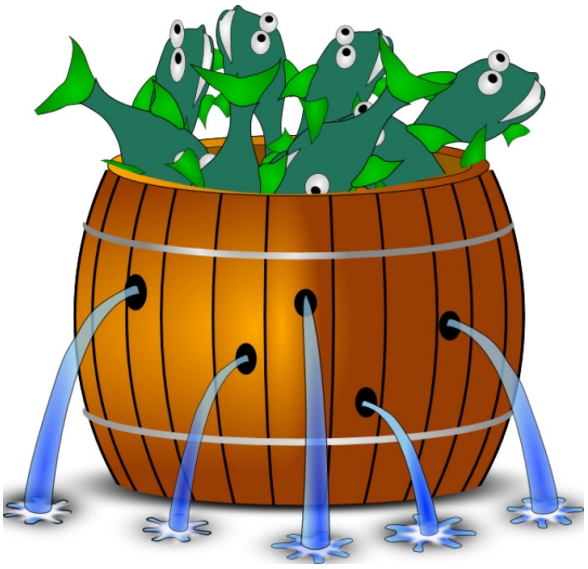
Barrelfish Project
ETH Zurich



Coreboot in Barrelfish

Barrelfish Technical Note 23

Barrelfish project

02.03.2017

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
0.1	28.02.2017	GZ	Initial Version
0.1	02.03.2017	RA	Adding basic structure

Contents

Chapter 1

Introduction

This document describes the way Barrelfish boots cores on supported architectures. We first explain the terminology used throughout this document. Then, we will give an overview of supported operations and finally, explain in detail how the implementation of these operations on differ for various supported architectures.

1.1 Boot drivers

Barrelfish uses *boot drivers*, which is a piece of code running on a "home core" and manages a "target core". It encapsulates the hardware functionality to boot, suspend, resume, and power-down the latter. Boot drivers run as processes, but closely resemble device drivers and could equally run as software objects within another process. The boot driver abstraction treats CPU cores much like peripheral devices, and allows us to reuse the OS's existing device and hot-plug management infrastructure to handle new cores and select drivers and kernels for them. The code for the Barrelfish boot driver can be found in `usr/drivers/cpuboot/`. From the source folder, a binary called "corectrl" is generated at build time and can be used to do target core management. The "corectrl" program is a command line tool that supports different operations as described below.

1.2 CPU driver

Is the binary that runs on a core and executes in privileged mode. CPU drivers are the Barrelfish equivalent of a kernel, except that there is one per core, and they share no state or synchronization. CPU drivers are typically non-preemptible, single-threaded, and mostly stateless. The CPU driver binary is selected and loaded by a *boot driver*, which among other things, is responsible of making sure the CPU driver is executed on a core. TN-021 contains a more detailed explanation of CPU drivers and the provided functionality.

The code for the Barrelfish CPU drivers can be found in `kernel/`.

1.3 Kernel Control Block (KCB)

The Kernel Control Block (or KCB) is a memory region that can only be accessed by kernel-space. However, a program, especially boot drivers, can create and reference KCBs in user-space through the capability system. The Kernel Control Block's purpose is to hold or keep pointers to all per-core state. All per-KCB state maintained by a kernel must be reachable from the KCB itself or reconstructible by reading the KCB.

Usually, a KCB is created for each core we start and passed to the new core on start-up. On boot-up, the kernel checks if the KCB it received is initialized. In that case, it will use the KCB and start to dispatch the applications it contains. If the KCB is uninitialized, the kernel sets all the entries of the KCB to its default values and continues bootstrapping the core (and the KCB block) by starting the monitor application, which is critical for communication across cores in user-space.

Every kernel has the possibility to maintain multiple KCBs. The kernel maintains a ring structure of KCBs it owns. A CPU driver supports switching to another KCB on demand. In case a CPU driver is running multiple KCBs it divides the time-slices evenly among different KCBs and switches KCBs after a given amount of time has passed. Currently, the CPU driver uses round-robin scheduling for having multiple KCBs on one core but other models are possible as well.

Chapter 2

Generic Operations

The boot driver (currently called "corectrl") has support for a range of operations, which we describe in more detail in this chapter. As a summary the operations are listed here:

- boot <target coreid>: Boots a new core with a KCB.
- stop <target coreid>: Stop execution on an existing core.
- update <target coreid>: Update the CPU driver on an existing core.
- give <from kcbid><to kcbid>: Give kcb from one core to another.
- rmkcb <kcbid>: Remove a running KCB from a core.
- lscpu: List current status of all cores.
- lskcb: List current KCBs.

Note that the implementation of "corectrl" in it's current from is essentially a command line tool that must be invoked for every operation. This means that the boot driver is currently state-less which is fine for just executing the operations since all information required can be reconstructed from the SKB.

There are a few optional flags that can be passed to the different operations:

- -d, –debug: Print debug information
- -k, –kernel <binary>: Overwrite default kernel binary.
- -x, –monitor <binary>: Overwrite default monitor binary.
- -a, –kargs <args>: Overwrite default kernel command line arguments.
- -n, –newkcb: Create a new KCB even if there is already one for that core.
- -m, –nomsg: Don't wait for a monitor message.

