Acknowledgements

Paul, Rebecca, Tim, et al.
Contents

1. Introduction 7

2. Barrelfish Kernel API 8
   2.1. System Calls 8
       2.1.1. Invoke 8
       2.1.2. Yield 8
       2.1.3. Debug system calls 8
   2.2. Dispatch and Execution 9
       2.2.1. Disabled 9
       2.2.2. Register save areas 9
       2.2.3. Dispatcher Entry Points 11
   2.3. Inter-Dispatcher Communication 12
       2.3.1. Endpoints 12
       2.3.2. Message Transfer 13
       2.3.3. Capability transfer 13
       2.3.4. Interrupt delivery 14
       2.3.5. Exception delivery 14
   2.4. Virtual Memory 14
   2.5. Initial Address Space 14
       2.5.1. User-Space Perspective 14
       2.5.2. Kernel Perspective 17
   2.6. Scheduling 17
   2.7. TODO 17

3. Barrelfish Library API 19
   3.1. Capabilities 19
       3.1.1. Data types 19
       3.1.2. Functions 19
       3.1.3. Invocations 19
       3.1.4. Syscalls 19
   3.2. VSpace management 19
   3.3. Dispatch and threading 19
   3.4. Spawning domains 19
       3.4.1. Initial capability space 19

A. Glossary 21
B. Implementation 23
   B.1. Mapping Database .................................................. 23
   B.1.1. Implementation details .......................................... 23
   B.1.2. Invariants ......................................................... 24
   B.1.3. Current Limitations ............................................. 24

C. Architecture-Specific Features 25
   C.1. x86-64 ................................................................. 25
   C.1.1. VSpace ............................................................... 25
   C.1.2. IO capabilities .................................................... 25
   C.1.3. Interrupts and Exceptions ...................................... 26
List of Tables

2.1. Dispatcher control structure ................................. 10
List of Figures

2.1. Dispatcher state save areas .............................................. 11
2.2. Typical dispatcher states ................................................. 12
2.3. init’s initial capability space layout ................................... 16
3.1. initial capability space layout of user tasks ........................ 20
1. Introduction

Barrelfish is...
2. Barrelfish Kernel API

2.1. System Calls

2.1.1. Invoke

This system call takes at least one argument, which must 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 provided by capability invocation, and make use of this call. The possible invocations for every capability type are described in the capability management document.

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.

2.1.2. Yield

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.

2.1.3. Debug system calls

The following debug system calls may also be supported, depending on build options, but are not part of the regular kernel interface.

No-op

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

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

Debug
[TODO: document me]

2.2. 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 2.1.

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).

2.2.1. 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.

2.2.2. 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 2.1 (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.

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 2.2.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Size</th>
<th>Kernel R/W</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>word</td>
<td>R/W</td>
<td>If non-zero, the kernel will not upcall the dispatcher, except to deliver a trap.</td>
</tr>
<tr>
<td>haswork</td>
<td>pointer</td>
<td>R</td>
<td>If non-zero, the kernel will consider this dispatcher eligible to run.</td>
</tr>
<tr>
<td>crit_pe_low</td>
<td>pointer</td>
<td>R</td>
<td>Address of first instruction in dispatcher’s critical code section.</td>
</tr>
<tr>
<td>crit_pe_high</td>
<td>pointer</td>
<td>R</td>
<td>Address immediately after last instruction of dispatcher’s critical code section.</td>
</tr>
<tr>
<td>entry points</td>
<td>4 function descriptors</td>
<td>R</td>
<td>Functions at which the dispatcher code may be invoked</td>
</tr>
<tr>
<td>enabled_save_area</td>
<td>arch specific</td>
<td>W</td>
<td>Area for kernel to save register state when enabled</td>
</tr>
<tr>
<td>disabled_save_area</td>
<td>arch specific</td>
<td>R/W</td>
<td>Area for kernel to save and restore register state when disabled</td>
</tr>
<tr>
<td>trap_save_area</td>
<td>arch specific</td>
<td>W</td>
<td>Area for kernel to save register state when a trap or a pagefault while disabled occurs</td>
</tr>
<tr>
<td>recv_cptr</td>
<td>capability pointer</td>
<td>R</td>
<td>Address of CNode to store received capabilities of next local IDC into</td>
</tr>
<tr>
<td>recv_bits</td>
<td>word</td>
<td>R</td>
<td>Number of valid bits within recv_cptr</td>
</tr>
<tr>
<td>recv_slot</td>
<td>word</td>
<td>R</td>
<td>Slot within CNode to store received capability of next local IDC into</td>
</tr>
</tbody>
</table>
2.2.3. 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
2.3. Inter-Dispatcher Communication

Inter-dispatcher communication (IDC) is a kernel-supported mechanism to allow dispatchers to communicate by sending messages. IDC is executed by invoking an IDC endpoint capability referring to a receiving dispatcher.

