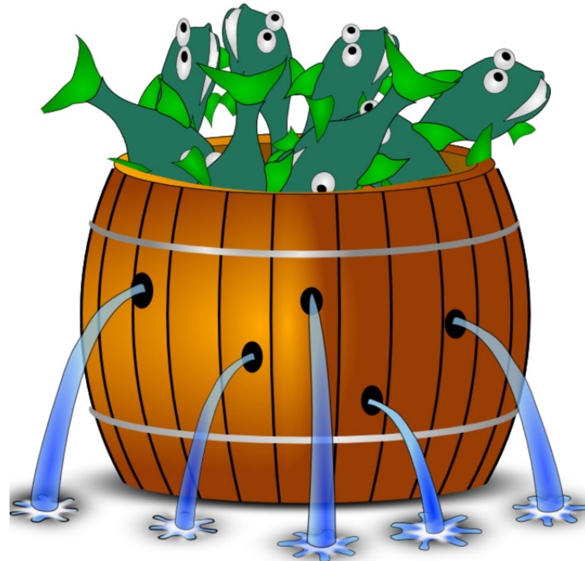
Barrelfish Project
ETH Zurich



CPU drivers in Barrelfish

Barrelfish Technical Note 21

Barrelfish project

01.12.2015

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	01.12.2015	GZ	Initial Version

Contents

1 Introduction **5**
1.1 General design decisions 5
1.2 Code file structure and layout 5

2 x86-64 **6**
2.1 Boot process 6
2.1.1 BSP Core 6
2.1.2 APP Core 7
2.2 Virtual Address Space 7
2.3 IO capabilities 7
2.4 Global Descriptor Table (GDT) 8
2.5 Interrupts and Exceptions 8
2.6 Local Descriptor Table (LDT) 10
2.7 Registers 10
2.8 Hardware devices 10
2.8.1 Serial port 10
2.8.2 PIC – Programmable Interrupt Controller 11
2.8.3 xAPIC – Advanced Programmable Interrupt Controller 11
2.8.4 System call API 11

Chapter 1

Introduction

This document describes the CPU driver, the part of Barrelfish that typically runs in privileged mode (also known as kernel) on our supported architectures.

Barrelfish currently supports the following CPU drivers for different CPU architectures and platforms:

- x86-32
- x86-64
- k1om
- ARMv7
- ...
- ARMv8

1.1 General design decisions

- No dynamic memory allocation
- No preemption
- ...

1.2 Code file structure and layout

TODO: Should explain things such as naming, where goes architecture dependent, platform specific code? What libraries we use in the kernel? Where is the shared code between libbarrelfish and a cpudriver?

Chapter 2

x86-64

The x86-64 implementation of Barrelfish is specific to the AMD64 and Intel 64 architectures. This text will refer to features of those architectures. Those and further features can be found in [?] and [?] for the Intel 64 and AMD64 architectures, respectively.

2.1 Boot process

We first describe the boot process for the initial BSP core, followed by the boot process of an APP core.

2.1.1 BSP Core

Barrelfish relies on a multiboot v1 [?] compliant boot-loader to load the initial kernel on the BSP core. In our current set-up we use GRUB as our boot-loader which contains an implementation of the multiboot standard.

On start-up, GRUB will search the supplied kernel module (on x86-64 this is the binary called elver in `tools/elver/`) for a magic byte sequence (defined by multiboot) and begin execution just after that sequence appeared (see `tools/elver/boot.S`).

`boot.S` in elver will set-up an preliminary GDT, an IA32-e page-table, and stack for execution. `elver.c` will then search for a binary called **`kernel`** or **`cpu`** in all the multiboot modules, relocate that module and then jump to the relocated kernel module. At this point, we have set-up a 1 GiB identity mapping of the physical address space using 2 MiB pages in order to address everything we need initially.

Note that the reason elver exists is because multiboot v1 does not support ELF64 images (or setting up long-mode). If we use a bootloader that supports loading relocatable ELF64 images into 64-bit mode, elver would be redundant.

After **`elver`** is done, execution in the proper BSP kernel program begins in `kernel/arch/x86_64/boot.S` which then calls *`arch_init`*, the first kernel C entry point.

2.1.2 APP Core

APP cores are booted using the coreboot infrastructure in Barrelfish. The logic that boots APP cores resides in `usr/drivers/cpuboot`.

The source code responsible for booting a new core on x86 is found in `usr/drivers/cpuboot/x86boot.c`, specifically in the function called *spawn_xcore_monitor*. *spawn_xcore_monitor* will load the **kernel** and **monitor** binary, and relocate the kernel. The function called *start_aps_x86_64_start* will afterwards map in the bootstrap code (which is defined in `init_ap_x86_64.S`) for booting the APP core. One complication for this code is that it has to resides below 1 MiB in physical memory since the new APP core starts in protected mode and therefore can not address anything above that limit in the beginning. Once the mapping is initiated, the entry point address for the new APP kernel will be written into this memory region. Finally, a set of system calls are invoked in order to send the necessary IPIs to bootstrap the new processor.

2.2 Virtual Address Space

The page table is constructed by copying VNode capabilities into VNodes to link intermediate page tables, and minting Frame / DeviceFrame capabilities into leaf VNodes to perform mappings.

