Thread Dispatching in Barrelfish

EIRINI DELIKOURA

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2014

TRITA-ICT-EX-2014:37
Thread Dispatching in Barreelfish

EIRINI DELIKOURA

Degree project in Software Engineering of Distributed Systems Master’s program at KTH Information and Communication Technology
Supervisor: Georgios Varisteas
Examiner: Mats Brorsson

PERSONNUMMER 840918-T487
Abstract

Current computer systems are becoming more and more complex. Even commodity computers nowadays have multiple cores while heterogeneous systems are about to become the mainstream computer architecture. Parallelism, synchronization and scaling are thus becoming issues of grave importance that need to be addressed efficiently.

In environments like that, creating dedicated software and Operating Systems is becoming a difficulty for performance enhancement. Developing code for just a specific machine can prove to be both expensive and wasteful since technology advances with such speed that what is considered state-of-the-art today, becomes quickly obsolete. The Multikernel schema and its implementation, the Barrelfish OS, target a group of different architectures and environments even when these environments “co-exist” on the same system.

We present a different approach on loading and executing programs and using our new scheduling policy we handle tasks rather than threads, balancing work-load and developing a dynamic environment with respect to scaling and performance. Our goal is to use our findings in order to establish a more controlled way to use resources.
Contents

List of Figures

1 Motivation 1

I Introduction 3

2 Background 5
  2.1 The Multikernel 5
  2.1.1 Motivation 5
  2.1.2 Introduction 6
  2.2 Barrelfish 8
    2.2.1 Structure 8
    2.2.2 CPU Driver 9
    2.2.3 Monitor 9
    2.2.4 Processes 9
    2.2.5 Memory Management 12
    2.2.6 Inter-dispatcher communication 16
    2.2.7 The glue that holds together: Hake 17

II Our Approach 19

3 Problem Statement 21
  3.1 Problem Description 21
  3.1.1 Problem Statement 21

4 Implementation 23
  4.1 Our Approach 23
    4.1.1 Dispatcherd 23
    4.1.2 Domain: Aquarium 24
    4.1.3 RPC 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Analysis</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Benchmarks</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Conclusions &amp; Future Work</td>
<td>35</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Conclusions</td>
<td>35</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Future Work</td>
<td>35</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>37</td>
</tr>
</tbody>
</table>

**Bibliography** 37
List of Figures

2.1 Main modules and structure of Barrelfish ...................... 8
2.2 Capabilities types and valid retype operations on them ........ 14

4.1 Representation of the tree-like CSpace structure .............. 28
4.2 Dynamic Linking of Aquarium .................................. 29
4.3 Static Linking of Aquarium ...................................... 30
4.4 Fib execution time in clock cycles $10^9$ 32
4.5 Fib execution time in clock cycles $10^9$ 33
4.6 Fib execution time in clock cycles $10^9$ 33
4.7 FFT execution time in clock cycles $10^6$ 34
4.8 FFT execution time in clock cycles $10^6$ 34
4.9 FFT execution time in clock cycles $10^6$ 34
Chapter 1

Motivation

The motivation behind this thesis was to create a different approach on the way that binaries and applications are loaded, scheduled and executed on the Barrellfish Operating System. Moving away from user-land, where everything takes place in Barrellfish because of its minimal kernel and its extensive library, we aim to develop a more controlled and co-operative way that threads are spawned and run.

In Barrellfish users and programmers are free not only to implement their own scheduling policies but also to decide on the system resources that they are about to use. How many cores can I use? How many threads can I spawn? All these are decisions that the user can make on his own. While this can be freeing it can also lead to dangerous resources reservation and system exhaustion.

Another key issue that is addressed with this thesis is the “seclusion” that exists between cores. Since everything concerning system state is replicated in Barrellfish, one can imagine that communication is a little more complicated than just having cores share resources and information. We try to create a domain, an application, that execution is more evenly and dynamically distributed and cores can share tasks between them.

Additional, with this different approach, we aim to move to a more task-centric model, instead of the thread-centric already existing model. According to this, tasks that are to be executed can be shared between cores and are independent of the domain and the core they were initially spawned on.

Our thesis implements a new approach on the way a domain is executed since we focus on having multiple threads execute on the same virtual address space but on different cores. That means that our cores do not have to perform thread context switches that even though are lightweight, especially in Barrellfish, they still consume time and resources. Furthermore, within this implementation we have threads co-operating in executing different tasks and we actually explicit handle their course of execution.
Part I

Introduction
Chapter 2

Background

2.1 The Multikernel

2.1.1 Motivation

From instruction-level parallelism and super-scalar computers to Olukotun’s Hydra and IBM’s Power4, the silicon industry has been preoccupied with parallel computing for a long time. Attacking the problem on the hardware level has been the main approach, especially since it was because of hardware restrictions that systems’ design paradigm had to change.

As soon as it was established that clock rates of processors could not keep increasing with the same rate without major voltage and heating issues occurring, the multi-core architecture was introduced in order to continue enhancing systems’ performance. During the last decade, the multi-core architecture has not just gained grounds. Rather, it has become the default system design approach in order to overcome all the physical constraints deriving from increased system frequencies. Systems’ core counts keep increasing and we are now moving from multi-core to many-core systems. It can be argued that if it weren’t for the multi-core architecture, Moore’s Law would not still hold and systems’ performance would have more or less staled.

Power consumption of a system, whether we are talking about a supercomputer or a smart phone, is nowadays the concern that is driving the silicon industry towards new technologies with respect to energy efficiency. This is one of the reasons that besides the multi-core architecture, another design property that is currently drawing a lot of attention is heterogeneity. More and more semiconductor companies are releasing chips which carry different types of cores. If accommodating multiple copies of the same core on a single DIE has created challenges in the past, one can only imagine the new difficulties that arise from having to deal with multiple types of hardware units. Different ISAs, cache structures and interconnect are only some of those challenges. This is even more evident if we take into consideration the shift towards “dumping” computational workload to a system’s Graphics Processing Unit [12].
Thus, we can see that in order to be able to exploit all those new features and hardware characteristics, software also has to adapt. We can therefore understand the importance of Operating Systems evolving accordingly and adjusting to current trends and conditions. It is one thing having to develop software with respect to all those new principles and quite another having to implement a whole Operating System that will have to accommodate those changes. Such an effort is being made by Baumann et al. The resulting Operating System architecture, the Multikernel employs principles and techniques from the distributed systems field.

