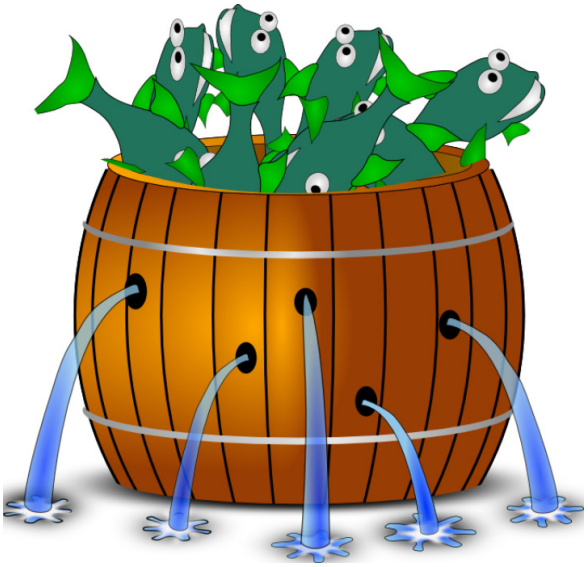
Barrelfish Project
ETH Zurich



Barrelfish Architecture Overview

Barrelfish Technical Note 000

Team Barrelfish

04.12.2013

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.0	24.06.2010	ikuz, amarp	Initial version
2.0	04.12.2013	troscoe,shindep	Extensively updated

Contents

Chapter 1

Barrelfish Architecture Overview

This document is intended as an introduction to Barrelfish. It presents a high level overview of the architecture of the Barrelfish Operating System, together with the most important concepts used inside the OS at various levels.

It does not provide a "getting started" guide to running Barrelfish on a PC, or in an emulation enviroment like Qemu or GEM5 - this is provided in other documentation.

1.1 High level overview

Barrelfish is "multikernel" operating system [3]: it consists of a small kernel running on each core (one kernel per core), and while rest of the OS is structured as a distributed system of single-core processes atop these kernels. Kernels share no memory, even on a machine with cache-coherent shared RAM, and the rest of the OS does not use shared memory except for transferring messages and data between cores, and booting other cores. Applications can use multiple cores and share address spaces (and therefore cache-coherent shared memory) between cores, but this facility is provided by user-space runtime libraries.

Figure 1.1 shows an overview of the components of the OS. Each core runs a kernel which is called a "CPU driver". The CPU driver maintains capabilities, executes syscalls on capabilities, schedules dispatchers, and processes interrupts, pagefaults, traps, and exceptions.

The kernel schedules and runs "Dispatchers". Dispatchers are an implementation of a form of scheduler activations [1] (the term is borrowed from K42 [13]), thus each dispatcher can run and schedule its own threads. Multiple dispatchers can be combined into a domain. Typically this is used to combine related dispatchers running on different cores. Often dispatchers in a domain share (all or part of) their vspace. Unless dispatchers in a domain run on the same core (which is possible, but rare) they cannot share a cspace (since cspaces are core specific).

Typically we refer to user level processes (or services or servers) as "user domains". Often these consist of a single dispatcher running on a single core.

1.2 CPU drivers

The kernel which runs on a given core in a Barrelfish machine is called a *CPU driver*. Each core runs a separate instance of the CPU driver, and there is no reason why these drivers have to be the same code.

In a heterogeneous Barrelfish system, CPU drivers will be different for each architecture. However, even on a homogeneous machine, CPU drivers for different cores can be specialized for some purposes (for

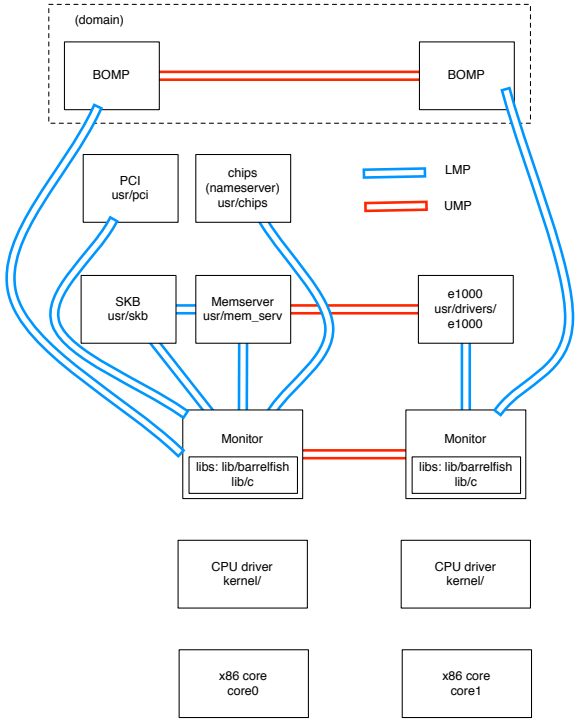


Figure 1.1: High level overview of the Barrelfish OS architecture

example, some might suppose virtualization extensions, while others might be optimized for running a single application).

