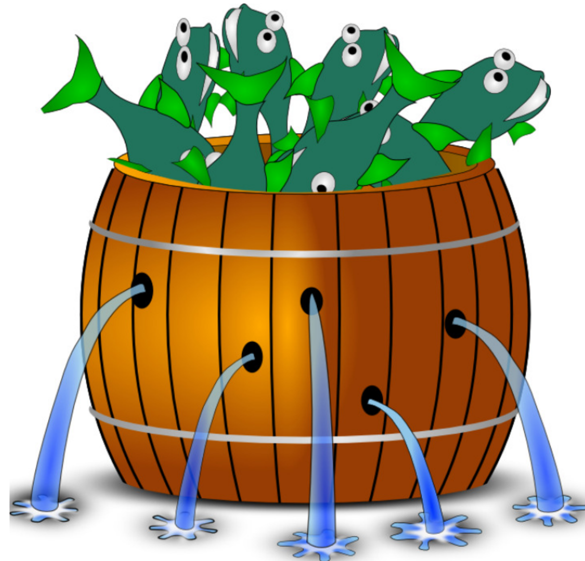
Barrelfish Project
ETH Zurich



Barrelfish Specification

Barrelfish Technical Note 10

Barrelfish project

01.01.2015

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	01.01.2009	AB,SP,TR,AS,AK	Initial Version
0.2	01.01.2015	GZ	Update ABI etc.

Contents

1 Barrelfish Kernel API **4**

1.1 System Calls 4

1.1.1 SYSCALL_INVOKE – Capability Invocation Interface 4

1.1.2 SYSCALL_YIELD – Yield the CPU 5

1.1.3 SYSCALL_DEBUG – Debug system calls 5

1.1.4 SYSCALL_REBOOT – Reboot the system 7

1.1.5 SYSCALL_NOP 7

1.1.6 SYSCALL_PRINT 8

1.1.7 SYSCALL_SUSPEND 8

1.1.8 SYSCALL_GET_ABS_TIME 8

1.2 Dispatch and Execution 8

1.2.1 Disabled 8

1.2.2 Register save areas 8

1.2.3 Dispatcher Entry Points 10

1.2.4 Interrupt delivery 10

1.2.5 Exception delivery 11

1.3 Scheduling 11

1.4 TODO 12

2 Barrelfish Library API **13**

2.0.1 Initial Capability Space 13

Chapter 1

Barrelfish Kernel API

1.1 System Calls

The section defines the specification of the common system call API that is provided by a Barrelfish CPU driver. Currently we have the following system calls:

SYSCALL_INVOKE	Invoke a capability.
SYSCALL_YIELD	Yield the CPU.
SYSCALL_LRPC	Fast LRPC.
SYSCALL_DEBUG	Benchmarking and debug syscalls.
SYSCALL_REBOOT	Reboot the machine.
SYSCALL_NOP	No operation.
SYSCALL_PRINT	Write to console.
SYSCALL_SUSPEND	Suspend the CPU.
SYSCALL_GET_ABS_TIME	Get time elapsed since boot.

1.1.1 SYSCALL_INVOKE – Capability Invocation Interface

The invoke call acts as a generic system call to apply operation on various OS objects (also known as capabilities). For any given object, a distinct set of operations are applicable depending on the capability type.

This system call takes at least one argument, which must be the address of a capability in the caller's CSpace. The remaining arguments, if any, are interpreted based on the type of this first capability.

Other than yielding, all kernel operations including IDC are (or should be) provided by capability invocation, and make use of this call. The possible invocations for every capability type are described in the capability management document (TN-013).

This system call may only be used while the caller is *enabled*. The reason is that the caller must be prepared to receive a reply immediately and that is only possible when enabled, as it requires the kernel to enter the dispatcher at the IDC entry point.

1.1.2 SYSCALL_YIELD – Yield the CPU

This system call yields the CPU. It takes a single argument, which must be either the CSpace address of a dispatcher capability, or `CPTR_NULL`. In the first case, the given dispatcher is run unconditionally; in the latter case, the scheduler picks which dispatcher to run.