Barrelfish, is a new experimental Operating System, implementing the Multikernel architecture. Its modular nature, along with its lightweight kernel is targeted towards diverse, multi- or even many-core systems.

The motivation of this thesis is to present a new, alternative approach on the way that Barrelfish, loads, schedules and executes processes. Our approach is different in the sense that processes that have been so far free to exhaust system resources, are now controlled by the Operating System. Additionally, the workload is being handled co-operatively by all present cores.

2.1.2 Introduction

In the Operating Systems’ design space, whether we are considering the monolithic kernel, or its extreme opposite, the microkernel, we are referring to systems that assume shared, coherent memory, with one kernel instance running on all cores [4]. Although this architecture has been satisfactory so far, this does not apply any more.

The ever increasing core count, along with the heterogeneous and dynamic nature of systems has been the main initiative behind the Multikernel architecture which addresses all the resulting design difficulties by assuring that three crucial requirements are met [6]:

1. The structure of the Operating System is as hardware neutral as possible.
2. All inter-core communication is made explicit via message passing.
3. State is replicated instead of shared.

The heterogeneity of processors in a single die gives the illusion of our system being a distributed system. Thus, we understand that by making the Operating System as hardware neutral as possible, different types of processors on the same machine can be accommodated in a more efficient manner and achieve better performance and scalability. This “mix’n’mix” of various cores is the main reason for having one kernel image running per core in the Multikernel paradigm.

Shared resources (memory and a single shared bus) has been the mainstream way of inter-core communication for a while. As the number of processors per system kept increasing, came the realization that this mechanism is not optimal for many-core machines and can not scale efficiently. This observation lead to different types of hardware interconnects along with various cache coherence protocols [15].
2.1. THE MULTIKERNEL

Of course, when it comes to deciding upon which communication paradigm should be used in a system, many factors have to be taken into account. The architecture of the system is one of them and maybe, the most important one [8]. Moving away from the traditional shared memory schema and making use of interconnect topologies to implement explicit communication is all that more suitable for the Multikernel, especially if one considers that state, in this case, is replicated amidst cores instead of shared. The inherent “distributed” nature of the Multikernel only renders the choice of message passing over shared memory that more obvious.

Of course, all those design features do not come without a drawback. Consistency and memory management, once responsibilities entirely of the Operating System, are, in the Multikernel, issues that the developer has to take into consideration. Cache memory and cache coherence, developed as a crucial factor of optimized performance, are eliminated in the Multikernel because of its peculiar structure.
2.2 Barrelfish

2.2.1 Structure

Barrelfish is a novel Operating System, developed by ETH Zurich and Microsoft Research. The initiative was to accommodate multi and many-core systems and create an Operating System that scales along with the number of cores per system [1].

The main characteristic of Barrelfish that differentiates it from other Operating Systems is the fact that there is a separate kernel image running on each and every core of the system. This lightweight module running on privileged mode is called the CPU driver.

It is somewhat customary to turn to replication of state to as an optimization technique, on top of shared memory. The one-kernel-per-core attribute of Barrelfish though, makes it viable to choose replication as the main way to manage system’s state, ridding it of shared memory and structures. Not having to consider locks or multiple processors accessing a shared structure is only one of the few advantages that comes with this choice [14]. Additionally, by having a distinct CPU driver running on every core we have the ability to “tune” this kernel image according to the processor’s architecture [7].

Moving away from kernel-space and onto user-space, on top of the CPU driver, we find the monitor. Like with the CPU driver, we have one monitor per core. This way, Barrelfish resembles a networked system, with the distributed CPU drivers and monitors carrying out the core functions of a traditional Operating System. It is on top of the monitor module that drivers, daemon processes and all other user-level applications are running.

![Figure 2.1. Main modules and structure of Barrelfish](image-url)
2.2. BARRELFISH

2.2.2 CPU Driver

As mentioned before, the CPU driver is a minimal kernel instance running on a single core. It executes basic, core-local operations such as system calls, context switching, authorization and delivers hardware interrupts. Since there is one CPU driver per core, it is optimal to keep them simple. This is the reason for many functions, typically carried out by the kernel in other Operating Systems, being "delegated" to user-land in Barrelfish.

Even though Barrelfish is opting for hardware neutrality, the CPU driver is still the component of the Operating Systems that is most architectural dependent. Plus, they can be configured accordingly, not only for a specific processor family but also for a specific purpose. This results in kernels with different configurations even in a homogeneous system.

The scheduling that the CPU driver performs is minimal, since thread scheduling has been eliminated from its tasks. This comes as a consequence of the single threaded and non-preemptable nature of the CPU driver.

2.2.3 Monitor

The monitor is the process in Barrelfish that is mainly responsible for the whole system coordination and cooperation. We find one monitor per core, running on top of the respective CPU driver. This module runs executes on user-land. The set of monitors that can be found in one system are connected to each other, thus forming a network that is used to keep the system updated and coherent.

Apart from keeping the system consistent and the collection of cores up-to-date, the monitor is in charge with inter-core and inter-process communication. Since we are always talking about explicit message passing, it is the transportation of messages via specific interconnects, that the monitor is burdened with. Additionally, the monitor delivers system calls from other user-space processes to the kernel and is also greatly involved in the memory management of a system.

What we have described so far, renders obvious that many low-level tasks, traditionally part of the most privileged code are being carried out by the monitor which, for one, can impede system’s performance but also makes it more fault-tolerant.

2.2.4 Processes

Domains

When we talk about processes, in Barrelfish, we refer to them as domains. In that context, domain is a notion more abstract and general than the typical process. The actual scheduling unit of a domain’s thread of execution is the dispatcher. A domain is nothing more than a collection of dispatchers. To be more precise, for every domain there is a dispatcher unit running on every core that this specific process spans on.
CHAPTER 2. BACKGROUND

As far scheduling goes, we mentioned before that the CPU driver is responsible for some minimal scheduling. This refers only to scheduling the dispatchers. In Barrelfish there are 2 policies implemented for scheduling domains (and not threads). The classic Round-Robin scheduling policy and the Rate-Based Earliest Deadline.

On the other hand, the dispatcher is the unit handling the actual thread scheduling. That means that the dispatchers handle the local threads that are, of course, private to the given domain. In the case of a domain running on multiple cores, we have a set of dispatchers executing the actual program code, handling memory and threads locally along with any scheduling code and communicating with each other. This inter-dispatcher communication is inevitably assisted by the respective monitor units.

