functions. The bitmask is then freed with `numa_bitmask_free`.

`numa_allocate_nodemask`
: Allocate bitmask for setting node affinity. Changing the created mask is possible with functions mentioned above.

`numa_alloc`
: Allocate page-aligned block of memory using default task policy. The implementation creates new address space area to allow different policy settings.
Allocated memory must be deallocated using `numa_free`.

`numa_alloc_local`
: Allocate memory from local node. Like `numa_alloc`, the returned block is page aligned.

`numa_alloc_interleaved`
: Allocate interleaved memory. Like `numa_alloc`, the returned block is page aligned.

`numa_run_on_node`
: Execute current thread only on given NUMA node (i. e. on any CPU from that node).

`numa_task_run_on_node_np`
: Non-portable function that sets CPU mask for the whole task. It is possible to specify whether existing threads shall be affected or not.

`numa_set_membind`
: Sets from which nodes the task may allocate memory from.

C.3 Added system calls

Below is a list of added or changed system calls. The description also specifies parameters, details can be found as Doxygen comments in the source code.

`SYS_TASK_SET_CPU_AFFINITY`
: Set CPU affinity mask of the the whole task. The parameters are task id, length of the bitmask and actual bitmask as an array of bytes. Last argument is a boolean flag whether the new mask shall be propagated to existing threads as well (see discussion in 5.7). See 5.8 for explanation why it is currently possible that any task can change CPU mask of any other task.

`SYS_TASK_GET_CPU_AFFINITY`
: Retrieve CPU affinity mask of the whole task. The parameters is also task id, length of the prepared mask and pointer to allocated array. Kernel returns actual length of the bitmask as an extra parameter. That is used in library wrapper for more comfortable usage of this system call (see below).

`SYS_THREAD_SET_CPU_AFFINITY`
: Set CPU affinity mask of a single thread. Parameters has similar meaning as with call for changing affinity of the whole task.

`SYS_THREAD_GET_CPU_AFFINITY`
: Get CPU affinity mask of a single thread. Counterpart of `SYS_TASK_GET_CPU_AFFINITY` on thread level.

`SYS_AS_SET_AFFINITY`
: Set node affinity mask and allocation policy of whole address space. The parameters are task id (user space has no other means to identify an address space) bitmask, allocation policy (see 5.5.2) and index of preferred node (when applicable).

`SYS_AS_GET_AFFINITY`
: Returns affinity mask and allocation policy of whole address space. Parameters are very similar to `SYS_AS_SET_AFFINITY` but here the kernel writes the information user space later reads. Extra argument is again actual length of the bitmask.

`SYS_AS_AREA_SET_AFFINITY`
: Set node affinity mask and allocation policy of a single address space area. Parameters are the same as for `SYS_AS_SET_AFFINITY`, address space area is specified with virtual address from within the area.

`SYS_AS_AREA_GET_AFFINITY`
: Get node affinity mask and allocation policy of a single address space area.

`SYS_AS_AREA_MIGRATE`
: Migrate address space area to different node. Parameters are virtual address from the area that is supposed to be migrated and two node numbers. The first number marks the original node, the second one the target node. Frames from other than the original node are not migrated.

`SYS_PAGE_FIND_MAPPING`
: Tells physical address of a frame backing a virtual page. This system call was extended to return node number as well.

`SYS_PAGE_MIGRATE`
: Migrate single page to a different node. Argument is virtual address of the page and target node.

All the calls returns success as the return value from the `__SYSCALL` macro. Standard HelenOS error codes are used, `EOK` meaning success (having value of zero).

Obviously, the system calls are not called directly by the programmer but rather using a library wrapper. These wrappers are part of the base C library and provide more high-level approach.

For example, system calls for reading affinity bitmask expects that the caller prepares memory block big enough to allow kernel store the bitmask in it. The library wrapper does this automatically and returns allocated memory to the user. The implementation uses simple feature of 'get affinity' system calls that passing `NULL` as address where to store the bitmask is interpreted as a valid call that does not copy any bitmask but merely sets number of bits needed. The library then allocated the memory and executes the system call once more.

Bibliography

[1] *HelenOS homepage* [on-line]. 2011-06-17 [cited 2011-07-15]. Available on-line: ⟨http://www.helenos.org⟩

[2] Aleksandar Milenković. *Achieving High Performance in Bus-Based Shared-Memory Multiprocessors*. University of Belgrade, July 2000. Also available on-line: ⟨http://www.ece.uah.edu/~milenka/docs/milenkovic_conc00.pdf⟩

[3] *Buddy memory allocation – Wikipedia, the free encyclopedia* [on-line]. 2011-05-22, revision 430428327 [cited 2011-07-15]. Available on-line: ⟨http://en.wikipedia.org/wiki/Buddy_memory_allocation⟩

[4] Jeff Bonwick. *The Slab Allocator: An Object-Caching Kernel Memory Allocator. USENIX Summer 1994 Technical Conference, Boston.* 1994-06-06 – 1994-06-10. Also available on-line: ⟨http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html⟩

[5] Jeff Bonwick, Jonathan Adams. *Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. Proceedings of the 2001 USENIX Annual Technical Conference, Boston.* 2001-06-25 – 2001-06-30. Also available on-line: ⟨http://www.usenix.org/event/usenix01/bonwick.html⟩