CPU drivers are single-threaded and non-preemptible. They run with interrupts disabled on their core, and share no state with other cores. CPU drivers can be thought of as serially executing exception handlers for events such as interrupts, faults, and system calls from user-space tasks. Each such handler executes in bounded time and runs to completion. If there are no such handlers to execute, the CPU driver executes a user-space task.

The functions of a CPU driver are to provide:

- Scheduling of different user-space *dispatchers* on the local core.
- Core-local communication of short messages between dispatchers using a variant of Lightweight RPC [4] or L4 RPC [15].
- Secure access to the core hardware, MMU, APIC, etc.
- Local access control to kernel objects and physical memory by means of capabilities

CPU drivers do not provide kernel threads, for two reasons. Firstly, the abstraction of the processor provided to user space programs is a dispatcher, rather than a thread. Secondly, since the kernel is single-threaded and non-preemptible, it uses only a single, statically allocated stack for all operations. After executing a single exception handler, the contents of this stack are discarded and reset.

CPU drivers schedule dispatchers. The scheduling algorithm employed by a given CPU driver binary is configurable at compile time, and two are currently implemented:

- A simple round-robin scheduler. This is primarily used for debugging purposes, since its behavior is easy to understand.
- A rate-based scheduler implementing a version of the RBED algorithm [7]. This is the preferred per-core scheduler for Barrelfish, and provides efficient scheduling of tasks with a variety of hard and soft realtime jobs with good support for best-effort processes as well.

1.3 Dispatchers

The dispatcher is the unit of kernel scheduling, and on a single core roughly corresponds to the concept of process in Unix.

A dispatcher can be thought of as the local component of an application (often called a *domain* in Barrelfish) on a particular core. An application which spans multiple cores (for example, an OpenMP program) has a dispatcher on each core that it might potentially execute on.

Operating system tasks in Barrelfish are always single-core by design, and therefore have only one dispatcher. Applications, however, frequently have more, one on each core.

Dispatchers do not migrate between cores.

When a CPU driver decides to run a dispatcher as a result of a scheduling decision, it *upcalls* [8] into the dispatcher as an exit from kernel mode in the manner described below. The user-level code associated with the dispatcher then executes user-space thread scheduling, as well as handling other events (such as a page fault signal from the kernel or the arrival of an inter-domain message).

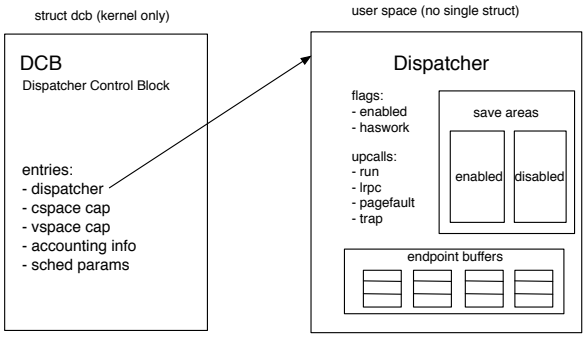


Figure 1.2: Dispatcher Control Block

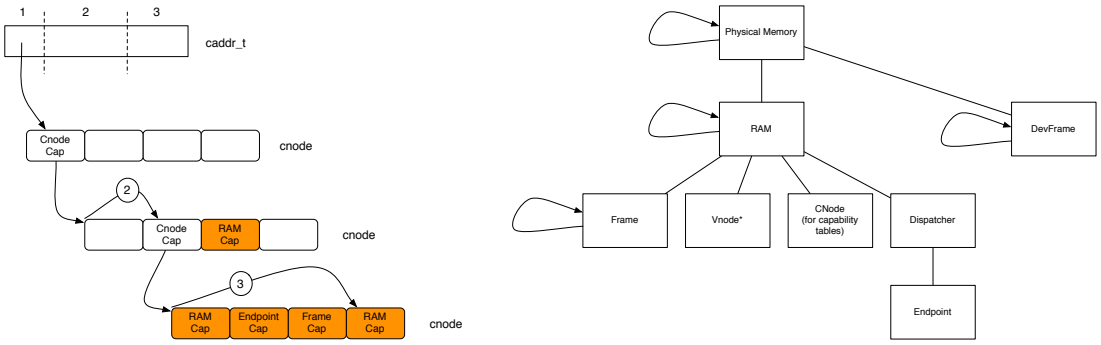
The kernel maintains a DCB (dispatcher control block) for each dispatcher (Figure 1.2). The DCB contains entries that define the dispatcher's cspace (capability tables), vspace (page tables), some scheduling parameters, and a pointer to a user space dispatcher structure (actually it consists of several structs). This struct manages the scheduling of the dispatcher's threads.