The dispatcher module resides in both kernel and user space. Part of it is accessible only to the CPU driver and part of it is shared between kernel and user space and part of it resides only in user space. Since the kernel is the one that schedules the dispatchers, it is responsible for restoring its state and resuming execution. This is the reason that the part of the dispatcher that is shared, contains fields like its entry points, its save areas, the time scheduled to execute and so on. On the other hand, the part of it that handles threads can is restricted to operate in user space since the CPU driver is not at all involved in that functionality.

All the functions that are necessary for the domains to run as intended and provide the developer with core functionality like creating a new thread or a new dispatcher, getting the domain’s current state or even sending a capability is part of the Barrelfish library and the respective source file is domain.h. Although the developer is free to implement any scheduling policies she wants and can interfere with thread/dispatcher management, the actual dispatcher unit is implemented by the Operating System and it is a built-in module. It is not something that is expected by the developer to be created.

There are two distinct modes that the dispatcher executes on. The disabled and the enabled mode.

When the dispatcher works on the disabled mode, it is handling scheduling issues, managing thread queues and Thread Control Blocks and running dispatcher code in general. On the other hand, when the dispatcher is running on the enabled mode, it is doing nothing more that actually executing the program code.

This modes of operation, besides distinguishing between the different operations that a dispatcher can perform at any given time, they also indicate where its state will be saved and restored from. Having said that, when a dispatcher is preempted while running program code, i.e. it is in enabled mode, its state is stored at the enabled_save_area. On the other hand, when the dispatcher that is preempted was running dispatcher code, its state is saved on the disabled_save_area.

Another thing that is resolved based on the execution mode of the dispatcher is the way that resumes execution after being preempted. When the dispatcher was last “seen” running dispatcher code, it is the kernel that is actually responsible of handling the context switch and restoring its state before the dispatcher resumes.
2.2. BARRELFISH

```c
/// Architectures generic kernel/user dispatcher
struct dispatcher_shared_generic {
    /// Disabled flag
    uint32_t disabled;

    lvaddr_t dispatcher_run;
    lvaddr_t dispatcher_lrpc;
    lvaddr_t dispatcher_pagefault;
    lvaddr_t dispatcher_pagefault_disabled;
    lvaddr_t dispatcher_trap;
    ...
};
```

In the listing above we see the dispatcher structure that is shared between the CPU driver and user-space and some of its fields, like the disabled flag and the various entry points that a dispatcher is up-called from, in order to resume execution.

In case the dispatcher was previously running in enabled mode, the kernel just enters it at an entry point, oblivious to its registers’ state, as it is the dispatcher itself that is further responsible for restoring its own state. Depending on the specific state of the dispatcher when it was preempted, there are five different entry points that a dispatcher can be up-called from:

1. **Run** entry point.
2. **PageFault** entry point.
3. **PageFault_Disabled** entry point.
4. **Trap** entry point.
5. **LMP** entry point.

**Threads**

The implementation of threads is Barrelish is similar to that of POSIX threads. In the default library we find the thread structure, implemented as `threads_priv.h` as well as a set of functions that are used to handle them which can be found in the file `threads.h`. All the basic functionality is supported like creating and running threads, joining and detaching them, adding and removing them from queues as well
as terminating them. Of course, threads are completely architecture independent.

```c
struct registers_x86_64 {
    uint64_t rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp,
    r8, r9, r10, r11, r12, r13, r14, r15, rip,
    eflags; uint16_t fs, gs;
};
```

In the listing above we see the set of the registers that represent the dispatcher’s state and are used in context switches and scheduling.

### 2.2.5 Memory Management

#### Capabilities

Another key aspect of Barrelfish, that makes it stand out of other Operating Systems is its memory management system. Memory handling in Barrelfish is following the SL4’s scheme, which means explicit memory management, based on capabilities. In essence capabilities are tokens, keys that grant access to various in-memory objects, structures even virtual address spaces.

Whether we are talking about a specific RAM region, a frame, a hardware page table or even a dispatcher, a domain has to hold the respective capability of that object in order to be able to read, write and act in general on that object. Since capabilities can represent various types of resources, for their actual implementation there are different structs with the appropriate fields. For instance, in the listing below, we see a capability structure representing a RAM region.
2.2. BARRELFISH

```c
/* RAM memory object */

address genpaddr base; /*Base address of untyped region */

size_bits uint8 bits; /*Address bits that untyped region bears */
```

The capability system in Barrelfish is \textit{typed}. This means that one of the possible operations on them is \textit{retype}. In addition to that, capabilities types are \textit{derived}. This follows that starting from a generic memory type capability, i.e. the \textit{PhysAddr}, all other types occur from retyping such a capability [13]. Besides changing its type, the retype operation can also be used to split a capability into multiple ones.

It should be mentioned that the actual capability structures are kept local to the kernel and they are accessible only by the CPU driver itself and the monitor. The rest of the domains are using but mere references to those structures, called \textit{caprefs} [10].

Although every domain is free to perform any action it wishes on a capref, i.e. create, copy, retype, etc., this action is executed on the original capability only \textbf{after} the CPU driver has validated and authenticated it. So, the CPU driver can keep track of which domain operates on which capability, thus all memory transactions are supervised and we have a robust memory management system.

Maintaining a robust and synchronized system when nothing is actually shared can prove to be difficult. That is why in Barrelfish there are extra precautions that need to be taken in both user and kernel space in order to guarantee that there is no illegal activity being performed or that our system is kept coherent. The majority of those are actually operations performed on capabilities. Some of those operations include copying capabilities, sending them to remote cores, retyping them or deleting them. Once again all that functionality is part of \textit{libarrelfish} and can be found in the source file \textit{capabilities.h}

**CSpace**

In Barrelfish, all capabilities are stored in specific “containers”, the \textit{CNodes}. CNodes are special regions of RAM that hold capabilities, intuitively, they are arrays of closely related capabilities. Once again, it is only the CPU driver and the monitor that have direct access on the CNodes, while all user-space domains, can actually access them using references, called \textit{cnoderefs}.

Since a CNode is a memory region, there is a type of capability corresponding to this kind of resource as well [5]. That means that a domain can have access to a CNode \textbf{if and only if} holds the capability for the specific CNode. This results in every domain having a unique tree-like structure, created out of all its CNodes
for storing all of its capabilities. This is the so called CSpace of a domain. So, besides their Virtual Address Space domains also have their own CSpace, holding the necessary capabilities for the resources this domain wishes to access.