2.3.1. Endpoints

IDC communication takes place between dispatchers via endpoints. An endpoint is created by retyping a dispatcher capability into an IDC endpoint capability. It refers to exactly one dispatcher, and to one endpoint buffer structure within that dispatcher. An endpoint buffer is a kernel-specified data structure located within the dispatcher frame, where the kernel delivers IDC messages.
The kernel guarantees messages to either be delivered to the receiving dispatcher or to return to the sender with an error status code in the event that the receiver is unable to receive the message. This implies that messages are never dropped silently by the kernel but does not guarantee that messages are never dropped on the whole communication path, which involves the receiving dispatcher.

It should be noted that endpoint capabilities may be freely copied, and do not uniquely identify a sender. An endpoint capability can be transferred to several dispatchers, all of whom may use the same endpoint and thus the same buffer when sending messages.

2.3.2. Message Transfer

To send IDC, a dispatcher invokes an endpoint capability to the receiving dispatcher. The message it wishes to send is provided as argument to the invocation, as well as flags to specify additional parameters influencing the message transfer.

An IDC message is delivered by the kernel allocating space in the receiver’s endpoint buffer, and writing the message contents. The receiver must poll its endpoints to detect incoming messages, and consume them in order to free space in the endpoint buffer for new messages. [TODO: detail!]

A dispatcher that executes an IDC invocation is considered to have yielded the CPU while enabled. Therefore, the next time it is entered may be either at the Run or LRPC entry points.

2.3.3. Capability transfer

IDC can also be used to transfer capabilities from the sending dispatcher’s domain to a receiving dispatcher’s domain. [TODO: document cap transfer!]

Flags

If the “sync” flag is set and the message transfer succeeds, the kernel will immediately dispatch the receiver. Effectively, the sender yields to the receiver.

If the “yield” flag is set, and the message fails for one of the following reasons:

- the receiver’s message buffer is full
- the sender specified a capability, but it cannot be delivered because the receiver’s capability receive slot is non-empty

… then the kernel will also immediately dispatch the receiver (without performing a message transfer). Again, in this case the sender effectively yields to the receiver.

LRPC

[TODO: document!]

In this mode of IDC, the kernel performs a controlled context switch from the sending to the receiving dispatcher, preserving the capability invocation register state which is used to deliver
the message. The sender dispatcher is not blocked, however it implicitly donates the remainder of its timeslice to the receiver.

If the receiving dispatcher is disabled, and the “yield” flag was set, the kernel sets the return register in the sending dispatcher’s enabled_save_area to SYS_ERR_TARGET_DISABLED. The kernel then switches to and resumes the target dispatcher. In effect, an LRPC operation when the target is disabled becomes a directed yield of the CPU to the target dispatcher. If the “yield” flag was not set, the kernel simply returns the same error code to the sender and runs the sender.

2.3.4. 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 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.

Refer to Appendix C for more information about valid hardware interrupts for an architecture-specific implementation of Barrelfish.

2.3.5. 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 pagefault 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.

2.4. Virtual Memory

[TODO: Our memory model is based on capabilities and is quite similar to seL4.]

2.5. Initial Address Space

Our address space initialization is similar to the one of seL4, but we do not follow their boot protocol to the word. Here is our version:

We have a special program called init that is run by the kernel after bootup as an ELF64 executable. In order to function, it has to receive some information by the kernel. We show first how it receives this information from its (init’s) own perspective and then how the kernel gathers and transmits this information to init.

2.5.1. User-Space Perspective

init’s virtual address space size at startup is at most 4 MBytes (the amount of pagetable kernel memory left for it), mapped as 4K pages, starting from 0x200000 (2 Meg). init’s text/data
Listing 2.1: bootinfo structure

```c
struct bootinfo {
    // Base address of small memory caps region
capaddr_t small_untyped_base;
    // Number of small memory caps
    size_t small_untyped_count;
    // Base address of large memory caps region
    capaddr_t large_untyped_base;
    // Number of large memory caps
    size_t large_untyped_count;
    // Number of entries in regions array
    size_t regions_length;
    // Memory regions array
    struct mem_region regions [MAX_MEM_REGIONS];
};
```

Listing 2.2: mem_region structure

```c
struct mem_region {
    paddr_t base; // Address of the start of the region
    size_t size; // Size of region in bytes
    enum region_type type; // Type of region
    uint64_t data; // Additional data, based on region type
};
```

segments should be aligned consecutively and start at 0x400000 (4 Meg), leaving it 2 MBytes for its text and data.