The dispatcher can be in one of two modes: enabled and disabled. It is in enabled mode when running user thread code. It is in disabled mode when running dispatcher code (e.g., when managing TCBs, run queues, etc.). The dispatcher defines a number of upcall entry points that are invoked by the kernel when it schedules the dispatcher.

The main upcall is `run()`. When the kernel decides to schedule a dispatcher, it brings in the base page table pointed to by the vspace capability, and calls the dispatcher's `run()` upcall. The dispatcher must decide which thread it wants to run, restore the thread's state, and then run it. Note that when `run()` is invoked the dispatcher first runs in disabled mode until the thread actually starts executing at which point the dispatcher switches to enabled mode.

When a dispatcher running in enabled mode is preempted, the kernel saves all register state to a save area (labelled "enabled"). When the dispatcher is next restarted it can restore this register state and run the preempted thread, or it can decide to schedule another thread, in which case it must first save the saved registers to a TCB.

When the kernel preempts a dispatcher running in disabled state, it stores the register state in a save area labelled "disabled". When the kernel schedules a dispatcher that is in disabled state, it does not invoke `run()`. Instead it restores the registers stored in the disabled save area. This causes the dispatcher to resume execution from where it was preempted.



(a) capref to capability translation.

(b) Capability Hierarchy

Figure 1.3: Capabilities

1.4 Runtime libraries

The basic runtime of any Barrelfish program is two libraries: a standard C library (currently a version of `newlib` [16]), and `libbarrelfish`. In practice the two libraries are inter-dependent, though there is an ongoing effort to minimize truly cyclic dependencies. The principal cycles involve `malloc()` and communication with the memory server.

The barrelfish library implements the following functionality:

- User-level thread scheduling, based on the upcall model of the dispatcher.
- Physical memory management by communication with the memory server.
- Capability management (maintaining the domain's cspace), including allocating new CNodes where necessary.
- Construction of virtual address spaces using capability invocations.
- User-level page fault handling.
- Communications using message channels, including blocking and nonblocking calls on channels, and the implementation of `waitsets` (analogous to Unix `select()` and `epoll()`).

1.5 Capabilities

Barrelfish uses a single model of *capabilities* [14] to control access to all physical memory, kernel objects, communication end-points, and other miscellaneous access rights.

The Barrelfish capability model is similar to the seL4 model [9], with a considerably larger type system and extensions for distributed capability management between cores.

Kernel objects are referenced by partitioned capabilities. The actual capability can only be directly accessed and manipulated by the kernel, while user level only has access to capability references (`struct capref`), which are addresses in a cspace. User level can only manipulate capabilities using kernel system calls (to which it passes capability references). In processing a system call the kernel looks up the capability reference in the appropriate cspace to find the actual capability; this process is similar to the translation of a virtual address to a physical address (Figure 1.3a). A capability reference can be used to get `caddr_t` and `valid bits` which are used in system calls (see `include/barrelfish/caddr.h`). Valid bits are used when the user wants to access cnode type capabilities (else, the translation process would continue by looking at entries in the cnode).

A Dispatcher only has access to the capability references in its cspace. As in seL4, capabilities are typed, and there is a strictly defined derivation hierarchy (e.g. Figure 1.3b). The type system for capabilities is defined using a domain-specific language called Hamlet.

1.6 Physical Memory

All regions of physical address space are referred to by means of capabilities. At present, a memory capability refers to a naturally-aligned, power-of-two-sized area of at least a physical page in size, though this restriction is likely to go away in the future.

A memory capability can be split into smaller capabilities, and this basic mechanism is used for low-level physical memory allocation.

Memory capabilities are *typed*, and each region type supports a limited set of operations. Initially, memory consists of untyped RAM and *device frames*, which are used to refer to memory-mapped I/O regions.