In order to resolve the address of a capability in a given CSpace we have to traverse the whole tree-like structure, starting from the respective root. By using the base address of every CNode and the valid bits in the address, we can establish the right address. Of course capabilities are not placed in random positions in a CSpace. There is a specific region for holding capabilities representing page tables, a specific region for storing the dispatcher frame and so on.

In a domain’s CSpace, capabilities are not placed in random CNodes. There are specific types of CNodes that hold specific types of capabilities. For example, all the capabilities that are associated with dispatcher frames and endpoints are stored in the Task CNode whereas all the capabilities that have to do with page a domains’ page tables and virtual address space are kept in the Page CNode.

The capabilities of a domain are not only restricted with respect in which CNode they will be placed, but also concerning the specific slot of the CNode in question that they will be stored. These specifications render the form of the CSpace of every domain quite structured and specific. The fact is that this attribute of the domains makes it easier to handle, map and debug a CSpace.

Its somewhat “modular” nature also facilitates the initialization and spawning of a domain, since it is completed in several distinct steps. During every step a specific aspect of a domain is set. For example, first we set-up the CSpace, then the Virtual address space and then we initialize the dispatcher.
2.2. BARRELFISH

In the listing above we see part of the CSpace of the fish domain. Fish is the shell created for Barrelfish. The part of the CSPace that is shown represents the CNode Page that holds the capabilities for its paging tables. As we can see we have all the four levels of page tables that can be found in the Intel x86_64 architecture. Besides the slot for every capability, we see its c-address, its type and its base address.

**VSpace**

In Barrelfish there are 4 basic structures that form the Virtual Address Space of a domain. These are:

1. **VSpace.** The VSpace, as it is implied by each name, is a structure used to represent the Virtual Address Space of a domain. In essence it represents the collection of page-tables that form the VAS and are used to translate between the physical and virtual addresses. It is linked with exactly one Pmap object and multiple VRegion objects.

2. **VRegion.** The VRegion is collection of consecutive virtual addresses. It is implemented also as a struct that is associated with only one VSpace object, one Memory Object and a few other fields like its size, the next VRegion in line etc.

3. **Memory Object.** The Memory Object is used to manipulate blocks of memory. It can be linked with one or more VRegions and it is a structure used to maintain pointers to specific functions for manipulating the virtual memory areas that the given Memory Object is associated with.

4. **Pmap.** The Pmap module is the one that actually performs the mappings, the lookups, the unmappings and all the operations that should be performed on page-tables. It is the sole module of the whole virtual space implementation.

```plaintext
fish .0: slot 1 caddr 0x8000000 (5 bits) is a CNode cap (bits 7, rights mask 0x1f)
fish .0: slot 0 caddr 0x8000000 (12 bits) is a x86_64 PML4 at 0x18d8c000
fish .0: slot 1 caddr 0x8100000 (12 bits) is a x86_64 PDPT at 0x18d98000
fish .0: slot 2 caddr 0x8200000 (12 bits) is a x86_64 Page directory at 0x18d99000
fish .0: slot 3 caddr 0x8300000 (12 bits) is a x86_64 Page table at 0x18d9a000
fish .0: slot 4 caddr 0x8400000 (12 bits) is a x86_64 Page table at 0x18d9c000
```
that is actually architecture specific. We should mention again that there can be no more than one Pmap object associated with one VSpace.

One other structure that is related to a virtual address space of a domain is the capability type VNode. As such, it grants access to specific types of page-tables and page-table directories. For example for the x86_64 implementation, there are 4 types of VNodes, one representing every level of page table directories, according to the specific of the Intel architecture. So we have:

1. Capability: VNode_x86_64_pml4 for the top level table \textbf{Page Map Level 4}.
2. Capability: VNode_x86_64_pdpt for the table \textbf{Page Directory Pointer Table}
3. Capability: VNode_x86_64_pdir for the table \textbf{Page Directory}
4. Capability: VNode_x86_64_ptable for the actual table \textbf{Page Table}

As with every other capability, these are also stored in a specific CNode in a domain’s CSpace. This is the \texttt{PAGE\_CNODE} and its first element, that is the capability residing in slot 0 is a capability referring to the top level table, in this case, a VNode\_x86\_64\_pml4 capability.

```cap
VNode_x86_64_pml4 from RAM {
    /* PML4 */

    /* Base address of VNode */
    address genpaddr base;
    size_bits { vnode_size };
};
```

In the listing above we have the capability structure representing a pml4 table in the x86_64 architecture. It is indicative of all of the other tables in such a system, as they have exactly the same fields. We notice here that the \texttt{base} field is the physical address for the VNode.

\subsection*{2.2.6 Inter-dispatcher communication}

As we have seen so far, Barrelish is a modular OS. We have the kernel, the monitor the dispatchers and various other processes and daemons, running on the background in order to make this Operating System work without unexpected errors. In order for them to accomplish their goal they have, of course, to communicate and given that in Barrelish there is nothing shared between the cores, this can be challenging.
2.2. BARRELFISH

Inevitably, the Barrelfish team has implement the whole of the system’s communication using explicit message passing and RPCs. Every module that wishes to offer a service to another unit, or should be exchanging information in order to perform its task, should either export a service or “subscribe” to one, somewhat like a pub-sub system. On the other end of the channel lays the respective “other half” of the transaction. The communication ends can be either on the same core or on different cores. The difference is that in the former case there is only one monitor involved while on the later case the respective monitors of every core involved have to “meddle”.

As we have mentioned above, the monitor units are responsible for keeping the system up-to-date. Thus, their involvement in any kind of communication between domains is very crucial. Every domain that wishes to export a service, registers its name and just like a server program stands and waits for incoming connections that will be requesting one of its services. On the other endpoint, the requester module initiates the appropriate connection by contacting the monitor of its core and uses it to push forwards the request in question. The involvement of the monitor on all of there operations is the reason that during the start-up of every domain, there is a connection establish between it and the respective monitor.

Roughly all of the implementation of the interfaces that are used for this request-response communication is automatically generated. The programmer has to declare the specifics, like the method signatures, the arguments that are being sent and received and the functions handling the requests, but the main bulk of the communication is generated during compile time by the Flounder.