This system call may only be used while the caller is *disabled*. Furthermore, it clears the caller's *disabled* flag, so the next time it will be entered is at the run entry point.

1.1.3 SYSCALL_DEBUG – Debug system calls

The debug system call (SYSCALL_DEBUG) de-multiplexes using the second system call argument and is defined for the following operations. Those calls may not be supported, depending on build options, and are not part of the regular kernel interface.

DEBUG_CONTEXT_COUNTER_RESET

Sets the `context_switch_counter` to 0.

DEBUG_CONTEXT_COUNTER_READ

Returns `context_switch_counter`.

DEBUG_TIMESLICE_COUNTER_READ

Returns `kernel_now`.

DEBUG_FLUSH_CACHE

Executes *wbinvd* on x86-64.

DEBUG_SEND_IPI

Sends an interrupt to a remote core.

Arguments

- **destination** Target core.
- **shorthand** ?
- **vector** IRQ number.

Note

Is this needed with the IPI capability?

DEBUG_SET_BREAKPOINT

Sets a hardware breakpoint at an address.

Arguments

- **addr** Where to break.

mode ?
length ?

Note
Use dr7 and dr0 on x86-64.

DEBUG_SEND_NOTIFY

Does only exist as a definition?

DEBUG_SLEEP

Does only exist as a definition?

DEBUG_HARDWARE_TIMER_READ

Returns *tsc_read*.

Note
Exists only for ARM.

DEBUG_HARDWARE_TIMER_HERTZ_READ

Returns *tsc_get_hz*.

Note
Exists only on ARM.

DEBUG_HARDWARE_GLOBAL_TIMER_LOW

Returns *gt_read_low*. The lower 32 bits of the timer.

Note
Exists only in OMAP, and returns 0 on GEM 5.

DEBUG_HARDWARE_GLOBAL_TIMER_HIGH

Returns global timer *gt_read_high*. The higher 32 bits of the timer.

Note
Exists only in OMAP, and returns 0 on GEM 5.

DEBUG_GET_TSC_PER_MS

Returns TSC (*rdtsc*) clock rate in ticks per ms.

Note
Implemention for x86 only.

DEBUG_GET_APIC_TIMER

Returns the XAPIC timer counter.

Note
Implemention for x86-64 only.

DEBUG_GET_APIC_TICKS_PER_SEC

Returns ticks per seconds of the APIC timer.

Note
Calibrated against RTC clock. Implemention for x86-64 only.

DEBUG_FEIGN_FRAME_CAP

Fabricates an arbitrary DevFrame cap.

Note
Implemention for x86-32 bit only. Not used?

DEBUG_TRACE_PMEM_CTRL

Enables tracing for capabilities.

Arguments
- **`types`** ?
- **`start`** ?
- **`size`** ?

Note
Implemention for x86-64 and aarch64 only.

DEBUG_GET_APIC_ID

Returns the xAPIC ID.

Note
Implemention for x86-64 only.

1.1.4 SYSCALL_REBOOT – Reboot the system

This call unconditionally hard reboots the system. [*This call should be removed -AB*]

1.1.5 SYSCALL_NOP

This call takes no arguments, and returns directly to the caller. It always succeeds.

1.1.6 SYSCALL_PRINT

This call takes two arguments: an address in the caller's vspace, which must be mapped, and a size, and prints the string found at that address to the console. It may fail if any part of the string is not accessible to the calling domain.

1.1.7 SYSCALL_SUSPEND

[*should probably be a cap invocation*]

1.1.8 SYSCALL_GET_ABS_TIME

[*Figure out proper time API, they appear in various DEBUG syscalls as well.*]

1.2 Dispatch and Execution

A dispatcher consists of code executing at user-level and a data structure located in pinned memory, split into two regions. One region is only accessible from the kernel, the other region is shared read/write between user and kernel. The fields in the kernel-defined part of the structure are described in Table 1.1.

Beyond these fields, the user may define and use their own data structures (eg. a stack for the dispatcher code to execute on, thread management structures, etc).