RAM capabilities can be retyped into other types; the precise number is defined in the Hamlet language in a file currently located in `/capabilities/caps.hl`. Some of the more common types of memory capability are:

- *Frame* capabilities are RAM capabilities which can be mapped into a user's virtual address space.
- *CNode* capabilities are used to hold the bit representation of other capabilities, and construct a domain's cspace. CNodes can never be mapped as writeable virtual memory, since this would remove the security of the capability system by allowing an application to forge a capability.
- *Dispatcher* capabilities hold dispatcher control blocks.
- *Page table* capabilities refer to memory which holds page table pages. There is a different capability type for each level of each type of MMU architecture.

1.7 Virtual Memory

Barrelfish applications are *self-paging* [12]: they securely construct and maintain their own page tables, and page faults are reflected back to the application by means of an upcall.

An application constructs its own virtual address space (generally abbreviated to *vspace*) using the capability system. It acquires RAM from a memory allocator in the form of RAM capabilities, and then retypes these to frames and page tables.

To take a 32-bit x86 machine without Physical Address Extensions (PAE) as a simple example, to map a frame into its vspace an application invokes the capability refering to a Page Table page using a system call. The arguments to this capability invocation are (1) a capability for a 4kB frame, (2) a slot number (from 0 to 1023), and (3) a set of mapping flags. This will cause the kernel to insert a Page Table Entry (PTE) with the appropriate flags into the corresponding slot in the Page Table.

Note that, while the user determines exactly what mapping is entered in the page table, the process is secure: the user cannot map a frame that they do not already hold a capability for, that capability must refer to a frame (and not some other type of memory, such as a dispatcher control block or CNode), and they must also hold a capability for the page table itself.

Similarly, in this example, for the Page Table page to be useful it must itself be referenced from a Page Directory. To ensure this, the user program must perform a similar invocation on the Page Directory capability, passing the slot number, flags, and the capability to the Page Table page. As before, the capability type system allows neither the construction of an invalid page table for any architecture, nor the mapping of any page that the user has no authorization for.

This model also has the additional feature that any core can construct a valid page table for any other core, even if the cores have different MMU architectures. An x86_64 core can construct a valid and secure ARMv7 virtual address space, and pass a capability for the L1 Page Table of this vspace to an ARMv7 core for installation.

Furthermore, since the application is responsible for constructing page tables and allocating physical memory itself, it can handle page faults by changing its own mappings, potentially paging data to and from stable storage in the process.

Naturally, this pushes significant complexity into the application. Paging functionaly is generally hidden in the runtime `libbarrelfish` library, though is available for direct manipulation for non-common use cases.

1.8 Inter-domain communication

Barrelfish provides a uniform interface for passing messages between domains, which handles message formating and marshalling, name lookup, and end-point binding.

This interface is *channel-based*: for two domains to communicate, they must first establish a channel (which may involve agreeing on some shared state). Messages are thereafter sent on this channel. The process of establishing channel state is known as *binding*.

A channel is typed, and the type determines the kinds of messages that can be sent on it. As with many RPC systems, channel types are defined using an *interface definition language*, which in Barrelfish is known as Flounder. Flounder interface types are generally found in the `/if` directory in the Barrelfish tree. Messages can pass both simple data types and, optionally, capabilities.

To establish a communications channel, a domain typically has to acquire an *interface reference* from somewhere, which identifies the remote endpoint it will try to connect to. In the common case, this is obtained from the System Knowledge Base, acting as a name server. After having bound an interface reference, the result is a local stub object at either end of the channel which can be called to send a message, and also implements dispatch of received messages.

In turn, an interface reference is created by a server creating a service (analogous to a listening socket) and registering this with a name service.

There are a variety of underlying implementations of message passing in Barrelfish, known as *Message Transports*. Each one is highly optimized for a particular hardware scenario. In general, the binding process automatically selects the appropriate transport for a channel, though this process can also be overwritten.

A message transport itself consists of several components:

1. An *interconnect driver* or ICD sends small, fixed-size units of data between domains. This is highly optimized: for same-core message passing, the LMP ("local message passing") ICD is based on L4's RPC path, while between cache-coherent cores the UMP ("user-level message passing") ICD is similar to URPC [5] or FastForward [11] and transfers single cache lines without involving the kernel. Other ICDs exist for specialized messaging hardware, and/or network communication.

   ICDs do *not* export a standard interface; each one is different.

2. *Stubs* generated by Flounder from an interface specification provide the uniform interface to clients and servers, and perform the abstraction of ICDs, as well as handling message fragmentation and reassembly in the cases where an application message is larger than the unit transferred by the ICD. Since different ICDs do not export the same interface, there is a different Flounder code generator for each ICD type, enabling cross-layer optimizations not possible otherwise.

