Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 147 b

Systems Group, Department of Computer Science, ETH Zurich

A Debugging Interface for Barrelfish

by

Marc Tanner

Supervised by

Prof. Timothy Roscoe
David Cock

February 2016–July 2016

inf | Informatik
Computer Science

Contents

Chapter 1

Introduction

Writing correct, bug-free code is a difficult task and rarely achievable on the first try. As Kernighan and Plauger noted in *The Elements of Programming Style*[15]:

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Hence program authors need as much assistance as possible to investigate the inevitable failures. While operating system (OS) developers have powerful tools such as external debuggers, simulators and emulators at their disposal, convenient debugging of user space applications requires tight integration with core OS services.

We introduce a debugging interface for the Barrelfish research OS, supporting a similar debugging environment to that commonly found on popular Unix systems. More concretely, it should be possible to debug Barrelfish applications using the GNU Debugger (GDB). Towards that goal, we provide mechanisms to configure breakpoints and watchpoints, inspect and modify memory regions as well as core CPU registers and support single instruction stepping.

While most of the concepts are architecture independent, our proof of concept implementation is targeted at the ARMv7 platform.

The work is structured as follows: Chapter 2 provides the necessary background information about Barrelfish, the available hardware debug features and communication interfaces with the host debugger. Chapter 3 discuses the design and implementation of the new debug subsystem which is then evaluated in chapter 4. Finally, we summarize our work and conclude with ideas for future work in chapter 5.

Chapter 2

Background

2.1 Barrelfish

In recent years, modern machines have featured an increasing number of cores integrated by a complex interconnect network. A computer thus resembles a local distributed system[7] made up of heterogeneous components. Barrelfish[1], a research OS primarily developed at ETH Zurich in collaboration with Microsoft Research and other industry partners, embraces this viewpoint and employs the Multikernel architecture[6].

The main idea is to run per-core independent OS instances which coordinate by explicit inter-core message passing. With very few exceptions, state is replicated rather than shared. This not only enables seamless support for heterogeneous systems, but also promises to yield performance improvements over more traditional shared memory designs for large scale systems.

The Barrelfish architecture also shares similarities with an exokernel[12]. A small kernel (CPU driver) provides minimal hardware abstraction, enforces protection and performs authorization. Physical resource management (e.g. virtual memory) is delegated to user space where the system run-time library `libbarrelfish` provides a reasonable default implementation. This offers a great deal of flexibility, but as we will see also poses some challenges for a debugging subsystem.

As in a microkernel[18], message passing is used for communication among a number of unprivileged user space processes which provide device drivers and core system services for applications.

The remainder of this section introduces various Barrelfish concepts relevant to our work. It provides the necessary background information needed to understand the changes to core system components as detailed in chapter 3.

2.1.1 CPU Driver

Each core runs an independent, small, privileged kernel generally referred to as a *CPU Driver*. The kernel is single-threaded, non-preemptible and avoids dynamic memory allocations[20]. Core tasks of the CPU driver include[3]:

[1] `http://www.barrelfish.org/`

- Enforcing protection and performing authorization of kernel objects and physical resources by means of capabilities.
- Providing secure access to core hardware such as the MMU or the debug components.
- Scheduling of dispatchers on the local core.
- Fast local messaging using a mechanism similar to Lightweight RPC[8] or L4 IPC[19].
- Processing of hardware interrupts and subsequent delivery to user space device drivers.
- Handling of faults and system calls from user space tasks.

Most services of the CPU driver are requested by user space through capability invocations performed by means of a system call interface. For our work, the CPU driver is extended to securely manage all hardware debug resources. In order to inspect running process state and configure hardware break- and watchpoints, new interfaces will be introduced in section 3.2.

2.1.2 Monitor

CPU drivers are completely self contained and do not communicate with each other. Instead they are complemented by monitors: specially trusted user space processes, responsible for inter-core coordination. All monitor instances form an inter-core network, which is used to synchronize state replication in a multi core system. Agreement protocols are used to maintain consistency properties across core boundaries. As such monitors implement large parts of the low-level OS functionality, typically found in a monolithic kernel, in a distributed manner[3].

Monitors play an important role in system startup. They start per-core instances of important system services and provide them with the required capabilities. Furthermore, they are instrumental in bootstrapping interprocess communication. Upon domain creation each dispatcher is initiated with a communication channel to its local monitor.

2.1.3 Capabilities

Capabilities[17] are used for access control to all kernel objects, physical memory and hardware resources. Barrelfish uses a partitioned capability scheme similar to that of seL4[16]. A *capability* is a typed memory area only available to the local CPU driver. Multiple capabilities can be stored in a *capability node* (CNode). A *capability space* (CSpace) is made up of a set of CNodes stored in a guarded page table structure[20]. User space only indirectly refers to capabilities by means of *capability references* which are looked up relative to a per-dispatcher CSpace. Given such a reference, certain pre-defined capability operations can be invoked through a system call interface.

Capabilities are typed and can be retyped based on a fixed derivation hierarchy[1]. Hamlet, a Domain Specific Language (DSL) implemented using Filet-o-Fish[11], is used to specify the in-memory representation as well as the capability

type system. The CPU driver enforces the type system and makes sure only authorized changes are granted.

Except for inter-core communication purposes, capabilities are generally created by the kernel during system boot-up and can then be retyped or split. At system start, the kernel places capabilities in well known CNode slots in the CSpace of `init`. Init then copies them to the `monitor` which in turn propagates specific capabilities to core system services.

Inter core capability transfer and more generally consistency among capability spaces across cores is maintained by the trusted monitors.

As we will see in section 3.2, the capability type system was extended with new capability types which model the available hardware debug resources. The monitor was modified to hand over all necessary debug related capabilities to a new user space debugging service.

2.1.4 Virtual Memory Management

Barrelfish uses a variant of self-paging[14] to handle virtual memory management. User space domains are responsible for maintaining their own virtual address space. All memory management is performed through appropriate capability invocations where the CPU driver ensures correctness.

More concretely, a task allocates RAM capabilities, retypes them to platform-specific page table capabilities and inserts them into its root page table by means of a suitable capability invocation. Similarly, RAM capabilities can be retyped to mappable frame capabilities which are then inserted into a page table to construct a virtual address space[6].