[6] A. Rodriguez, A. González, M. P. Malumbres. *Performance evaluation of parallel MPEG-4 video coding algorithms on clusters of workstations*. Technical University of Valencia, 2004-06-25. Also available on-line: ⟨http://ppl.stanford.edu/cs315a/pub/Main/CUDAEncoderProject/Performance_Evaluation_of_Parallel_MPEG-4_video_encoing_algorithms_on_clusters_of_workstations.pdf⟩

[7] Advanced Micro Devices, Inc.. *AMD SimNow^{TM} Simulator — AMD Developer Central* [on-line]. 2011 [cited 2011-07-15]. Available on-line: ⟨http://developer.amd.com/cpu/simnow/⟩

[8] *ACPI – Advanced Configuration and Power Interface* [on-line]. 2010-08-23 [cited 2011-07-15]. Available on-line: ⟨http://www.acpi.info⟩

[9] *Advanced Configuration and Power Interface Specification (revision 3.0).* Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation, 2004-09-02. Also available on-line: ⟨http://www.acpi.info/spec30.htm⟩

[10] *Linux Programmer's Manual : mbind – Set memory policy for a memory range* [man]. 2008-08-15. Available in GNU/Linux shell: ⟨man -s 2 mbind⟩

[11] Hubertus Franke, Rusty Russell, Matthew Kirkwood. *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. Proceedings of the Ottawa Linux Symposium.* 2002-06-26 – 2002-06-29. Also available on-line: ⟨http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf⟩

[12] Wickman Cliff, Lameter Christoph. *SGI - Developer Central Open Source — numactl and libnuma* [on-line]. [cited 2011-07-27]. Available on-line: ⟨http://oss.sgi.com/projects/libnuma/⟩

[13] Andi Kleen. *An NUMA API for Linux.* SUSE Labs, August 2004. Also available on-line: ⟨http://halobates.de/numaapi3.pdf⟩

[14] Jonathan de Boyne Pollard. *The known problems with threads on Linux* [on-line]. 1999 – 2003, 2010 [cited 2011-07-15]. Available on-line: ⟨http://homepage.ntlworld.com./jonathan.deboynepollard/FGA/linux-thread-problems.html⟩

[15] *Linux Programmer's Manual : gettid – Get thread identification* [man]. 2008-04-14. Available in GNU/Linux shell: ⟨man -s 2 gettid⟩

[16] *Linux Administrator's Manual : numactl – Control NUMA policy for processes or shared memory* [man]. March 2004. Available in GNU/Linux shell: ⟨man -s 8 numactl⟩

[17] *Linux Programmer's Manual : numa – NUMA policy library* [man]. December 2007. Available in GNU/Linux shell: ⟨man -s 3 numa⟩

[18] *The MINIX 3 Operating System* [on-line]. 2011-04-28 [cited 2011-07-15]. Available on-line: ⟨http://www.minix3.org/⟩

[19] *GNU Hurd* [on-line]. 2011-03-31 [cited 2011-07-15]. Available on-line: ⟨http://www.gnu.org/software/hurd/⟩

[20] *mach* [on-line]. 2010-12-21 [cited 2011-07-15]. Available on-line: ⟨http://www.gnu.org/software/hurd/microkernel/mach.html⟩

[21] Keith Loepere. *Mach 3 Kernel Principles.* Open Software Foundation, Carnegie Mellon University, 1992-07-15. Also available on-line: ⟨http://www.cs.cmu.edu/afs/cs/project/mach/public/doc/osf/kernel_principles.ps⟩

[22] Free Software Foundation. *external pager mechanism* [on-line]. 2011-02-17, revision bdd896e0b81cfb40c8d24a78f9022f6cd1ae5e8c [cited 2011-07-15]. Available on-line: ⟨http://www.gnu.org/software/hurd/microkernel/mach/external_pager_mechanism.html⟩

[23] IBM, Research. *K42* [on-line]. March 2006 [cited 2011-07-15]. Available on-line: ⟨http://www.research.ibm.com/K42/⟩

[24] Jonathan Appavoo, Marc Auslander, Maria Butrico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis. *K42: an Open-Source Linux-Compatible Scalable Operating System Kernel*. IBM T. J. Watson Research Center, 2005. Also available on-line: ⟨http://www.research.ibm.com/K42/papers/open-src.pdf⟩

[25] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis. *Memory Management in K42*. IBM T. J. Watson Research Center, August 2002. Also available on-line: ⟨http://www.research.ibm.com/K42/white-papers/MemoryMgmt.pdf⟩

[26] Christoph Lameter. *Local and Remote Memory: Memory in a Linux/NUMA System*. Silicon Graphics, Inc., 2006-06-20. Also available on-line: ⟨http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.7986&rep=rep1&type=pdf⟩

[27] Jeff Corbet. *The SLUB allocator* [on-line]. 2007-04-11 [cited 2011-07-15]. Available on-line: ⟨http://lwn.net/Articles/229984/⟩

[28] *About – QEMU* [on-line]. 2011-07-19, revision 1439 [cited 2011-07-20]. Available on-line: ⟨http://wiki.qemu.org/Main_Page⟩

[29] *Levenshtein distance – Wikipedia, the free encyclopedia* [on-line]. 2011-04-07, revision 422831210 [cited 2011-07-15]. Available on-line: ⟨http://en.wikipedia.org/wiki/Levenshtein_distance⟩

[30] *pcc – pcc portable c compiler* [on-line]. 2011-05-14, revision 1.25 [cited 2011-07-26]. Available on-line: ⟨http://pcc.ludd.ltu.se/⟩