2.2.7 The glue that holds together: Hake

At this point and after having gone through the basic specific of Barrelfish we should mention that the tool that makes them all work together is Hake. Hake is used to compile everything, the kernel, the libraries, the user space space domains and binaries. It is also used to create the appropriate Makefiles and determine which domains are going to be installed and where.
Part II

Our Approach
Chapter 3

Problem Statement

3.1 Problem Description

3.1.1 Problem Statement

While Barrelish aims to maintain a minimal kernel, resulting in a CPU driver that implements only some basic functions, many functional issues and decisions are delegated to user-land and the programmer. One of those critical key issues is thread scheduling. Barrelish, like Windows and Sun Solaris falls into the category of the Operating Systems that differentiate between applications and threads, as opposed to Linux that makes no distinction between the two of them [3]. Since the kernel is the one that actually schedules dispatchers and in turn domain execution, it is the dispatcher itself that is responsible of scheduling threads. This means that thread scheduling is done in user mode.

One of the problems that occurs with moving that kind of responsibility to user space is system resources exhaustion. As it is up to the programmer to implement his own scheduling thread policy, one can end up with an application spanning on an arbitrary number of cores, with multiple threads burdening the system, when in fact there is no need for such extensive parallelism.

Another issue that is raised when multiple threads are spawned and when multiple cores are employed to a domains’ execution is the overhead of their synchronization. Because of the particular architecture of Barrelish and its applications we can understand that domains are somewhat “secluded”. Since state is not shared but replicated throughout all Barrelish, spanning a domain on multiple cores is more complicated than on other systems since specific capabilities have to be copied, monitored and maintained in order for a domain to run without any unexpected errors and traps. Plus, constant context switches and TLB flushes can lead to system’s performance being further impaired.

All of those reasons make it obvious that manipulating threads and facing all the consequential execution issues that should be better and closer monitored by the system and should not be left entirely to the programmer, in danger of “hogging” the system.
Although in Barrelfish it is easier to complete context switches between threads of the same domain, it is somewhat more elaborate to complete a context switch between processes and thus, dispatchers. The intuition behind our project was to remove all those decisions and implementation issues from user-space and somewhat restrict the programmer’s “saying” on them. We wish to remove any scheduling responsibilities from the dispatcher and that, can be achieved by restricting the runnable threads on the dispatcher to just one, that is removing any queue of threads on the dispatcher structure.

Additionally we wish to load and execute domains that will span on multiple cores and still be able to co-operate with each other. We wish, in essence to have a more co-operative way of running a domain. One of the basic steps in order to accomplish that, is having dispatchers share structures and resources.

Having in mind that all cross-core communication is explicit via message passing and that everything has to go through the respective monitors of the cores that wish to communicate, we can understand that even spanning domains and sharing virtual address spaces can be challenging and error prone.

Moving memory management responsibilities to the kernel would make all those issues easier to handle and would keep shared memory safe and consistent but would invalidate the initiative of keeping the CPU driver as lightweight and architecture neutral as possible. That is why, in our case, it only seemed fit to change the dispatcher structure and functions.
Chapter 4

Implementation

4.1 Our Approach

4.1.1 Dispatcherd

As it was previously mentioned, the dispatcher is the module that schedules the execution threads of a domain. This means that there is one dispatcher unit for every core that a domain might span on. Since it was our goal to create a more controlled scheduling unit, we had to decouple the notions of the dispatcher and the domain.

For our implementation we have modified the dispatcher unit in such a way that there is no need for thread scheduling. From the default version of the Barrelfish dispatcher structure we have removed the queue that all the threads, eligible to run, were placed on. That is the runq queue, thus transforming our approach of the dispatcher into a single-threaded one.

In the listing below we see part of the dispatcher structure that resides in user space. As we can see there is a struct thread *current pointing to the currently executing thread and the respective struct thread *runq, pointing to the rest of the threads that should be executed in the future.

We have already explained that the dispatcher is located in both kernel and user space. Since for our implementation we wish to modify the part that schedules threads, we altered only the one located in user space. There was no need for further meddling with the dispatcher structure that is handled by the kernel.

Another reason that pushed us towards the single threaded dispatcher implementation was the burdening of the system with the context switches and the scheduling. Every dispatcher runs only on one core and cannot make use of threads on different cores, so we wanted to rid it from the latencies that occur because of those multiple threads. Since we are talking for only one domain on only one core, having multiple threads on them would only render the system slower. That way, our system has a more task centric approach towards executing a domain.
struct dispatcher_generic {
    /// stack for traps and disabled pagefaults
    uintptr_t trap_stack [DISPATCHER_STACK_WORDS];
    /// all other dispatcher upcalls run on this stack
    uintptr_t stack [DISPATCHER_STACK_WORDS];
    /// Currently-running (or last-run) thread, if any
    struct thread *current;
    /// Thread run queue (all threads eligible to run)
    struct thread *runq;
    /// Cap to this dispatcher
    struct capref dcb_cap;
    ...
}

As we can see in the listing that follows, the dispatcher struct that we have
created does not have any structure for queueing up threads. We have called our
dispenser, Dispatcherd, in the sense that it spawns and waits on the background,
waiting for incoming requests to execute just like a daemon. It does not just execute
a domain and then exits.

struct dispatcherd GENERIC {
    /// stack for traps and disabled pagefaults
    uintptr_t trap_stack [DISPATCHER_STACK_WORDS];
    /// all other dispatcher upcalls run on this stack
    uintptr_t stack [DISPATCHER_STACK_WORDS];
    /// Currently-running (or last-run) thread, if any
    struct thread *current;
    /// Cap to this dispatcher
    struct capref dcb_cap;
    ...
}

4.1.2 Domain: Aquarium

The main change that we wish to accomplish with this project is to create a
dispenser that, besides running only one thread, it is also independent of a domain
in the sense that it will be a unit capable of executing various domains.

We have seen that in Barrelish, applications are nothing more than a collection
of dispatchers. We can not stressed enough that a domain is not something tangible,
something that is implemented using a structure or a source file. It is just a notion
that is used in Barrelfish to group together all of the dispatchers that are running
a specific binary.
4.1. OUR APPROACH

In the context of this project and the decoupling we wish to achieve, we have created a domain, the *Aquarium* domain that during its spawning process, it bypasses the default Barrelfish dispatcher and spawns on our own custom dispatcher structure. The intuition behind this peculiar domain is to be kept running on the background, somewhat like a daemon.