The run-time library `libbarrelfish` provides a default implementation suitable for most use cases. It also handles page faults delivered by the kernel through an upcall mechanism[10].

Most relevant to our work, the debug subsystem will have to provide a capability based mechanism to inspect and manipulate the virtual address space of a foreign process.

2.1.5 Domains and Dispatchers

In Barrelfish a process context is made up of a *domain* which contains a number of *dispatchers* (typically one per-core the domain runs on). Dispatchers represent a core bound scheduler entity. In a single-core environment, they resemble the concept of a process in a more traditional operating system.

Address spaces can be shared among dispatchers in a domain, but CSpaces are dispatcher - and hence - core specific.

The CPU driver maintains a Dispatcher Control Block (DCB) which holds scheduler information, contains references to the root page table i.e. the virtual address space, capability space and communication endpoints[20].

A second structure is shared between the kernel and user space to coordinate user level threading, as will be discussed in the next section. It contains upcall entry points into user space to signal page faults, traps, inter-domain messages or the availability of a scheduler time slice.

2.1.6 User Level Threading

`libbarrelfish` implements an user level threading library on top of the dispatcher abstraction provided by the CPU driver using a form of scheduler activation[2].

For debugging purposes, a thread is a particular interesting object because it represents an independent flow of control using a dedicated stack as well as a set of core registers. Therefore it is crucial to understand the interaction between user level and kernel mode code involved in thread scheduling.

A dispatcher can either be *enabled* or *disabled* depending on the type of code it currently executes. Enabled mode refers to the fact that the dispatcher currently runs application code, while disabled indicates that the dispatcher is executing book keeping code inside the threading library itself.

Each state has an associated *save area*, in a structure shared between the kernel and user space, where core registers can be safely stored when a dispatcher is preempted.

When a disabled dispatcher is resumed, the kernel restores the previously saved register contents and execution continues inside the threading library.

When an enabled dispatcher is resumed, the CPU driver changes its mode to disabled and then upcalls into the `run` handler. The user level thread scheduler can now decide whether it wants to resume the preempted user level thread or switch to another runnable thread. In the latter case, the register contents of the preempted user level thread currently found in the enabled save area are copied to a user space Thread Control Block (TCB). In both cases the dispatcher is switched to enabled state immediately before application code is resumed.

A critical observation relevant to our work is, that the CPU driver only has access to the register contents of the most recently preempted user space thread. In the context of debugging, the register set of a currently inactive dispatcher can indirectly be inspected and modified by manipulating its respective save area.

2.1.7 Inter-dispatcher Communication

A number of core system services in Barrelfish are provided by user space daemons exporting a remote procedure call (RPC) interface. Therefore inter-dispatcher communication is crucial. Barrelfish uses the concept of point to point channels to transfer typed messages defined in a interface definition language called Flounder[5]. System services *publish* an interface to a name server. Clients can then query the nameserver for an interface reference to which they can *bind* to establish a shared channel. The generated Flounder stubs take care of marshalling, message dispatching as well as message fragmentation and reassembly. Furthermore, the Flounder stub compiler can target different interconnect driver (ICD) backends, thus providing a common interface over different transport mechanism.

Message delivery on the same core is implemented by locale message passing (LMP) based on the concepts of L4 IPC[19].

Communication crossing core boundaries is performed over a cache-coherent memory region using a variant of user level RPC[9]. Intimidate knowledge of the cache coherency protocol is exploited to transfer cache-line sized frames between

cores without kernel intervention. This ICD is referred to as User-level Message Passing (UMP)[20].

Our debugging service (see section 3.3) also exposes its functionality using a Flounder based RPC interface. Similarly, integration with the user mode serial driver uses the same communication primitives. LMP is explicitly used by the CPU driver to inform interested user space domains about the occurrence of hardware debug events.

2.1.8 Domain Creation and Process Management

Barrelfish currently only has a very primitive form of process or domain management[4]. During system boot up, `init` starts the `monitor` which typically starts a number of core system services including `startd` and `spawnd`. The former spawns further boot modules. While the latter publishes a Flounder RPC interface to spawn user domains. Furthermore, `spawnd` maintains a process list, allocates domain IDs and provides an interface to *kill* a running domain.

All the previously mentioned services, which start domains in one way or another, have one thing in common: they delegate the task of creating a dispatcher, initializing its virtual address and capability space, loading and relocation of ELF image to two system libraries `libspawndomain` and `libelf`.

While discussing domain creation in a debugging context, we will see that our user space debugging service re-uses the same basic infrastructure.

2.1.9 Serial Subsystem

The serial console is one of the main input/output channels used to access a running Barrelfish system.

The bootstrap processor (BSP) runs a user space serial driver (`/usr/serial`) which exports both a *basic* and a *terminal* service. The former provides a remote procedure call (RPC) interface to read/write data and register a callback to be invoked whenever input is available. The latter integrates with the `libterm_server` library to provide terminal session support[13]. The used `libc` (a variant of newlib) has been modified such that the `stdio(3)` library functions use `libterm_client` to coordinate I/O by means of inter-dispatcher communication.

The CPU driver bypasses the user space serial subsystem completely and directly interferes with the UART device to print various diagnostic messages. Interleaving of messages from different cores is prevented by using a shared spinlock. This is one of the very few cases where the CPU drivers use locking to protect shared state.

Coordinating access to the serial console is important for our work because it not only carries regular system I/O, but potentially also serves as a communication channel to the host debugger.

2.1.10 Angler and Fish

Angler is a session initialization manager roughly comparable to `getty(8)` as typically found on a Unix system. At system startup, it starts a new terminal session and then spawns `fish`, the Barrelfish shell. Fish provides a simple

command line interface to navigate the file system, launch new user domains and inspect system state.

2.1.11 Mackerel

Device drivers often contain a lot of tedious and error prone bit twiddling code to manipulate hardware state. Barrelfish improves upon this hand written code by employing Mackerel[21], a Domain Specific Language (DSL) used to specify the in-memory format of registers and other data structures dictated by hardware interfaces.

Given such a hardware description, the Mackerel compiler generates a C header file with inline functions to access, manipulate and pretty print register values. By default, Mackerel assumes that device communication takes place through a memory mapped area. However the concept of *address spaces* enables Mackerel to integrate with almost any device interface. A driver developer only has to provide a set of matching C functions for raw device access. These pluggable backends improve portability by abstracting away different hardware access mechanisms.