1.2.1 Disabled

A dispatcher is considered disabled by the kernel if either of the following conditions is true:

- its disabled word is non-zero
- its program counter is within the range specified by the crit_pc_low and crit_pc_high fields

The disabled state of a dispatcher controls where the kernel saves its registers, and is described in the following subsection. When the kernel resumes a dispatcher that was last running while disabled, it restores its machine state and resumes execution at the saved instruction, rather than upcalling it at an entry point.

1.2.2 Register save areas

The dispatcher structure contains enough space for three full copies of the machine register state to be saved. The trap_save_area is used whenever the dispatcher takes a trap, regardless of whether it is enabled or disabled. Otherwise, the disabled_save_area is used whenever the dispatcher is disabled (see above), and the enabled_save_area is used in all other cases.

Figure 1.1 (Trap and PageFault states have been left out for brevity) shows important dispatcher states and into which register save area state is saved upon a state transition. The starting state for a domain is "notrunning" and depicted with a bold border in the Figure.

Table 1.1: Dispatcher control structure

Field name	Size	Kernel R/W	Short description
disabled	word	R/W	If non-zero, the kernel will not upcall the dispatcher, except to deliver a trap.
haswork	pointer	R	If non-zero, the kernel will consider this dispatcher eligible to run.
crit_pc_low	pointer	R	Address of first instruction in dispatcher's critical code section.
crit_pc_high	pointer	R	Address immediately after last instruction of dispatcher's critical code section.
entry points	4 function descriptors	R	Functions at which the dispatcher code may be invoked
enabled_save_area	arch specific	W	Area for kernel to save register state when enabled
disabled_save_area	arch specific	R/W	Area for kernel to save and restore register state when disabled
trap_save_area	arch specific	W	Area for kernel to save register state when a trap or a pagefault while disabled occurs
recv_cptr	capability pointer	R	Address of CNode to store received capabilities of next local IDC into
recv_bits	word	R	Number of valid bits within recv_cptr
recv_slot	word	R	Slot within CNode to store received capability of next local IDC into

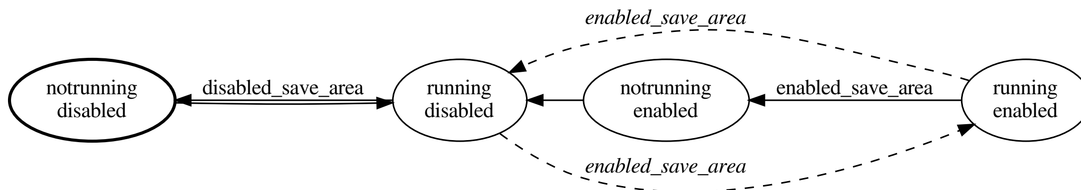


Figure 1.1: Dispatcher state save areas. Trap and PageFault states omitted for brevity. Regular text and lines denote state changes by the kernel. Dashed lines and italic text denote state changes by user-space, which do not necessarily have to use the denoted save area. The starting state is in the bold node.

Arrows from right to left involve saving state into the labeled area. Arrows from left to right involve restoring state from the labeled area. It can be seen that no state can be overwritten. The kernel can recognize a disabled dispatcher by looking at the disabled flag, as well as the domain's instruction pointer. Nothing else needs to be examined.

The dispatcher states are also depicted in Figure 1.2.

1.2.3 Dispatcher Entry Points

Unless restoring it from a disabled context, the kernel always enters a dispatcher at one of the following entry points. Whenever the kernel invokes a dispatcher at any of its entry points, it sets the disabled bit on. One (ABI-specific) register always points to the dispatcher structure. The value of all other registers depends on the entry point at which the dispatcher is invoked, and is described below.

The entry points are:

Run A dispatcher is entered at this entry point when it was not previously running, the last time it ran it was either enabled or yielded the CPU, and the kernel has given it the CPU. Other than the register that holds a pointer to the dispatcher itself, all other registers are undefined. The dispatcher's last machine state is saved in the enabled_save_area.

PageFault A dispatcher is entered at this entry point when it suffers a page fault while enabled. On entry, the dispatcher register is set, and the argument registers contain information about the cause of the fault. Volatile registers are saved in the enabled_save_area; all other registers contain the user state at the time of the fault.