Our domain is kept on the standby, like a server and it is its duty to initialize, spawn and execute different domains, using its own dispatcher(s). That means that from now on, the domains we wish to spawn will **not** have their own dispatchers. Rather, they will be executed using our Aquarium dispatchers. Our server domain will thus act as a “container” and all the other applications we wish to run, will be spawned “inside” Aquarium.

To that extend, we should also mention that apart from “sharing” dispatchers, the domains we want to load and execute using our Aquarium domain and its respective dispatcher structures, also have to share virtual address space. This is inevitable if we want our domains to use the Aquarium’s dispatcher structure and if we wish to achieve a decoupling between our dispatcher and the domains.

**Spawning Aquarium**

In Barrelfish there are 2 modules responsible for initializing and bootstrapping user domains, the *spawnd* and the *startd* domains. As their names suggest they are both daemon processes that, after having exported their services, they are kept running on the background waiting to handle appropriate requests. They are both part of the Barrelfish library, that means that they operate in user space.

We should mention here that the process of spawning domains can vary given that there are different types of processes. For example the monitor module is spawned before the two modules mentioned above so it is spawned using a different sequence of steps. Plus, domains can be boot modules or just plain applications. In any case, the initialization of a domain is handled appropriately.

Before actually yielding the CPU to the new domain and calling its main function in order to start executing, there are different aspects of the domain that have to be set. The three major initializations that have to be completed are:

1. Setting up the CSpace. Basic CNodes are created, like the **Root CNode** and the **Task CNode** and initial capabilities are placed to the respective slots.

2. Setting up the VSpace. The virtual address space of the domain is initialized and spawned. This is done by creating the **Page CNode** of the new domain.

3. Setting up the Dispatcher. Different capabilities are created and mapped to handle the dispatcher and make it “runnable”. This includes setting up the appropriate registers that form the dispatcher’s state.

After these steps have concluded, once again we mention that these actions are handled by startd and especially, spawn domain and the new binary is loaded in
memory, the program can execute. Our intention to spawn the Aquarium domain on our custom dispatcher, involves of course, modifying the second step and adjusting the spawnd module to facilitate our implementation.

Further to that, we had to bypass the call to the function _spawn_setup_dispatcher_ of the Barrelfish library and execute our own _spawn_setup_dispatcherd_, which was created on the same source file to keep the library complete, and spawns the dispatcher structure. The other two steps need no modification, so the default functions of Barrelfish are called.

```c
/* Setup dispatcher frame */
if (strcmp(name, "aquarium") == 0) {
   err = spawn_setup_dispatcherd(si, coreid, name,
                                entry, arch_info);
   debug_printf("Spawning %s on dispatcherd ", name);
} else {
   err = spawn_setup_dispatcher(si, coreid, name,
                                entry, arch_info);
   debug_printf("Spawning %s on dispatcher ", name);
}
```

In the listing above we see the conditional clause that delegates the initialization of the dispatcher unit to the appropriate function. Once the dispatcher is able to start executing, the appropriate system calls are made and the CPU is yielded to the newly spawned domain.

Before the actual program code is executed there are some additional initializations that have to be done. This time all the appropriate actions are taken by the dispatcherd itself and not by a different module like in the case of spawnd. The dispatcherd is working in disabled mode and is taking care, among other things, the bootstrapping of the thread system.

```bash
startd.0: starting app /x86_64/sbin/fish on core 0
spawnd.0: Spawning fish on dispatcher
spawnd.0: spawning /x86_64/sbin/fish on core 0
startd.0: starting app /x86_64/sbin/aquarium on core 0
spawnd.0: Spawning aquarium on dispatcherd
spawnd.0: spawning /x86_64/sbin/aquarium on core 0
```

**Entering the dispatcherd**

Every time the CPU driver is about to dispatch a domain, the main thing that is of concern is whether the dispatcher of that domain was last seen in enabled or disabled mode. This is what will determine how the domain will resume and will re-
4.1. OUR APPROACH

solve whether the domain’s state will be restored from the enabled_save_area or the disabled_save_area. Those two fields can be found in the shared dispatcher structured as core local virtual addresses. In any case, CPU control is passed over to the domain and once again we are operating in user-space.

When we run enter a domain for the first time the procedure is a little different. After all the necessary information has been set by the spawn daemon as described above, the domain is called from assembly code to further initialize the domain and afterwards start execution. The elements that are initialized on the dispatcher’s stack are:

1. The dispatcher itself

2. The libarrelfish

3. The main thread that we run.

Spanning Aquarium

For our implementation, we have the Aquarium domain executing on every core during boot-time. Since we do not want to have different Aquarium instances on every core but just one, the Aquarium domain should not be spawned on every core. Rather, it should span on every present core, or at least on the ones we wish to use.

Spanning of course means sharing virtual address space and since everything, memory related, in Barrelfish is managed using capabilities, the appropriate invocations have to be done on the appropriate capabilities. Shared address space can be achieved by copying the capabilities the represent the specific page tables between cores and keeping them consistent. Of course this is unique for Barrelfish since nothing is shared between cores and everything has to be replicated.

In order to actually implement the shared address space, after our Aquarium domain has spawned on core 0, we have to send to all of the remote cores the capability that represents the root of our page tables, tree like structure. This is a VNode type capability, to be more precise a VNode_x86_64_pml4 capability, that resides on slot 0 of the page Cnode.

Although the dispatchers on the different cores share virtual address space, they do not share CSpace. As we have mentioned before, CSpace is unique to the domain and the dispatcher, this can be understood since nothing is shared amidst cores, so it could not be possible for two dispatchers running on different cores to share their CSpace.

So, after sending the VNode capability to the remote core, the accountable module, in this case the monitor, has to setup the CSpace for our domain on the remote core. Then, the VNode can be “embodied” in the newly spawned CSpace by copying it. Since all inter-dispatcher communication and message passing is done through the monitor, it is via them that the VNode capability will be sent from the originating core to all the others for the spanning to proceed.
Besides the VNode capability that is copied and sent, the dispatcher on the initial core also has to create the dispatcher that will be running on the remote core, map the respective page frame representing the new dispatcher in its virtual address space and afterwards send it to the remote core, alongside with the VNode. This is an essential step since by doing so, it manages to keep the virtual address space updated and the stack for the dispatcher has been created and mapped in our virtual address space.

Finally, after the dispatcher has been created and all the necessary information for spawning a domain has been set, the thread that our domain will run on is also created and our domain spans on multiple cores.