In the context of our work, Mackerel is used by the CPU driver to access the hardware debug components.

2.2 ARM v7 Debug Architecture

This section summarizes the relevant portions of the ARM v7 Debug Architecture as specified in Part C of the *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*, introduces the required nomenclature and describes the software interface to the available hardware debug facilities.

2.2.1 Debugging Modes

The ARM architecture provides both a *non-invasive* and an *invasive* debug type. While the former allows the observation of data and program flow (e.g. by means of trace support, profiling and performance monitors), modification of the main processor state, as required for run-control debugging, is restricted to the latter.

The invasive debug component supports two run-time configurable modes of operation. In *halting debug-mode* any debug event will immediately halt the complete processor and defer any further action to an external debugger. More relevant to our work is *monitor debug-mode* where software debug events trigger debug exceptions which can be handled by the operating system.

The invasive debug features can be used to compromise system security. As a consequence the debug architecture specifies complex interactions with the Security Extensions. For the scope of this work we assume that the Security Extensions are disabled and non-secure invasive debug mode is enabled. All further discussion assumes the debug hardware to be configured for monitor debug-mode.

2.2.2 Software Interface to the Debug Hardware

A software interface to the debug architecture is exposed through a set of *debug registers*. These registers can be accessed by means of a co-processor 14 (CP14) interface. Co-processor register contents can be read and written with the `mrc` and `mcr` assembly instructions, respectively. In both cases, the register number is encoded in the instruction operands. Optionally a subset of the debug registers can also be made available through a memory-mapped interface.

2.2.3 Breakpoints and Watchpoints

The `BKPT` instruction can be used for software breakpoints. Upon its execution an unconditional BKPT instruction debug event is generated.

Hardware breakpoints are supported by 2 to 16 Breakpoint Register Pairs (BRP) which comprise of a Breakpoint Control Register (BCR) and a Breakpoint Value Register (BVR). A BRP can be associated to another BRP thereby forming a linked BRP. For a breakpoint event to occur the condition of *both* involved BRPs has to be fulfilled. Multiple BRPs can be linked to the same BRP.

Each BCR can be set to one of the following non-exhaustive breakpoint types:

- Linked instruction address match: the instruction address value equals the value in the BVR.
- Linked instruction address mismatch: the instruction address value *does not* equal the value in the BVR.
- Linked Context ID match: the Context ID equals the current value of the `CONTEXTIDR` register made of a 24 bit Process Identifier and 8 bit Address Space Identifier. Not all, but at least 1 BCR supports this type.

We now illustrate the necessary steps to configure a hardware breakpoint matching an instruction virtual address (IVA) in a particular context. Two different BRP, i and j, are used. The former is concerned with the IVA match, while the latter restricts the match to the given context. The association between the two BRP is formed by linking the BRPi to BRPj. The BCRi is configured for linked instruction address match and the corresponding BVRi contains the instruction address to match. Furthermore, BCRj is set to linked Context ID match and BVRj holds the Context ID. Such a configuration is depicted in figure 2.1.

Both software and hardware breakpoint events generate a Prefetch Abort exception. This in turn executes the corresponding exception handler as registered in the exception vector.

Analogously, hardware watchpoints are defined by a Watchpoint Register Pair (WRP) comprised by a Watchpoint Control Register (WCR) and a Watchpoint Value Register (WVR). A WRP can be restricted to a certain process and address space by linking it to a BRP configured for linked Context ID match. A WCR can be used to match a whole memory range by configuring byte address selection and masking. For our work we will only ever use matching on a properly aligned machine word sized region stored in the WVR.

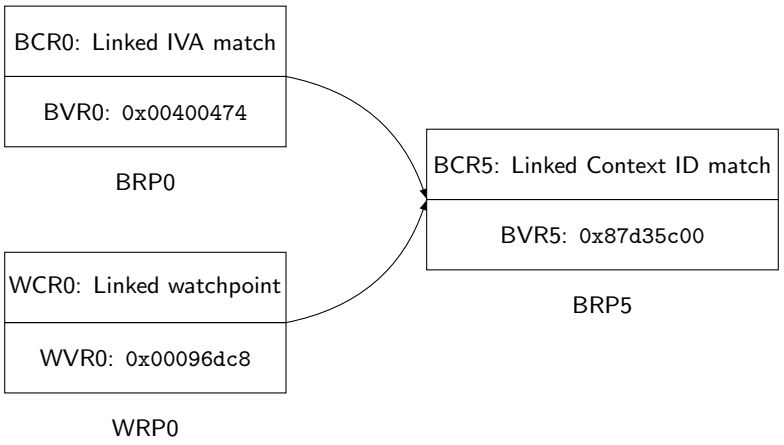

Figure 2.1: Linked matches: both a breakpoint (BRP0) and watchpoint (WRP0) is configured to match in a particular context defined by BRP5.

If all conditions are met, a Data Abort exception is raised and further actions take place according to the state in the exception vector. As such the behavior is similar to a page fault.

The exact debug event can be recovered by decoding the relevant fields of the `IFSR` (Instruction Fault Status Register), `DFSR` (Data Fault Status Register) and `DBGDSCR.MOE` (Debug Status and Control Register).

2.3 Pandaboard ES Debug Features

For our proof of concept we chose the Pandaboard ES[2] Rev B1 which features a Texas Instruments OMAP4460 SoC with a cache-coherent dual core ARMv7a Cortex A9.

The Cortex A9 conforms to the ARM v7 Debug architecture, as described in the previous section, and implements all CP14 registers. It provides:

- 6 Breakpoint Register Pairs, 2 of which support Context ID matches
- 4 Watchpoint Register Pairs

Hence a maximum of 5 hardware breakpoints can be set simultaneously in a single process context.

The Pandaboard also features a JTAG connector for whole system debugging with an external debugger.

It also provides a single RS-232 port connected to the UART2 interface of the OMAP4460 SoC which is used for regular console output. Additional UARTs are available, but by default not wired up to an external board interface.