3. Finally, some stubs can also invoke separate *Notification Drivers*, which provide a synchronous signal to the receiving domain in cases where the transfer of the message payload itself does not. For example, UMP messages have to be polled by the reciever by default, but on some architectures

an additional notification driver can be invoked by the stub to cause an inter-processor interrupt (IPI) after some time if the other end of the channel appears to be asleep.

In addition, stubs can also handle other areas of complexity resulting from the optimized nature of interconnect drivers. For example, capabilities cannot be sent directly over a UMP channel, since their bit representations cannot be safely made available to user-space applications. Instead, the stubs for UMP transport send capabilities over a separate, secure channel via the Monitors on the sending and receiving cores, which are themselves contacted using LMP (which can transfer capabilities) by the sending and receiving domains.

In this way, transfer of pure data over message transports can be extremely fast, at the cost of extra delay when transferring capabilities.

1.9 System Knowledge Base

Barrelfish includes a system service called the System Knowledge Base, which is used to store, query, unify, and compute on a variety of data about the current running state of the system. The SKB is widely used in Barrelfish; examples are:

- The Octopus lock manager and pub/sub system is built as an extension to the SKB (with a somewhat different interface language).
- Devices are discovered and entered into the SKB, and the SKB contains rules to determine the appropriate driver for each device. This forms device management (see below).
- The PCI driver uses constraint solving in the SKB to correctly configure PCI devices and bridges. The SKB contains a list of known PCI quirks, bugs, and special cases that have to taken in to account when allocating BAR values for address ranges. Interrupt routing is also performed in the SKB.
- The SKB functions as a general-purpose name server / interface trader for the rest of the OS.
- The results of online hardware profiling are entered into the SKB at boottime, and can be used to construct optimal communication patterns in the system.

The SKB implemention for Intel architectures is currently based on a port of the eCLiPse constraint logic programming system [2]. This consists of a Prolog interpreter with constraint solving extensions, and has been extended with a fast key-value store which can also retain capabilities. Aside from the Octopus interface, it is possible to communicate with the SKB by sending Prolog expressions as messages.

1.10 Device drivers

Device drivers in Barrelfish are implemented as individual dispatchers or domains. When they start up, they are supplied with arguments which include various option variables, together with a set of capabilities which authorize the driver to access the hardware device.

The capabilities a driver needs are generally:

1. *Device frame* capabilities for regions of the physical address space containing hardware registers; these regions are then memory-mapped by the driver domain as part of its initialization sequence.
2. *Interrupt* capabilities, which can be used to direct the local CPU driver to deliver particular interrupt vectors to the driver domain.
3. On x86 machines, *I/O capabilities* allow access to regions of the I/O port space for the driver.
4. For message-based device interfaces, such as USB, there may be *communication end-point capabilities* which allow messages to be sent between the driver domain and the driver for the host interface adaptor itself.

Hardware interrupts are received by a core, demultiplexed if necessary, and then turned into local inter-domain messages which are dispatched to the appropriate driver domain. Each first-level interrupt handler disables the interrupt as it is raised. As in L4, interrupts are renabled by the driver domain sending a reply message back to the CPU driver.

Driver domains themselves then export their functionality to clients (such as file systems, or network stacks) via further message-passing interfaces.

1.11 Device management

Device management in Barrelfish is performed by a combination of components:

- The System Knowledge Base is used to store information about all discovered hardware in the system. It also holds the information about which driver binaries should be used with which devices, and contains rules to determine on which core each device's driver domain should run.
- Octopus, the Barrelfish locking and pub/sub system built on top of the SKB, is used to propagate events corresponding to device discovery, hotplug, and driver startup and shutdown.
- Kaluga is the Barrelfish device manager, and is responsible to starting up driver domains based on information in the SKB and in response to events disseminated by Octopus. It handles passing the appropriate authorizations (in the form of device, I/O, and interrupt capabilities) to new driver domains.
- The driver domains themselves populate the SKB with further information about devices. For example, the PCI driver stores information about enumerated buses and functions in the SKB, and USB Host Adaptor drivers so something similar for enumerated devices.

In newer versions of Barrelfish, the same framework is also used for booting (and suspending or shutting down) cores. In addition to an appropriate CPU driver, each core other than the bootstrap core has a *Boot Driver*, which is responsible for booting the new core and encapsules the protocol (usually platform-specific) for core startup.