**Sharing Virtual Address Space**

**Dynamic Linking**

In order to be able to spawn and execute different domains “inside” our Aquarium domain, those domains have to share their virtual address space. One way to achieve that, is to use dynamic linking and the techniques used in shared libraries. Here we should note that it is one thing for the dispatcher of the same domain that run on different cores and share a virtual address space and quite another to run another binary, another process in an initial one.

Dynamic linking is different in the sense that the linker “lazily” waits till the last minute, that is during run time, to complete the linking between the application and various libraries or other object files. This way the application becomes more memory efficient, since it is not but when they are actually needed that the routines are loaded in memory and linked. A very common use for dynamic linking is shared libraries.

The feature that makes shared libraries unique and able to work is the fact that
4.1. OUR APPROACH

their code can be executed from any given address, they do not have to be assigned a specific address in order to be able to run. This ability, combined with dynamic linking renders shared libraries extremely ubiquitous. The libraries can be loaded only once in memory and still be able to be used by multiple processes. Since shared libraries can run no matter where they reside in memory we do not have to maintain different copies of the same library for every application that it is linked against.

The whole process of late linking and transforming the position independent code of shared libraries suitable for being called, is a responsibility of the dynamic linker. This means that even though our application might not be complete in file, our linker has to be. The linker is actually a complete binary, that is part of the Operating System. In an application binary we can find the information of the dynamic linker’s location can be found, assuming that everything has been compiled properly [9].

Finally, we should mention that given Barrelfish, is also using the ELF format for binaries, which is the default format and most convenient for dynamic linking due to being segmented, dynamic linking is a strong candidate to consider when loading domains comes in mind.

\[
\text{aquarium compiled file} \quad \rightarrow \quad \text{aux\_domain.so file} \quad \rightarrow \quad \text{dynamic linker} \quad \rightarrow \quad \text{executable} \quad ||\text{relocation info} \quad \rightarrow \quad \text{shared library}
\]

Figure 4.2. Dynamic Linking of Aquarium

Static linking

On the other hand lays the static linker. In this case scenario, everything is linked statically right after being compiled. It is the straightforward and simpler approach. The application is at its final form even in file and everything is copied inline before runtime. This might not be as memory efficient as dynamic linking but one can understand that it is much faster.

Not quite as elegant as the dynamic linker, when statically linking all symbols are assigned addresses after being compiled. There are no relocations to take place and our binary, once it is complete, can run on a system, no matter if the system has a dynamic linker or supports position independent code. To that extend, we see that is it more portable since everything that is needed for our application to run is already mapped in the binary.
On the other hand, it is obvious that being so much more memory consuming can be a serious disadvantage, especially since in contemporary systems multiple processes running on the same core is a given and memory resources and their optimal management is vital for scalable and robust systems.

Having said all of the above, for our implementation we have chosen the static linking, especially since Barrelfish does not provide shared libraries at this point. Additionally, since there are only so many applications that are running on Barrelfish, implementing shared libraries did not seem much of a necessity to spare memory. At least not for now that everything in Barrelfish is minimal and not so much resource consuming.

\[
\text{aquarium obj file} \quad \text{library.o file} \quad \text{linker} \quad \text{complete executable}
\]

Figure 4.3. Static Linking of Aquarium

### 4.1.3 RPC

As we have already stated, it is our main goal to be able to spawn domains inside our own Aquarium domain. This means that different modules and domains have to communicate with each other in order to start the execution of the different domains on specific cores.

Whether the auxiliary domains are dynamically or statically linked we want to be able to tell our container domain to execute them. This is of course done using fish, the Barrelfish terminal. Fish is a domain that is booted during start-up (using the default booting procedure and the default dispatcher structures) and which should be able to reach our server and “ignite” the execution of the secondary, auxiliary domains.

Since in Barrelfish all communication is done using explicit message passing, it is convenient to create a client for every “server” domain and use that client to interact with it. That was also our approach as we have implemented an *aqua_client* that can be used by any domain, in this case fish and make the appropriate calls to our server. This client was thus integrated in the libbarrelfish, that is the library of the Barrelfish, that provides the main functionality of the OS and without which no domain can execute.
4.1. OUR APPROACH

Additionally, we have created our own interface in order to render it possible for different domains to reach our server, via the aforementioned client. Our server after being spawned and then spanned, it exports its services and stands “alive” waiting for incoming requests. If a request arrives from another domain, at this case the fish domain, appropriate actions are taken by invoking specific functions.

```c
static struct aqua_rx_vtbl rx_vtbl = {
    .spawn_domain_on_aqua_call=spawn_on_aqua_handler,
    .span_domain_on_aqua_call=span_on_aqua_handler,
    .fib_domain_on_aqua_call=fib_on_aqua_handler,
    .fft_domain_on_aqua_call=fft_on_aqua_handler,
};
```

On the listing above we see the structure responsible for setting the appropriate handlers for every possible request our server process might receive. On the left hand of the equation we have the requests and on the right hand the functions that “serve” them.
CHAPTER 4. IMPLEMENTATION

4.2 Analysis

4.2.1 Benchmarks

In order to evaluate the performance of our system and benchmark we have chosen algorithms and implementation from the BOTS suite. For the comparison of the different implementations, we vary the values of cores and threads used in our system.

Fibonacci

For our implementation of Fibonacci benchmarks we have created different execution scenarios. First of, we have created a binary using the traditional Barrelfish approach. In this case, our binary is executed using the default Barrelfish dispatchers, it is spawned on core 0 and using specific routines we time the total execution time.

The exactly same code is “transformed” into a library and statically linked against our Aquarium domain. After Aquarium is spawned it executes this benchmark. In both case we have used the same code, the only thing that changes is the way the code is loaded in memory and executed. Additionally in both cases it is only the execution time that we measure.

For the second scenario our implementation of the dispatcher structure is used again. For the case of running the Fibonacci using the Aquarium domain we distinguish between two cases. For the first one we run benchmark on core 0 and for the second one we execute it on core 1.

<table>
<thead>
<tr>
<th>Fibonacci</th>
<th>Fibonacci in Aquarium (1 core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>34</td>
</tr>
<tr>
<td>49</td>
<td>40</td>
</tr>
<tr>
<td>36</td>
<td>58</td>
</tr>
<tr>
<td>53</td>
<td>43</td>
</tr>
<tr>
<td>58</td>
<td>44</td>
</tr>
</tbody>
</table>

Figure 4.4. Fib execution time in clock cycles $10^9$

In Figure 4 we see the time execution, in clock cycles of both Fibonacci running as a stand-alone Barrelfish domain and as an auxiliary domain run inside the Aquarium domain. As we notice we can see that in the later case the time execution has a tendency to be lower. For this specific table our Aquarium domain runs on only one core, it has been spanned.