[2] `http://www.pandaboard.org/`

2.4 GDB Remote Serial Protocol

The GNU Debugger[3] (GDB) is a powerful debugger often used on Unix systems and developed in conjunction with the GNU Binutils. Besides native debugging, where both the debugger and the application under debug are running on the same operating system, it also features a remote debugging mode.

Much of the heavy lifting (e.g. disassembling) is performed on the *host* machine while only a limited set of operations are performed on the *target* device:

- Read/Write memory regions
- Read/Write core registers
- Insert/Remove breakpoints
- Insert/Remove watchpoints
- Process state control

Although Barrelfish provides a limited POSIX compatibility library, porting GDB to run natively on Barrelfish would be a major undertaking and out of scope for our work. Therefore we target a remote debugging scenario.

2.4.1 Packet Format and Protocol Conventions

The GDB Remote Serial Protocol used for communication between the host `gdb` process and the target `gdbserver` is introduced in this section. The closest to a protocol specification can be found in Appendix E of the *Debugging with gdb* manual[23]. We use the same notation for host to target communication (and vice versa) indicated by `->` and `<-`, respectively.

Communication is packet based, the basic format is:

```
$packet-data#checksum
```

A packet always starts with a `$` followed by the variable length packet data which is eventually terminated by a `#`. This end marker is immediately followed by a two digit hexadecimal encoded checksum computed as the unsigned 8bit sum modulo 256 of every packet data byte. In the packet data any occurrence of the special symbols $x \in \{\texttt{\$}, \texttt{\#}, \texttt{*}, \texttt{\}}\}$ has to be escaped as `}`y where $y = x$`^0x20` (`^` denotes a bitwise XOR). By default, all packages are acknowledged positively `+` or negatively `-`, the latter of which should trigger a re-transmission. Packets can contain fields separated by either one of `;`, `,` or `:`. Binary data is generally transmitted in a hexadecimal encoding. The default number format is also hexadecimal, but leading zeros are suppressed. Responses can optionally also use a run-length encoding scheme to save space in the transferred message payload. Unsupported packages must not be dropped, but acknowledged with an empty response: `$#00`. Successful completion of a request is generally signaled with an `OK` data reply packet. In case of an error some packages use a two digit error code `E`*nn*, however "that number is not well defined". In practice some implementations use suitable constants (i.e. those smaller than 256) from `errno(3)` as specified by POSIX.

[3] `https://www.gnu.org/software/gdb/`

For illustration purposes we provide a short protocol transcription showing how single stepping is achieved. As previously explained, the leading arrows indicate the direction in which packets are sent. Only the packet data payload is shown, for clarity the packet header and checksum are omitted. Lines printed in italic and starting with `//` are comments, and not part of the data stream.

```
// receive packet for single stepping
-> s
// send acknowledgment
<- +
// time passes, instruction is executed
// send a stop reply signal (process has halted)
<- S05
// receive acknowledgment
-> +
// read register 1
-> g1
<- +
// hexadecimal encoded register value
<- ffffffff
-> +
```

Listing 2.1: Extract of a GDB remote serial protocol transcription for instruction single stepping.

Chapter 3

Design and Implementation

This chapter discusses design decisions and implementation choices made while developing the debugging interface for Barrelfish. The required changes to each system component are reviewed.

3.1 Debug System Architecture Overview

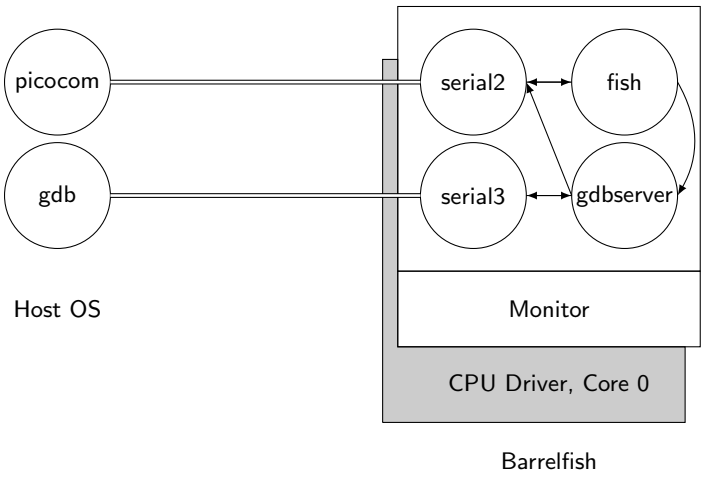

Figure 3.1: Debug system architecture overview (= represents a serial line, ↔ denotes Flounder based RPC communication channels).

Figure 3.1 illustrates the debug related system components and their respective communication channels. The privileged CPU driver exposes a capability based mechanism to configure the hardware debug unit. A number of regular user space domains (pictured as circles) run on top of the kernel. gdbserver implements the GDB remote serial protocol and coordinates debugging of Barrelfish application domains. Fish, the command shell, is used to initiate a new debugging session. Communication with the host operating system is established by two independent instances of a user level serial device driver. Two

distinct UART ports are used to provide separate channels for regular system I/O and the GDB remote serial protocol, respectively.

3.2 CPU Driver Changes

This section focuses on the necessary kernel changes needed to support a user space debugging service. Conceptually the following functionality has to be provided:

- Run-control of the process under debug (e.g. process resumption).
- Read/Write core registers of a different dispatcher.
- Read/Write memory regions of a different virtual address space.
- Configuration of hardware debug resources (e.g. break- and watchpoints).
- Debug Event delivery to a user space debugger.

Process state control, access to the core registers and an LMP based debug event delivery mechanism are provided by new dispatcher capability invocations. Remote memory manipulations are enabled by exposing a query interface to the kernel internal capability mapping database. A new invocation on the root page table capability provides a mean to retrieve a frame capability corresponding to a particular virtual address. The capability type system has been extended to cover the needs of the available hardware debug resources.

3.2.1 New Capability Types

Two new capability types were defined using Hamlet:

```
cap BreakReg_ARM {
    uint8 id;          /* register number 0-15 */
};

cap WatchReg_ARM {
    uint8 id;          /* register number 0-15 */
};
```

Listing 3.1: New capability definitions in Hamlet.