PageFault_Disabled A dispatcher is entered at this entry point when it suffers a page fault while disabled. On entry, the dispatcher register is set, and the argument registers contain information about the cause of the fault. Volatile registers are saved in the trap_save_area; all other registers contain the user state at the time of the fault.

Trap A dispatcher is entered at this entry point when it is running and it raises an exception (for example, illegal instruction, divide by zero, breakpoint, etc.). Unlike the other entry points, a dispatcher may be entered at its trap entry even when it was running disabled. The machine state at the time of the trap is saved in the trap_save_area, and the argument registers convey information about the cause of the trap.

LRPC A dispatcher is entered at this entry point when an LRPC message (see below) is delivered to it. This can only happen when it was not previously running, and was enabled. On entry, four registers are delivered containing the message payload, one stores the endpoint offset, and another contains the dispatcher pointer.

This diagram shows the states a *dispatcher* can be in and how it gets there. The exceptional states Trap and PageFault have been omitted for brevity.

1.2.4 Interrupt delivery

Hardware interrupts are delivered by the kernel as asynchronous IDC messages to a registered dispatcher. A dispatcher can be registered as for a specific IRQ by invoking the IRQTable

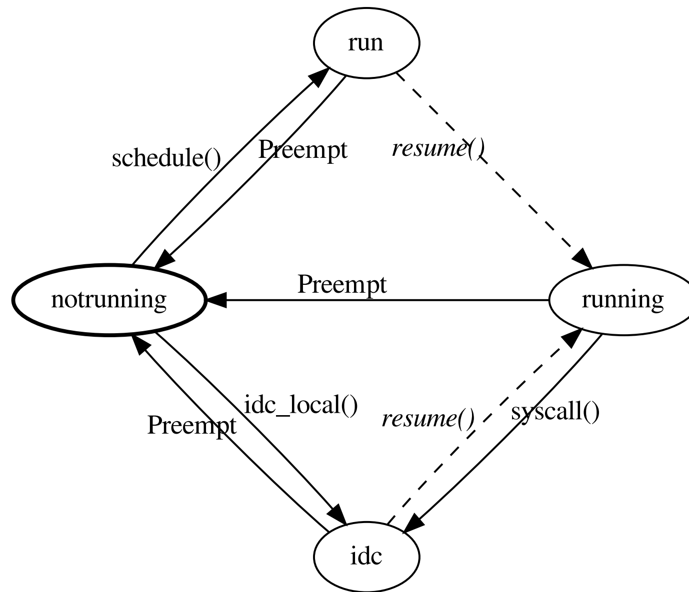


Figure 1.2: Typical dispatcher states. Trap and PageFault states omitted for brevity. Regular text and lines denote state changes by the kernel. Dashed lines and italic text denote state changes by user-space. The starting state is in bold.

capability, passing it an IDC endpoint to the dispatcher and the IRQ number. It is not possible for multiple IDC endpoints to be registered with the same IRQ number at any one time.

Henceforth, the kernel will send an IDC message using asynchronous delivery to the registered endpoint. Asynchronous IDC is used as it does not cause priority inversion by directly dispatching the target dispatcher.

1.2.5 Exception delivery

When a CPU exception happens in user-space, it is reflected to the dispatcher on which it appeared. Page faults are dispatched to the page-fault entry point of the dispatcher. All other exceptions are dispatched to the trap entry point of the dispatcher. The disabled flag of the dispatcher is ignored in all cases and state is saved to the trap save area.

1.3 Scheduling

Upon reception of a timer interrupt, the kernel calls 'schedule()', which selects the next dispatcher to run. At the moment, a simple round-robin scheduler is implemented that walks a circular singly-linked list forever. [*RBED, gang-scheduling*]

1.4 TODO

- virtual machine support
- timers
- resource management
- thread migration
- event tracing / performance monitoring

Chapter 2

Barrelfish Library API

[*Documentation of libbarrelfish*]

2.0.1 Initial Capability Space

The initial capability space of other domains is similar, but lacks the other cnodes in the root cnode, as illustrated in Figure 2.1.