1.12 Monitor

Each Barrelfish core runs a special process called the *Monitor*, which is responsible for distributed coordination between cores. All monitors maintain a network of communication channels among themselves; any monitor can talk to (and identify) any other monitor. All dispatchers on a core have a local message-passing channel to their monitor.

Monitors are trusted: they can fabricate capabilities. This is because they are responsible for transferring capabilities between cores, and so must be able to serialize a capability into bits and reconstruct it at the other end. Monitors therefore posess a special capability (the *Kernel capability*) which allows them to manipulate their local core's capability database.

The monitor performs many low-level OS functions which require inter-core communication (since CPU drivers by design do not communicate with each other, nor share any memory). For example:

- Monitors route inter-core bind requests for communication channels between domains which have not previously communicated directly.
- They send capabilities along side channels, since regular domains on different cores cannot send capabilities directly themselves without involving the kernel.
- Monitors help with domain startup by supplying dispatchers with useful initial capabilities (such as a communication channel back to the monitor, and ones to the memory allocator and SKB).

- They perform distributed capability revocation, using a form of two-phase commit protocol among themselves over the capability database.

The monitor contains a distributed implementation of the functionality found in the lower-levels of a monolithic kernel. The choice to make it into a user-space process rather than amalgamating it with the CPU driver was an engineering decision: it results in lower performance (many operations which would be a single syscall on Unix require two full context switches to and from the monitor on Barrelfish). However, running the monitor as a user-space process means it can be time-sliced along with other processes, can block when waiting for I/O (the CPU driver has no blocking operations), can be implemented using threads, and provides a useful degree of fault isolation (it is quite hard to crash the CPU driver in Barrelfish, since it performs no blocking operations and requires no dynamic memory allocation).

1.13 Memory Servers

Memory servers are responsible for serving RAM capabilities to domains. A memory server is started with an initial (small) set of capabilities which list the regions of memory the server is responsible, and they respond to requests from other domains for RAM capabilities of different sizes.

At system startup, there is an initial Memory Server running on the bootstrap core which is created with a capability to the entire range of RAM addressable from that core. However, the use of capabilities allows this to delegate management of subregions of this space to other servers. This is desirable for two reasons:

- It allows core to have their own memory allocators, greatly improving parallelism and scalability of the system. As in other systems, Barrelfish's per-core allocators can also steal memory from other cores if they become short.
- It permits different allocators for different types of memory, such as different NUMA nodes, or low-memory accessible to legacy DMA devices.

Each new dispatcher in Barrelfish is started with a bootstrapped message channel to its own local memory server.

1.14 Filing system

Barrelfish at present has no native file system. However, runtime libraries do provide a Virtual File System (VFS) interface, and a number of backends to this exist (and are used in the system), including:

- An NFSv3 client.
- A simple RAM-based file system.
- Access to the OS multiboot image via the VFS interface.
- A FAT file system which can be used with the AHCI driver and ATA library.

1.15 Network stack

At time of writing, the Barrelfish network stack is evolving, but Figure 1.4 shows the high-level structure. Barrelfish aims at removing as much OS code from the datapath as possible, and uses a design inspired by Nemesis [6] and Exokernel [10].

Each physical network interface has a driver which is started by the Kaluga device manager service. This driver is in turn capable of initiating a separate *queue manager* for each hardware queue supported

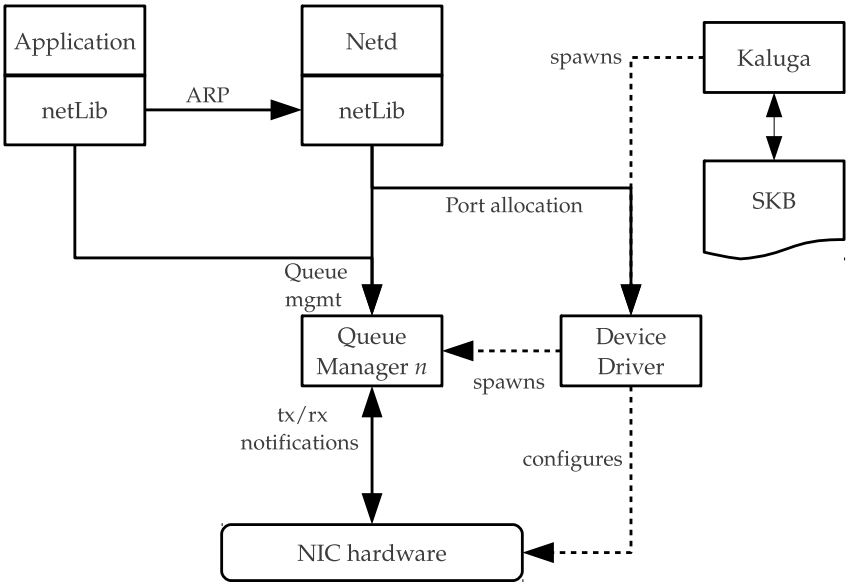


Figure 1.4: High level overview of the Barrelfish network stack

by the network hardware, and configuring the NIC hardware to demultiplex incoming packets to the appropriate queue.