They model a Breakpoint Register Pair (BRP) and a Watchpoint Register Pair (WRP) (see section 2.2 for details), respectively. More concretely, one such capability authorizes access to both the Breakpoint Control Register (BCR) and its associated Breakpoint Value Register (BVR).

We currently do not support multiple simultaneous debugging session, hence the in-memory layout of these capabilities is extremely simple: they only contain a register identifier. The actual register contents are always re-read directly from the hardware when needed.

The generic kernel startup code was modified to create two new capability nodes (CNodes) in known slots of the root CNode. These CNodes are then dynamically filled with register capabilities in a platform specific debug initialization routine, based on the number of available debug registers.

```
errval_t platform_debug_init(coreid_t core_id,
                             struct cte *break_cn,
                             struct cte *watch_cn);
```

The `BreakReg_ARM` capability type supports the following capability invocations to configure a hardware breakpoint within a particular virtual address space:

- `link(struct capref reg_cap, struct capref disp_cap)`

  It associates or links a BRP `reg_cap` to a dispatcher capability, meaning that the BRP will be configured for linked Context ID match. As Process ID we currently use the 24 most significant bits of the dispatcher (`struct dcb`) memory address. The 8 bit Address Space Identifier is currently not used and always set to zero (all TLB entries are untagged).

- `set(struct capref brp_cap, struct capref lbrp_cap lvaddr_t break_addr, enum breakpoint_flags)`

  This configures the BRP referenced by `brp_cap` to generate a debug event, in the context referred to by a second BRP `lbrp_cap`, at the instruction virtual address (IVA) specified by `break_addr`. The second BRP must previously have been set up for a linked Context ID match. The last argument indicates the break point type: match or mismatch and its status: enabled or disabled.

Analogously, the `WatchReg_ARM` capability type supports a similar `set` invocation to configure a WRP:

- `set(struct capref wrp_cap, struct capref lbrp_cap lvaddr_t watch_addr, enum watchpoint_flags)`

  This configures the WRP referenced by `wrp_cap` to generate a debug event in the context referred to by the BRP `lbrp_cap` (which must be a previously linked BRP) at the virtual address `watch_addr`. The last argument can be used to specify the type of watchpoint: read or write and its status: enabled or disabled.

3.2.2 New Capability Invocations for Existing Capability Types

When a hardware debug event occurs, the user space debugger needs to be notified. It then inspects the process state and will eventually resume execution. For these reasons, the existing dispatcher capability type gained the following new debug related capability invocations:

- `attach_debugger(struct capref disp_cap, struct capref ep_cap)`
- `detach_debugger(struct capref disp_cap)`

  Un/register a local endpoint to which debug events associated with the given dispatcher will be sent to in form of LMP messages.

- `resume(struct capref disp_cap, lvaddr_t pc)`

  Only valid if the dispatcher is currently in enabled state. Sets the program counter of the enabled save area to `pc` and makes the dispatcher runnable i.e. adds it to the run queue. While the system call returns immediately, the resumed dispatcher will be scheduled some time in the future, depending on current scheduler state. If the given `pc` is zero, the program counter remains unchanged and execution will resume at the current location.

- `read_register(struct capref disp_cap, int reg, uint32_t *value)`
- `write_register(struct capref disp_cap, int reg, uint32_t value)`

  Read/write a given register as indexed into the enabled save area of the dispatcher. `reg` must refer to one of the core registers as defined in `barrelfish_kpi/registers_arch.h` i.e. in range $0 - 16$. These invocations are only valid for enabled dispatchers. If applied to a dispatcher in state disabled, an error code is returned.

The debugger needs to be able to inspect and modify memory regions of the process being debugged. As outlined in the section about virtual memory management, we need to get hold of a frame capability to the underlying memory region which can then be mapped into the virtual address space of the debugger. While it would be possible to replicate the virtual address resolution in user space by mapping and walking the various levels of page tables, the kernel already has all the required information in its capability mapping database. Therefore it was deemed simpler to add new capability invocation to the VNode capability type representing a root page table:

- `resolve(struct capref cap, genvaddr_t vaddr,`
  `capaddr_t dest_cn, int dest_bits, capaddr_t dest_slot)`

  Given a root page table capability: resolve a virtual address `vaddr` and store a frame capability in the capability slot referenced by the last three parameters.

3.2.3 Debug Exception Handling and Delivery

As described in section 2.2.3, a debug event generated by a hardware breakpoint or watchpoint will cause a prefetch or data abort, respectively.

The Barrelfish CPU driver sets up the exception vector in such a way that both of these eventually end up in a common handler routine. This function was modified to check whether the cause of the exception is in fact a debug event. If not, the regular user space self-paging code inside `libbarrelfish` is invoked via an upcall.

Once a debug event is detected:

1. It is checked whether the currently running dispatcher control block (DCB) has an associated debug endpoint to which debug events should be delivered to. If no endpoint has been registered by using the `attach_debugger` capability invocation, all breakpoints for this dispatcher are cleared. The domain remains runnable and regular control flow resumes.

2. The currently running dispatcher is removed from the scheduler run-queue.

3. The concrete type of the debug event (breakpoint or watchpoint) is determined. A two word sized LMP message of the form

$$\{type, faulting\ address\}$$

   is delivered to the registered debug endpoint. This re-uses the same mechanism responsible for IRQ delivery to user space device drivers.

4. The scheduler is invoked to run some other dispatcher. Eventually the user space debug service will get a time slice and process the LMP message.

If a debug event occurs while in kernel mode, the kernel dumps diagnostic information and then panics. This does not concern our user level code debugging, hardware breakpoints and watchpoints are configured to only affect user mode.

3.2.4 Debug Hardware Access

The debug hardware configuration, exposed through the previously introduced capability invocations, is performed by means of a Mackerel device definition (see also section 2.1.11).

On the Pandaboard we chose the Extended Co-Processor 14 interface to access the debug registers. For this purpose, a new header file `cp14.h`, containing a number of inline functions implementing the low level device access, is introduced. As an example we will illustrate configuration of a Breakpoint Control Register (BCR). Notice that the register number is encoded in the operands of the assembly instruction:

```
static inline void cp14_write_dbgbcr(int bcr, uint32_t val)
{
    switch (bcr) {
    case 0:
        __asm volatile("mcr p14, 0, %0, c0, c0, 5" ::
                       "r" (val));
        break;
    case 1:
        __asm volatile("mcr p14, 0, %0, c0, c1, 5" ::
                       "r" (val));
        break;
    // up to register 15
    }
}
```

Listing 3.2: Low level debug register manipulation routine.