2.1 boot <target coreid>

Starts a CPU driver on a core. If the new core that has never been started before, it is brought online as follows:

The new core is detected by some platform-specific mechanism (e.g., ACPI) and its appearance registered with the device management subsystem. This is done by adding an octopus record to the SKB. The record name is of the format `hw.processor.ID` where ID is an identifier guaranteed to be unique by the SKB/Octopus. The record has the following mandatory, architecture independent fields:

- `enabled`: A boolean field that says if the core is usable or not (for example, on x86 if hyper-threads are disabled, this flag would be set to false for hyper-threads – the resulting core would not be booted by default).
- `barrelfish_id`: A unique integer identifier used by Barrelfish to refer to this core.
- `hw_id`: An identifier assigned by the hardware/platform to this core.
- `type`: A number that corresponds to that cores architecture as described in `enum cpu_type` (defined in barrelfish_kpi/cpu.h).

Barrelfish selects and starts an appropriate boot driver for the new core. This is done by propagating the added octopus record to kaluga which then invokes "corectrl" with the boot command.

The boot driver selects a kernel binary and arguments for the new core, and directs the boot driver to boot the kernel on the core. By default "corectrl" uses the `type` field in the octopus record to decide which binary to spawn on the target core. However, this can be overridden by manually selecting a kernel with the –kernel flag. Alternatively, the default parameters passed to the kernel can be overridden with the –kargs option.

The boot driver loads and relocates the kernel, and executes the hardware protocol to start the new core. This is done by using the file system to load the binaries and additionally a series of system calls to invoke protected operations (starting a core, sending an IPI etc.). In this process, also a new KCB is created for the core and initialized with default arguments.

The new kernel initializes and uses existing Barrelfish protocols for integrating into the running OS. This involves spawning the first user-space program (the monitor) on the new core. The default choice for the first user-space program loaded on the new core can be overridden by using the –monitor flag.

The monitor will inform the other boot driver about its existence by sending a message back to it. Then, the boot driver will tell its local monitor about the newly available and initialized core. In some cases (for example if something else is spawned than the default monitor binary) we may not want to wait for such a message. If the –nomsg flag is passed to "corectrl" this step is skipped.

The SKB is updated with information about the new core: The core is marked online and booted. The KCB that was created during the boot process by "corectrl" is stored in the capability storage in case it needs to be retrieved at a different point in time.

At least three special cases need to be considered:

- The core id is invalid: If we don't find the core in the SKB, the operation is aborted.
- The core is already running: In that case we abort with an error.
- The core has been booted before: By default "corectrl" will check if an existing KCB for the given core already exists. If yes, the capability for the KCB will be retrieved from

the SKB and passed to the new core. This behavior can be overridden by passing the –newkcb flag which makes sure that an existing KCB is not reused.

2.2 stop <target coreid>

Stops a CPU driver on a core. If the core is already stopped, nothing is done. The KCB running on this core will stop dispatching until the core is either restarted, or the KCB is given to another core.

In general the operations executed is follow:

- Invoke hardware specific protocol to stop the core.
- Optional: Update meta-data in SKB to mark core as stopped.

This operations does not take any optional flags or additional arguments.

Note that while this is deliberately very simple and low-overhead, just having the KCB stop to execute is generally a bad idea as there may be messages waiting to be received on that core etc. which can cause the whole system to lock-up. It makes sense for example to park the KCB after stopping (see the give operation later).

2.3 update <target coreid>

Update is implemented as a sequence of executing the `stop` and `start` command. It takes the same arguments and flags as described in the start section.

2.4 give <target kcbid><destination kcbid>

Removes a KCB on the core it is currently executing and transfers it to the core that is currently dispatching the destination KCB. This command involves several steps:

The boot driver instructs its local monitor to send a message to the monitor of the target KCB. The message instructs the receiving monitor to inform its local CPU driver to stop dispatching the KCB (or OSNode).

The boot driver sends message to the monitor of the destination KCB, passing along the cap reference of the target KCB. Upon receipt, the destination monitor will hand the reference to its CPU driver.

The destination CPU driver will do some minimal initialization, resetting timers and scheduling state, check for registered interrupts and up-call drivers to re-register them, and then start dispatching the target KCB.

The boot driver will update the SKB to change the core assigned to the target kcbid.

The following special cases need to be considered:

- The target or destination KCB ID is invalid: The operation is aborted.

- The destination KCB is not currently assigned a core: The operation is aborted.
- The target KCB is not currently running on a core: The step where we first send a message to the target KCB is skipped.

2.5 rmkcb <kcbid>

The boot driver instructs its local monitor to send a message to the monitor of the target KCB.

Upon receipt, the monitor of the target KCB will instruct the CPU driver to remove the KCB from the scheduling queue. If queue is now empty, the CPU driver will stop itself, otherwise it will switch to the next KCB in the list.

The following special cases need to be considered:

- The KCB id is invalid: If we don't find the KCB capability in the SKB, the operation is aborted.
- The KCB is not dispatched on any core: If the KCB is not currently associated with any core, it can't be removed and therefore the operation is a NOP.

[*Don't remember exactly if we need to send this to KCBID we want to remove or the core/KCB that is currently hosting us...*]

Chapter 3

Booting x86 Cores

3.1 Core discovery and identification

3.2 Booting a new core

Chapter 4

Booting ARMv7 Cores

[*explain ARMv7 specific boot arguements / protocols*]

4.1 Core discovery and identification

4.2 Booting a new core

Chapter 5

Booting ARMv8 Cores

[*explain ARMv8 specific boot arguements / protocols*]

5.1 Core discovery and identification

5.2 Booting a new core

5.2.1 Power State Coordination Interface (PSCI)

5.2.2 Parking Protocol

5.2.3 All at once

Raspberry Pi