Applications run their own network stack on a per-flow basis, either reading and writing packets directly to and from hardware queues (if possible), or with the queue manager performing an extra level of multiplexing on each queue. The queue manager receives events from the NIC hardware signalling transmission and reception of packets.

An additional domain, the network daemon (`netd`), is responsible for non-application network traffic (such as ARP requests and responses, and ICMP packets). This daemon also handles allocation of ports from talking to a given network interface.

Chapter 2

The Barrelfish source tree

The Barrelfish source tree is organized as follows:

`capabilities/`:
: Definitions of the capability type system, written in the Hamlet language.

`devices/`:
: Definitions of hardware devices, written in the Mackerel language.

`doc/`:
: LaTeXSource code for the Barrelfish Technical Notes, including this one.

`errors/`:
: Definitions of Barrelfish system error codes, written in the Fugu language.

`hake/`:
: Source code for the Hake build tool.

`if/`:
: Definitions of Barrelfish message-passing interface types, written in the Flounder language.

`include/`:
: Barrelfish general header files.

`kernel/`:
: Source code for Barrelfish CPU drivers.

`kernel/include/`:
: Header files internal to Barrelfish CPU drivers.

`lib/`:
: Source code for libraries.

`tools/`:
: Source code for a variety of tools used during the build process.

`trace_definitions/`:
: Constant definitions for use by the Barrelfish tracing infrastructure, written in the Pleco language.

`usr/`:
: Source code for Barrelfish binaries.

`usr/drivers/`:

Source code for Barrelfish device driver binaries.

Chapter 3

The Barrelfish build tree

Successfully building Barrelfish results in a series of files and directories in the build directory. Regardless of the architectures requested, the following are likely to be present:

`docs/`:

PDF files corresponding to the Barrelfish technical notes, including this one. Note that these files are probably not built by default, you make have to invoke `make docs`.

`hake/`:

Build files and binary for `hake`, the Barrelfish build tool. You should not need to look in here unless you need to edit `Config.hs`, the system-wide configuration file.

`Hakefiles.hs`:

A very large Haskell source file containing all the Hakefiles in the system, concatenated together along with some dynamic information on the files in the build tree. You should not need to refer to this file unless you encounter a bug in Hake.

`Makefile`:

A very large Makefile, which contains explicit rules to build every file (including intermediate results) in the Barrelfish build tree. It is sometimes useful to search through this file if the build is failing in some unexpected way. The only files this Makefile includes are `symbolic_targets.mk` and generated C dependency files; all other make information is in this one file.

`menu.lst`:

An initial file for booting Barrelfish via grub.

`skb_ramfs.cpio.gz`:

A RAM filing system image in the form of a CPIO archive containing support Prolog files for the Barrelfish System Knowledge Base (SKB).

`sshd_ramfs.cpio.gz`:

A RAM filing system image in the form of a CPIO archive containing support files for the Barrelfish port of the OpenSSH server.

`symbolic_targets.mk`:

A file containing symbolic `make` targets, since all rules in the main, generated Makefile are explicit. This file contains the definitions necessary to build complete platform images of Barrelfish for various hardware architectures.

`tools/bin/`:

Directory containing binaries (for the host system) of various tools needed to building Barrelfish, such as `flounder`, `fugu`, `hamlet`, and `mackerel`.

`tools/tools/`:

Intermediate object files for building the tools binaries

`tools/tmp/`:

Miscellaneous intermediate files, mostly from building Technical Note PDF files.

In addition, there will be a directory for each architecture for which this build was configured, such as `x86_64`, `86_32`, `ARMv7`, etc. This will contain the following items of interest (taking `x86_64` as a concrete example):

`x86_64/capabilities/`:

C code generated by Hamlet to encode the capability type system.

`x86_64/errors/`:

C code generated by Fugu to encode the core OS error conditions.

`x86_64/include/`:

Assorted include files, both generated and copied from the source tree.

`x86_64/include/dev`:

Device access function header files generated by Mackerel.