In Figure 5 we see the execution times of Fibonacci again. This time the clock cycles of executing as a conventional domain are compared to the ones when the
4.2. ANALYSIS

<table>
<thead>
<tr>
<th>Fibonacci in Aquarium</th>
<th>Fibonacci in Aquarium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2 cores)</td>
</tr>
<tr>
<td>43</td>
<td>34</td>
</tr>
<tr>
<td>21</td>
<td>36</td>
</tr>
<tr>
<td>37</td>
<td>23</td>
</tr>
<tr>
<td>83</td>
<td>60</td>
</tr>
<tr>
<td>60</td>
<td>25</td>
</tr>
</tbody>
</table>

**Figure 4.5.** Fib execution time in clock cycles $10^9$

program is being spawned in Aquarium using 2 cores. The difference in the time execution in this case scenario is quite obvious.

Finally, in Figure 6 we see some comparable execution times when the our benchmark program is run in Aquarium using 1 and 2 cores respectively. Once again, we notice that when our domain spans on 2 cores the clock cycles spent are less.

<table>
<thead>
<tr>
<th>Fibonacci in Aquarium</th>
<th>Fibonacci in Aquarium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1 core)</td>
</tr>
<tr>
<td></td>
<td>(2 cores)</td>
</tr>
<tr>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>26</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>25</td>
<td>18</td>
</tr>
</tbody>
</table>

**Figure 4.6.** Fib execution time in clock cycles $10^9$

**FFT**

For the next benchmark algorithm we have chosen the FFT algorithm from the BOTS package. Once again we have created different execution scenarios. For our first experiment we run the FFT program as a simple Barrelfish domain. That is, we created a separate domain that uses the default dispatchers. Additionally, we created the FFT library and it is linked against the Aquarium domain. This way we use our custom dispatchers to run the benchmark program.

It is very interesting that while our “mainstream” FFT domain run with no problems at all, the version of it that was spawned inside Aquarium never managed to execute, since we kept receiving **page faults** and the domain turned unrunnable.

For our next experiment we spanned the Aquarium domain on core 1 and set the thread of the dispatcher on that core to execute the FFT benchmark. This time we had no problem making the call and executing the FFT.
In this case scenario we noticed that the FFT standalone domain has slightly better time performance. The difference is negligible but we can notice a trend. For the shake of arguments, we created additional threads for the FFT plain domain and had them run. We did not span the domain, we created additional threads on the same dispatcher, on core 0. When we examine the results we see that we did not got any grave performance boost. At some point we even noticed great latencies.
4.3 Conclusions & Future Work

4.3.1 Conclusions

While using the Fibonacci we noticed that running it in our Aquarium domain the execution time was consistently lower. Whether it was run on one or two cores the time needed seemed to be lower in our version of the benchmark program. One reason for this could be that we limited the scheduling duties of our dispatcher by modifying it into a single-threaded unit.

The biggest difference in execution time was of course noticed when we run our benchmarks on two cores. This is only natural since core 0 is heavily burdened with multiple domains and modules, at least far more than core 1 is. This happens because by default, every program is spawned on core 0 unless it is stated explicitly otherwise. This means that the system scheduler (that is local to every core of course) performs context switches between domains with a higher frequency on core 0 than on core 1.

When the FFT is concerned, we see that it is better executed as an independent domain. Even when we employ two cores for the Aquarium domain, the timing of the FFT is better when is run using the default Barrelfish domain mechanism.

4.3.2 Future Work

Dynamic Linking

By using static linking we have restricted our domain to executing other compiled files that have been previously been linked against. Our implementation is also quite memory consuming and everything has to be loaded in memory before execution. Although this approach might work and come in handy at this point that the Barrelfish kernel and the domains are quite minimal this might change in a while. Since memory allocation and “preservation” is always of the matter, it would be a major optimization if the linking in our approach could be turned to dynamic.

To achieve that a dynamic linker should be created that will be able to complete the binaries online. This of course would lead to a different kind of binaries since they would have to compiled under specific instructions and would have to be turned into Position Independent Code (PIC) [2].

Shared Libraries

The optimization mentioned above would be helpful since it could be the base for creating shared libraries in Barrelfish. As they have not yet been implemented, it would be a great attribute for the OS because as we see shared libraries are becoming a necessity.
CHAPTER 4. IMPLEMENTATION

Memory Protection

Actually spawning a domain inside another domain does not require memory regulations only in the context of dynamic or static linking. It also adds another level of memory protection. Given that two or even more processes will be sharing the same virtual address space there is the danger of interfering with each other’s data.

So, if a dynamic linking approach is developed an implementation of memory protection is vital in order to keep data private to the given domains. One way to go would be virtualization. This way every spawned domain is assigned its own virtual address space the same way similar a process running on a virtual machine has its own virtual address space.

The two techniques used in virtualization are shadow page tables and nested page tables. The former is an example of using only software to mimic a virtual address space inside another, whereas the latter requires the presence of specific hardware components and infrastructure.

In the scope of that discussion we mention that Intel uses the Extended Page Tables (EPTs) to implement augmented linear address translation, while combined with the Virtual Process Identifiers (VPIDs) produces the effect of virtualization. In its latest version, this mechanism provides the processor with a way to keep track, cache and translate between different address spaces simultaneously using a combination of VPIDs and Processor Context Identifiers (PCIDs) [11].

On the other hand, shadowing page tables does not require any special hardware. It is of course slower than using nested pages.

Hybrid Dispatcher

Since our application is supposed to span on various cores and since core 0 is always the one with the majority of the domains and servers running on it, it would be of great interested to have our Aquarium domain run the dispatcher on core 0 in disabled mode. Being able to restrict the scheduling and the synchronization of the rest of the dispatchers on core 0 and have the rest of the dispatchers execute the actual program code, i.e. run in enabled mode. We would end up with a hybrid domain that runs its most “administrative” issues on core 0 and carries out the actual workload on the rest of the cores.

Since we wish to turn to a solution that is more dynamic and cooperative and offers a viable work-stealing implementation, the need for a synchronization unit becomes apparent. This way the more burdened core is not preoccupied with actually executing programs, while the “lighter” cores are doing so.
Bibliography