In Mackerel the in-memory format for a new register type (`dbgbcr`) was added. Each such register type is accompanied with a corresponding Mackerel address space to hook up the raw co-processor access functions.

```
space cp14_dbgbcr(idx) valuewise "CP14 access for DBGBCR";
regarray dbgbcr cp14_dbgbcr(0x00) [16] "Breakpoint Control
    Registers (DBGBCR)" type(dbgbcr);
```

Listing 3.3: Mackerel address space definition for a BCR.

We conclude this section with an example of a C code snippet illustrating the setting of a few register fields. More concretely, BCR 0 is linked to BCR 5.

```
// hook up raw device access functions
#define arm_debug_cp14_dbgbcr_write_32(dev, reg, val) \
        cp14_write_dbgbcr((reg), (val))
// include generated Mackerel device definition
#include <dev/arm_debug_dev.h>

// initialize Mackerel device
arm_debug_t dev;
arm_debug_initialize(&dev);

int reg = 0;      // register to configure
int lbrp_reg = 5; // linked BRP

// change a couple of register fields
arm_debug_dbgbcr_t bcr = 0;
bcr = arm_debug_dbgbcr_enable_insert(bcr, 1);
bcr = arm_debug_dbgbcr_linked_brp_insert(bcr, lbrp_reg);
// write register value back to register
arm_debug_dbgbcr_wr(&dev, reg, bcr);
```

Listing 3.4: Debug hardware configuration example using Mackerel.

Error prone, hand written bit twiddling code is avoided. Instead, a number of self documenting calls to generated inline functions are used. This device abstraction also allows to change the underlying hardware access mechanism. Controlling the debug hardware by means of a memory mapped interface would only require minimal changes to the Mackerel device definition and initialization routine.

3.3 gdbserver: a User Space Debugging Service

In user mode the main component is a new debugging service `/usr/gdbserver`. It exposes a Flounder interface to create and start a new domain under debug. `gdbserver` implements the GDB remote serial protocol as described in section 2.4 and enables debugging of Barrelfish application code through a host `gdb` instance. The low level transport mechanism is realized by connecting to the user space serial driver. In response to the received host GDB packets, the previously presented capability invocations are used to provide the necessary debug facilities.

3.3.1 Domain Creation

As alluded to in section 2.1.8, domain management in Barrelfish is currently rather primitive. In order to initiate a debugging session, we need to be able to create a dispatcher in a non-runnable state. The debugger also needs access to at least the dispatcher and root page table capabilities of the new process being debugged. Neither of these things are possible through the currently exposed RPC interface of `spawnd`.

To minimize changes to the existing system, it was decided to bypass `spwand` completely and instead directly link against `libspawndomain` for domain creation. While this approach duplicates some of the `spawnd` functionality, it is simpler because the debugger, as spawning domain, can hold on to all relevant capabilities.

3.3.2 Memory Access

The debugger needs access to the virtual address space of the process under debug. As we have seen in section 2.1.4, memory management in Barrelfish is performed by means of suitable capability invocations. Hence the debugger needs to gain access to a *frame capability* matching a given virtual address of a different address space. The exact procedure used to manipulate a foreign address space is:

1. Invoke `resolve` of the root page table capability (as created during domain startup) with the virtual address of interest. The CPU driver queries its internal book keeping data structures and if found, returns a matching frame capability referencing the physical memory area to which the virtual address currently maps to.

2. Map the frame capability into the address space of the debugger.

3. Perform the necessary inspections or modifications.

4. Unmap the frame capability from the address space of the debugger and release all associated resources.

3.3.3 Core CPU Register Access

Besides access to the virtual address space, the debug service also needs to be able to read and write core CPU registers. In Barrelfish this is further complicated by the use of scheduler activations and user level threading (see section 2.1.6).

The CPU driver is not aware of the user level threads. It only maintains per-dispatcher (*not* per-thread) *save areas* to temporarily store register contents. Therefore register access is restricted to the least recently running user space thread. Accessing the registers of a *disabled* dispatcher, i.e. one currently running code inside the user level threading library itself, is not supported.

These limitations stem from the fact that the underlying `read_register` and `write_register` invocations on the dispatcher capability always access the enabled save area of the referenced dispatcher. It is only guaranteed, that the register modifications take effect once the preempted thread resumes. However, resuming the dispatcher will trigger the `run` upcall and the user level thread scheduler might subsequently decide to run another thread.

3.3.4 Hardware Breakpoints and Watchpoints

Hardware breakpoints are of particular interest because besides regular instruction address match, they also support an instruction *mismatch* mode. This functionality is exploited to implement single instruction stepping. The idea is

to set a mismatch breakpoint at the current program counter location. As a result the CPU executes exactly one instruction, before a mismatch hit generates a debug event.

`gdbserver` has access to a number of hardware breakpoint and watchpoint register capabilities, located in well known slots of its capability space. Immediately after a debugging session is initiated, one of the BRPs is linked to the new dispatcher, thereby configuring it for linked context ID match. All further breakpoint manipulations will always refer to this linked BRP. This effectively restricts matches to the address space of the dispatcher being debugged. The same mechanism also applies for watchpoint registers. All remaining BRPs are configured in response to GDB set breakpoint packets.

3.3.5 Software Breakpoints

As the name indicates, software breakpoints do not need special hardware support. Instead they work by temporarily replacing parts of the executable code. The instruction at the desired breakpoint location is first stored to a safe scratch space and then replaced by a special *unconditional* breakpoint instruction. To remove an existing breakpoint, the swapping process is reversed i.e. the original instruction is restored. Care must be taken not to allow multiple software breakpoints at the same location. Otherwise, depending on the order in which the breakpoints are removed, the correctly restored original instruction might subsequently be overwritten again.

On ARM, with its fixed sized instruction set architecture (ISA), run time code modification is straight forward. Exactly one machine word is swapped out for a `BKPT` instruction. For architectures featuring variable length instruction encodings, the breakpoint instruction has typically the shortest possible representation and any remaining space can be filled with `NOP` instructions.