When minting a frame capability to a VNode, the frame must be at least as large as the smallest page size. The type-specific parameters are:

1. **Access flags:** The permissible set of flags is PTABLE_GLOBAL_PAGE — PTABLE_ATTR_INDEX — PTABLE_CACHE_DISABLED — PTABLE_WRITE_THROUGH. Access flags are set from frame capability access flags. All other flags are not settable from user-space (like PRESENT and SUPERVISOR).

2. **Number of base-page-sized pages to map:** If non-zero, this parameter allows the caller to prevent the entire frame capability from being mapped, by specifying the number of base-page-sized pages of the region (starting from offset zero) to map.

[*address space layout after initialization is done*]

2.3 IO capabilities

IO capabilities provide kernel-mediated access to the legacy IO space of the processor. Each IO capability allows access only to a specific range of ports.

The Mint invocation (see sec:mint) allows the permissible port range to be reduced (with the lower limit in the first type-specific parameter, and the upper limit in the second type-specific parameter).

At boot, an IO capability for the entire port space is passed to the initial user domain. Aside from being copied or minted, IO capabilities may not be created.

2.4 Global Descriptor Table (GDT)

The GDT table is loaded by the *gdt_reset* function during start-up and statically defined.

The table contains the following entries:

Index	Description
0	NULL segment
1	Kernel code segment
2	Kernel stack segment
3	User stack segment
4	User code segment
5	Task state segment
6	Task state segment (cont.)
7	Local descriptor table
8	Local descriptor table (cont.)

2.5 Interrupts and Exceptions

The initial (Interrupt Descriptor Table) IDT is set-up by *setup_default_idt* in `irq.c`. The number of entries in the IDT is set to 256 entries which are initialized in the following way:

Index	Description
0	Divide Error
1	Debug
2	Nonmaskable External Interrupt
3	Breakpoint
4	Overflow
5	Bound Range Exceeded
6	Undefined/Invalid Opcode
7	No Math Coprocessor
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid TSS
11	Segment Not Present
12	Stack Segment Fault
13	General Protection Fault
14	Page Fault
15	Unused
16	FPU Floating-Point Error
17	Alignment Check
18	Machine Check
19	SIMD Floating-Point Exception
32 ⋮ 47	PIC Interrupts
48 ⋮ 61	Generic Interrupts
62	Tracing IPI
63	Tracing IPI
64 ⋮ 247	Unused
248	Halt IPI (Stopping a core)
249	Inter core vector (IPI notifications)
250	APIC Timer
251	APIC Thermal
252	APIC Performance monitoring interrupt
253	APIC Error
254	APIC Spurious interrupt
255	Unused

The lower 32 interrupts are reserved as CPU exceptions. Except for a double fault exception, which is always handled by the kernel directly, an exception is forwarded to the dispatcher handling the domain on the CPU on which it appeared.

Page faults (interrupt 14) are dispatched to the ‘pagefault‘ entry point of the dispatcher. All other exceptions are dispatched to the ‘trap‘ entry point of the dispatcher.

There are 224 hardware interrupts, ranging from IRQ number 32 to 255. The kernel delivers

an interrupt that is not an exception and not the local APIC timer interrupt to user-space. The local APIC timer interrupt is used by the kernel for preemptive scheduling and not delivered to user-space.

Unused entries will be initialized by a special handler function. The slots reserved for generic interrupts can be allocated by user-space applications.

2.6 Local Descriptor Table (LDT)

The local descriptor table segment in the GDT will initially point to NULL as no LDT is installed.

User-space applications can install their own LDT table which is loaded on context-switching using the `maybe_reload_ldt` function.

2.7 Registers

Segment registers Segment registers are initialized by the `gdt_reset` function during start-up and each of them points to a GDT entry (index of the GDT table slot for each segment is given in brackets).

cs Kernel code segment (1)

ds NULL segment (0)

es NULL segment (0)

fs NULL segment (0)

gs NULL segment (0)

ss Kernel stack segment (2)

We also note that the **`fs`** and **`gs`** segment registers are preserved and restored across context switches.

General purpose registers

- **`rcx`** contains the start address when running a dispatcher for the first time.

[*Floating point / SIMD*] [*Machine specific registers (MSR)*]

2.8 Hardware devices

2.8.1 Serial port

On x86, the serial device (a PC16550 compatible controller) is initialized for the first time by the BSP core on boot-up.

By default serial port `0x3f8` will be used, but the port can be changed by using a command line argument supplied to the kernel.

Notable settings for the serial driver are:

- Interrupts are disabled.
- FIFOs are enabled.
- No stop bit.
- 8 data bits.
- No parity bit.
- BAUD rate is 115200.

The serial device is later re-initialized into a different state once the serial driver takes over the device. For example, interrupts will then be enabled and handled by the driver.

2.8.2 PIC – Programmable Interrupt Controller

[*describe*]

2.8.3 xAPIC – Advanced Programmable Interrupt Controller

[*describe*]

2.8.4 System call API

This section describe the architectural system calls that are not common with other architectures.

7 SYSCALL_X86_FPU_TRAP_ON: Turn FPU trap on (x86)

8 SYSCALL_X86_RELOAD_LDT: Reload the LDT register (x86_64)