We have a bootinfo structure, shown in Listing 2.1.

This structure is mapped into init’s virtual memory at address 0x200000 (2 Meg) and is at most a 4K page in size. small_untyped_base points to the capability to the CNode, holding a number (given by small_untyped_count) of small untyped capabilities. These can be used for easy setup of init’s own address space. large_untyped_base and large_untyped_count is similar for (much) larger untyped capabilities. Their sizes can be found in the regions array, of size regions_length entries. An entry is defined by a mem_region struct, shown in Listing 2.2.

Its fields should be self-explanatory. The possible region types are defined by enum region_type, shown in Listing 2.3.

These are the same as those in seL4.

**Initial Capability Address Space**

init’s initial CSpace is shown in Figure 2.3.
Figure 2.3: init's initial capability space layout
Listing 2.3: region_type enumeration

```c
enum region_type {
    RegionType_Empty,     // Empty memory
    RegionType_InitCaps,  // init’s caps mapped here
    RegionType_RootTask,  // Code/Data of init itself
    RegionType_Device,    // Memory-mapped device
    RegionType_CapsOnly   // Kernel-reserved memory
};
```

2.5.2. Kernel Perspective

In the following, ‘cn’ will be short for CNode, init_dcb is short for init’s DCB, replyep is short for init’s system call reply endpoint. The kernel sets up init’s domain as follows:

- It allocates physical pages for: rootcn, taskcn, smallcn, supercn, init_dcb, replyep.
- Map bootinfo, init_dcb, replyep, rootcn, pml4, pdpt, pdir and ptables in that order.
- Allocate 64 physical pages and put untyped caps to them into smallcn.
- Map taskcn, smallcn, supercn, in that order.
- Setup init’s DCB.
- Load init ELF64 binary into memory, map memory and allocate caps.
- Add all remaining memory as untyped caps to power of two large regions into supercn. They may not be more than 64.
- Fill bootinfo struct.
- schedule () init.

2.6. 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.

2.7. TODO

- virtual machine support
- timers
- resource management
• thread migration
• event tracing / performance monitoring
3. Barrelfish Library API

[TODO: documentation of libbarrelfish]

3.1. Capabilities

3.1.1. Data types

cap_info cnode_info
  get_cap_valid_bits get_cap_addr get_cnode_valid_bits get_cnode_addr build_cnode_info

3.1.2. Functions

cap_copy cap_mint cap_retype cnode_create cnode_create_raw
  ram_alloc
  async_endpoint_create local_endpoint_create

3.1.3. Invocations

invoke_*

3.1.4. Syscalls

syscall sys_yield cap_invoke cap_invoke_wait

3.2. VSpace management

struct vnode struct vlist
  vspace_alloc vspace_map vspace_map_raw vspace_free vspace_alloc_map vspace_map_attr
  vspace_map_attr_raw

3.3. Dispatch and threading

3.4. Spawning domains

3.4.1. 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 3.1.
Figure 3.1.: initial capability space layout of user tasks
A. Glossary

**Capability** Every kernel object is represented by a capability, allowing the user who holds that capability to manipulate it. We use partitioned capabilities: capabilities are stored in memory accessible only to the kernel, and are manipulated or invoked through the use of addresses in the CSpace.

**CSpace** The capability address space, in which all capabilities reside, is constructed and managed by user-space code through a hierarchy of page table-like structures, called CNodes. The protection domain of user code is determined by the capabilities existing in its CSpace.

**VSpace** The virtual address space

**Dispatcher** Kernel-scheduled entity, responsible for scheduling/managing the execution of user code. Dispatchers are identified by DCB capabilities. Every dispatcher has a CSpace and VSpace pointer, which determine its protection domain and virtual address space. Multiple dispatchers may share a CSpace or VSpace.

**Domain** Although not directly part of the Barrelfish kernel API, the word domain is used to refer to the user-level code sharing a protection domain and (usually) an address space. A domain consists of one or more dispatchers.

**DCB** Dispatcher control block, the kernel object associated with a dispatcher, and therefore also one of the system’s capability types. [I’d prefer to avoid this term, as it can be confusing. -AB]

**IDC** Inter-dispatcher communication, the kernel-mediated message-passing primitive. There are two types of IDC: the general case of asynchronous IDC, and an optimised local IDC variant possible only when both the sender and receiver execute on the same core.