The actual run time code modifications are performed using the regular memory read/write primitives as described in section 3.3.2.

3.3.6 Debug Event Notification

After domain creation `gdbserver` invokes the `attach_debugger` capability invocation to register a local LMP endpoint on which the CPU driver can deliver any debug events associated with the newly created dispatcher.

We now illustrate the steps leading to such a breakpoint event and how it is subsequently handled:

1. The host GDB sends a continue (`c`) packet.
2. The reception of the packet is acknowledged by sending `+`.
3. The dispatcher is resumed and runs until it hits a breakpoint.
4. The CPU driver removes the dispatcher from the run queue and delivers an LMP message to notify the debugger.
5. The `gdbserver` receives the LMP message and sends a *stop reply packet* to the host debugger to signal that the process has halted.

3.3.7 Serial Communication

As illustrated in figure 3.1, `gdbserver` does not directly talk to the UART device, but instead uses the Flounder based RPC interface to connect to the user level serial driver. Upon startup it registers itself as a new input consumer, such that all incoming data is rerouted to it.

The actual GDB protocol implementation is written in a I/O agnostic way. Swapping out the underlying transport mechanism, used to carry the GDB packet stream, should be possible with reasonable effort.

3.3.8 RPC Interface and Fish Integration

Analogous to the `spawn_domain` remote procedure call (RPC) provided by `spawnd`, `gdbserver` exposes a Flounder interface to launch and debug new domains:

```
interface gdbserver "Interface to debug domains" {
    rpc debug_domain(in string path,
                     in char argvbuf[argvbytes],
                     in char envbuf[envbytes],
                     in uint8 flags,
                     out errval err);
};
```

Listing 3.5: Exported Flounder interface of gdbserver.

The blocking `debug_domain` RPC, creates a new domain running the specified application with the supplied arguments and environment settings. It instructs the debug service to take over control of the serial port for communication purposes with a host GDB process. The RPC invocation will block during the complete debugging session. A reply is only sent once either GDB detaches or the domain terminates.

The command shell `fish` was extended to feature a new built in command

```
$ gdbserver application [arguments]
```

which initiates a new debugging session.

3.4 Serial I/O Demultiplexing

In order to enable stable remote debugging, a reliable communication channel to the host system is needed. Due to its simplicity, we chose the serial line as communication medium for our proof of concept.

Unfortunately the Pandaboard does by default only expose one of its UART lines through an external connector. Hence regular system I/O and the GDB remote serial protocol are transmitted over the same physical channel, requiring some form of software demultiplexing on the host system.

As illustrated in section 2.1.9, all application code interfaces with the serial line through a dedicated user space serial driver, while the kernel interferes directly with the hardware when printing diagnostic messages. The debugging service - like any user space task - can be preempted at any time. As a result GDB packets and kernel messages can be arbitrarily interleaved.

The GDB remote serial protocol (see section 2.4.1) uses special characters `$` and `#` for packet framing. The data payload is followed by a simple single byte

checksum. Tools like `kdmx`[1] exploit this structure to distinguish GDB packets from unrelated output. It serves as a proxy for the host `gdb` by demultiplexing both data streams and making them available as two separate pseudo tty devices. This works reasonably well, as long as the two data sources only alternate. But once they interleave, the concept breaks down. At least in theory, GDB packets will be retransmitted until they eventually arrive uncorrupted. However, regular system output is discarded. To work around this issue the raw kernel output routine was modified to delimit the printed message with special marker sequences. `kdmx` was adapted accordingly to properly handle interleaved messages.

Unfortunately the resulting communication channel was not as reliable as expected. Therefore we decided to perform some slight hardware modifications to hook up another UART port of the Pandaboard to an external interface. While it would have been convenient to use a stock Pandaboard without any hardware changes, in the end a separate communication channel dedicated to debugging proved to be a simpler solution. Not only does it minimize the required changes to existing system components, it also enforces a clean separation between regular system I/O and debugging related communication messages.

On the software side we run a new instance of the user space serial driver `/usr/serial` to manage the additional UART port. It exposes exactly the same Flounder interface as its counterpart controlling the serial port used for regular system I/O. The only difference is, that it is published to the name server under a different name (`serial_debug0`). The user mode debugging service then binds to this instance for communication purposes with the host debugger.

[1] `http://elinux.org/Kdmx`

Chapter 4

Evaluation and Future Work

In this chapter we are critically reviewing some of the design decision leading to the current implementation. We discuss limitations of our approach and mention ideas for further work.

One of the most severe limitation of the presented debugging subsystem is, that it is currently not possible to debug an already running user domain. At the moment it is only possible to initiate a completely new debugging session by spawning a new domain. This is unfortunate, because often a problem is complex to reproduce and therefore inspecting a still running process would be beneficial.

More generally, process management in Barrelfish is still rather primitive. The fact that `gdbserver` bypasses `spawnd` for domain creation is a serious layering violation. Having different code paths for domain creation, depending on whether a domain is being debugged, seems dangerous. At best, it is just unnecessary code duplication, at worst, it is a source of slight differences in run-time behavior. Ideally, all process management tasks would be centrally managed by spawnd. This should include functionality to initiate a new debugging session, either by spawning a new domain or by attaching to an already existing one. Towards this goal, the Flounder interface served by spawnd should be extended to provide a way to get hold of the dispatcher and root page table capabilities needed for debugging purposes. Such a design would also allow spawnd to enforce a security policy, determining which process is able to debug another one. Because attaching to existing domains is currently not supported, this aspect was so far deliberately neglected by our work.

Debugging of multithreaded applications is another area where future work is needed. Currently the debugging framework has a dispatcher-centric viewpoint. It is only possible to access register values of the most recently preempted user level thread. Per-thread breakpoints are not supported, dispatchers are the kernels scheduler entity. We employ an all-stop mode, GDB's non-stop multi-threaded debugging[22] is not supported. These limitations arise from Barrelfish's current threading architecture. The CPU driver shares only parts of the dispatcher structure with the user level threading library, but is unaware of the purely user space based thread control blocks. While this design enables

a great deal of flexibility for the user level threading library, it also rules out any CPU driver based thread manipulations. A much closer coupling between the debugging service and `libbarrelfish` of the process being debugged would be needed to support full featured multi-threaded debugging.