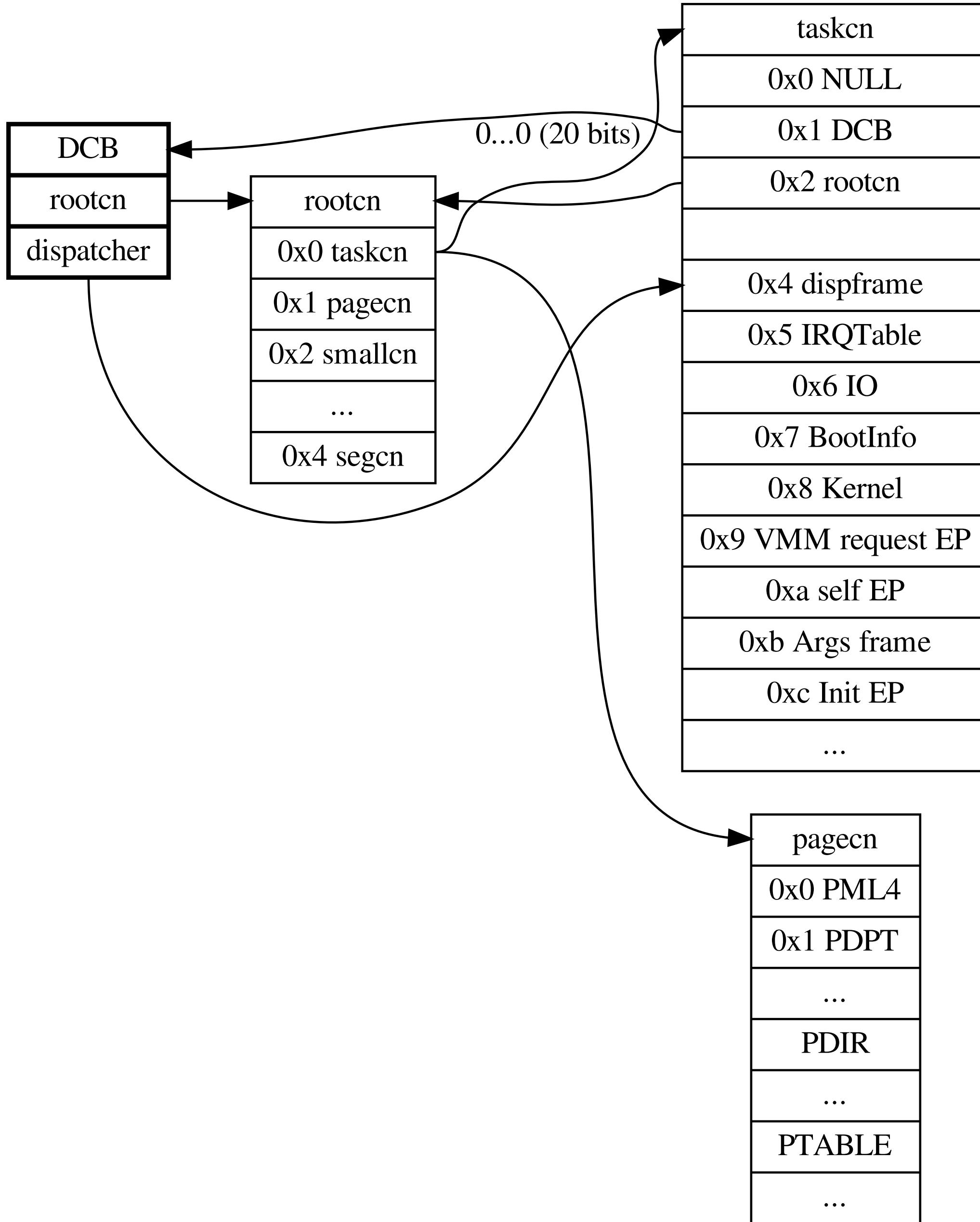


Figure 2.1: initial capability space layout of user tasks

Acknowledgements

Paul, Rebecca, Tim, et al.