**Endpoint** A type of capability that, when invoked, performs an IDC. There are two endpoint types (asynchronous and local) to match the two types of IDC.

**Channel** A uni-directional kernel-mediated communication path between dispatchers. All messages travel over channels. Holding a capability for a channel guarantees the right to send a message to it (although the message may not be sent for reasons other than protection).

**Mapping Database** The mapping database is used to facilitate retype and revoke operations. A capability that is not of type dispatcher, can only be retyped once. The mapping database facilitates this check.
When a capability is revoked, all its descendants and copies are deleted. The mapping database keeps track of descendants and copies of a capability allowing for proper execution of a revoke operation.

Each core has a single private mapping database. All capabilities on the core must be included in the database.

**Descendant** A capability X is a descendant of a capability A if:

- X was retyped from A,
- or X is a descendant of A1 and A1 is a copy of A,
- or X is a descendant of B and B is a descendant of A,
- or X is a copy of X1 and X1 is a descendant of A.

**Ancestor** A capability A is an ancestry of a capability X if X is a descendant of A.
B. Implementation

This chapter covers the implementation and algorithm of some subsystems.

B.1. Mapping Database

This section describes the mapping database in more detail. It covers the algorithms including implementation details and invariants on the database.

B.1.1. Implementation details

The database implements the following functions:

- **is copy** Checks if two capabilities are copies of each other. Two capabilities are copies if they are of the same type and they refer to the same kernel object. PhysAddr, RAM, Frame, DevFrame, CNode, Dispatcher, Kernel, EndPoint Capability types explicitly reference kernel objects so capabilities of such types can be tested simply. We cannot handle other capability types yet, comparing two VNode, we always return false and comparing two IO or IRQTable, we always return true.

- **is ancestor** Checks if one capability is a parent of another. In our current implementation, some capability types cannot have descendants and some capability types cannot have ancestors. For the rest, we check if the parent child relationship is possible based on the retyping type allowed and check if the kernel object the child refers to is inclusive in the range of kernel objects the parent refers to.

- **has descendants** Checks if a capability has any descendants. Walks the entire database checking if the capability has any descendants. The function returns true when the first descendant is found and if a capability other than a copy is found, it returns false.

- **has copies** Checks if a capability has any copies. Walks the entire database checking if the capability has any copies. The function returns true when the first copy is found and if a capability other than a descendant is found, it returns false.

- **insert after** Inserts a set of contiguous capabilities after the given capability

- **insert before** Inserts a set of contiguous capabilities before the given capability

- **set init mapping** Inserts a capability into the database in the appropriate location. If any copies or ancestors of the capability exist, the capability is inserted after a copy or after the closest ancestor. If no relatives exist in the database, the capability is inserted at the top of the database.
• *remove mapping* Removes the capability from the database

**B.1.2. Invariants**

Some invariants on the database that must be true at all times.

1. The next and prev pointers on a capability are never NULL.

2. There is only one database per core. Any capability on a core can be reach from another on the core.

3. Two separate databases do not share any capabilities. The set of next and prev pointers on one database is disjoint from the set of next and prev pointers on another.

4. Each capability on a core is on the database of the core. A capability will eventually be visited by starting at any other capability and walking the database.

5. The database is circular. Walking in either direction from any capability, the same capability will eventually be reached again.

6. The head of the database cannot have any ancestors.

**B.1.3. Current Limitations**

The database has the following limitations.

1. It does not handle VNode capabilities and other types. Other types are not as crucial, but VNode will become a priority shortly.

2. No indexing for quickly inserting brand new capabilities. The implementation starts at the head of the database and traverses the entire database till an appropriate location is found.

3. Not necessarily a limitation but an important note. The current implementation can report certain capabilities as descendants when one could have been created by copying another and report a descendant as a copy when it was created by retyping the ancestor. This requires the implementation of some functions to test for relationships in the correct order. This may lead to some unforeseen issues later.
C. Architecture-Specific Features

This chapter covers features specific to one implementation of Barrelfish on a specific hardware architecture.

C.1. 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 [2] and [1] for the Intel 64 and AMD64 architectures, respectively.

C.1.1. VSpace

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.

C.1.2. 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 ??) 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.
C.1.3. Interrupts and Exceptions

Interrupts

The lower 32 interrupts are reserved as CPU exceptions. Thus, 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.

Exceptions

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.
Bibliography