`x86_64/include/if`:

Stub definition header files generated by Flounder.

`x86_64/kernel/`:

Intermediate build files for all the CPU drivers built for this architecture.

`x86_64/lib/`:

Static libraries for the system, together with intermediate build files for the libraries.

`x86_64/sbin/`:

All the executable binaries built for this architecture. The CPU driver is generally known as `/sbin/cpu`, or a similar name more specific to a given core.

`x86_64/tools/`:

Miscellaneous tools specific to a target architecture. Typically ELF code, bootloaders, and code to calculate assembly offsets from C definitions.

`x86_64/trace_definitions/`:

Generated files giving constants for the tracing system in C and JSON (for use by Aquarium2).

`x86_64/usr/`:

Intermediate build files for all the executable binaries whose source is in `/usr`.

Chapter 4

An overview of Barrelfish documentation

Barrelfish a rapidly-changing research operating system, and consequently it is difficult and resource-intensive to maintain coherent, up-to-date documentation at the same time. However, efforts are made to document the system in a number of ways, summarized below.

4.1 Technical notes

The Barrelfish source contains a number of technical notes, which are rough-and-ready (and incomplete) documentation, tutorials, reference manuals, etc. for the system. The documents evolve over time, but are often the best source of documentation for the system.

This document is Barrelfish Technical Note 0. The remaining technical notes are as follows:

1. **Glossary:** A list of Barrelfish-specific terms with definitions.
2. **Mackerel:** Reference manual for the Barrelfish language for specifying hardware registers and in-memory datastructure layouts.
3. **Hake:** A comprehensive manual for the Barrelfish build utility.
4. **Virtual Memory in Barrelfish *obsolete*:** A brief description of virtual memory support in Barrelfish, now superceded by Simon Gerber's dissertation "Virtual Memory in a Multikernel".
5. **Barrelfish on the Intel Single-chip Cloud Computer:** A description of the Barrelfish support for the SCC (Rock Creek) processor, together with some performance evaluations and discussion of the hardware.
6. **Routing in Barrelfish:** An early design for routing inter-core messages over multiple hops within the system.
7. **Tracing and Visualization:** The Barrelfish trace infrastructure and associated tools.
8. **Message notifications:** A discussion of the use of notification drivers to mitigate the effect of polling when using UMP message channels.
9. **Specification** A document specifying the Barrelfish API and the core kernel implementation.
10. **Inter-dispatcher communication in Barrelfish:** Documentation for the IDC system, including the Flounder interface definition language, and some of the more commonly used interconnect drivers and message transports.

11. **Barrelfish OS Services:** A list of services running in a typical Barrelfish machine, together with their interdependencies.
12. **Capability Management in Barrelfish:** Documentation for the user-level capability primitives for maniulating caprefs and CNodes.
13. **Bulk Transfer:** A proposed design for transferring large contiguous areas of memory between Barrelfish domains.
14. **A Messaging interface to disks:** A comprehensive description of the Barrelfish AHCI driver.
15. **Serial ports:** A discussion of how serial ports are represented in Barrelfish, both inside the CPU driver and from user space.

Barrelfish technical notes are built from the Barrelfish tree (using `make Documentation`), and are also available pre-built from the Barrelfish web site at `http://www.barrelfish.org/`.

4.2 The Barrelfish Wiki

The Barrelfish project public wiki (at `http://wiki.barrelfish.org/`) contains a variety of additional documentation, including many contributions from users outside the core Barrelfish team.

4.3 Academic publications

Many publications related to Barrelfish can be found on the Barrelfish web site. They are not generally detailed documentation for the system, but often convey high-level design decisions, concepts, and rationales.

4.4 Doxygen

The Barrelfish libraries are annotated for use with Doxygen.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[2] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. Oct. 2009.

[4] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, 1991.

[6] R. Black, P. T. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *Proceedings of the 22Nd Annual IEEE Conference on Local Computer Networks*, LCN '97, pages 179–188, Washington, DC, USA, 1997. IEEE Computer Society.

[7] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 396–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 171–180, New York, NY, USA, 1985. ACM.

[9] D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st workshop on isolation and integration in embedded systems (IIES '08)*, pages 35–40, 2008.

[10] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, Feb. 2002.

[11] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.

[12] S. M. Hand. Self-paging in the nemesis operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[13] IBM K42 Team. *Scheduling in K42*, Aug. 2002. Available from `http://www.research.ibm.com/K42/`.

[14] H. M. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[15] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[16] Sourceware. Newlib. `http://www.sourceware.org/newlib/`, November 2013.