The Barrelfish architecture encourages to limit CPU driver functionality to the absolute minimum. With this goal in mind, it would have been possible to manually translate a virtual address (of the process being debugged) to its physical counterpart by replicating the page table traversal logic in software. However the kernel already maintains the necessary information, not taking advantage of that would needlessly complicate the implementation. Adding new kernel code, in form of the `resolve` capability invocation, seems warranted in this case.

Efficiency has so far not been a main concern of our work. We chose simple solutions even though they incur slight performance penalties. A concrete example of this approach are the new capability invocations to read/write core register values as introduced in section 3.3.3. They currently operate on a single register at a time. As a consequence, inspecting the complete set of core registers requires numerous user/kernel space transitions.

Similarly, access to the virtual address space of the process being debugged (see section 3.3.2) could be made more efficient. Instead of constantly mapping and unmapping frame capabilities, gdbserver could employ a caching layer. The idea being, that the most commonly used memory regions remain available in the address space of the debugger.

Finally, we draw some high-level comparisons between the presented Barrelfish debugging interface and the `ptrace(2)`[1] process tracing system call, commonly found on more traditional Unix-like operating systems.

The different design philosophies with regard to core operating system architecture are reflected in these interfaces. Barrelfish chooses to only implement the absolutely necessary privileged mechanisms in the small CPU driver. Additional functionality, built on top of these basic building blocks, is as much as possible pushed into regular user space services. As a concrete example, we provide capability invocations to configure hardware breakpoints, but leave the implementation of single instruction stepping to a user level debugging service. Linux, as an example of a monolithic system, provides an in-kernel implementation of the same functionality (`PTRACE_SINGLESTEP`). A similar tendency can be observed in the way access to the virtual address space is granted. The ptrace API provides explicit ways to read/write machine word sized memory regions. Barrelfish's CPU driver on the other hand only provides a way to retrieve a matching frame capability. Actual access to the memory region is delegated to the user level debugging service.

Another distinction stems from the capability based resource management of Barrelfish. Capabilities can easily be copied and propagated. At least in theory, any domain with access to a dispatcher capability is able to inspect its running state. This is in contrast to the ptrace API, where process tracing is restricted to one process at a time. This limitation arises, because tracing temporarily changes the Unix process hierarchy: the tracer becomes the new parent process of the tracee. In Unix this is necessary because `waitpid(2)` is used to signal process changes back to the debugger.

[1] `http://man7.org/linux/man-pages/man2/ptrace.2.html`

Chapter 5

Conclusion

The presented debugging interface for Barrelfish enables developers to debug their application code using the GNU debugger (GDB) running on a host operating system.

Barrelfish's capability system was extended to cover debugging needs. New capability types have been introduced to model available hardware debug resources, such as breakpoint and watchpoint registers. The CPU driver was modified to handle hardware debug exceptions. Furthermore, it provides new mechanisms to control dispatcher state, inspect memory regions and register contents. A user space debugging service implements the GDB remote serial protocol, coordinates communication with GDB and uses the new capability invocations to implement the necessary low level debug functionality.

While the current design introduces abstractions to enable a basic debugging environment, more advanced features, such as control of user level threads, will require future work. Similarly, extending the debugging interface to support domains spanning core boundaries, poses a number of interesting challenges. To provide a more comprehensive debugging experience, closer integration with core system components will be required.

Bibliography

[1] Ihor Kuz Akhilesh Singhania and Mark Nevill. *Capability Management in Barrelfish*. Tech. rep. 013. Version 2.2. Systems Group, ETH Zurich, 2013. `http://www.barrelfish.org/publications/TN-013-CapabilityManagement.pdf`.

[2] Thomas E Anderson et al. "Scheduler activations: Effective kernel support for the user-level management of parallelism". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 53–79.

[3] Team Barrelfish. *Barrelfish Architecture Overview*. Tech. rep. 000. Version 2.0. Systems Group, ETH Zurich, 2013. `http://www.barrelfish.org/publications/TN-000-Overview.pdf`.

[4] Team Barrelfish. *Barrelfish OS Services*. Tech. rep. 012. Version 1.0. Systems Group, ETH Zurich, 2010. `http://www.barrelfish.org/publications/TN-012-Services.pdf`.

[5] Andrew Baumann. *Inter-dispatcher communication in Barrelfish*. Tech. rep. 011. Version 0.3. Systems Group, ETH Zurich, 2011. `http://www.barrelfish.org/publications/TN-011-IDC.pdf`.

[6] Andrew Baumann et al. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 29–44.

[7] Andrew Baumann et al. "Your computer is already a distributed system. Why isn't your OS?" In: *Proceedings of the 12th Workshop on Hot Topics in Operating Systems, Monte Verita, Switzerland* (2009).

[8] Brian N Bershad et al. "Lightweight remote procedure call". In: *ACM Transactions on Computer Systems (TOCS)* 8.1 (1990), pp. 37–55.

[9] Brian N Bershad et al. "User-level interprocess communication for shared memory multiprocessors". In: *ACM Transactions on Computer Systems (TOCS)* 9.2 (1991), pp. 175–198.

[10] David D Clark. "The structuring of systems using upcalls". In: *ACM SIGOPS Operating Systems Review*. Vol. 19. 5. ACM. 1985, pp. 171–180.

[11] Pierre-Evariste Dagand, Andrew Baumann, and Timothy Roscoe. "Filet-o-Fish: practical and dependable domain-specific languages for OS development". In: *ACM SIGOPS Operating Systems Review* 43.4 (2010), pp. 35–39.

[12] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*. Vol. 29. 5. ACM, 1995.

[13] Raphael Fuchs. "A session control interface for a Multikernel". In: *Bachelor thesis, Systems Group, ETH Zurich* (2012).
`http://www.barrelfish.org/publications/fuchsr-bachelor-sessioncontrol.pdf`.

[14] Steven M Hand. "Self-paging in the Nemesis operating system". In: *OSDI*. Vol. 99. 1999, pp. 73–86.

[15] Brian W Kernighan and PJ Plauger. "The elements of programming style". In: *New York: McGraw-Hill, 1974, 2nd ed.* 1 (1974).

[16] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[17] Henry M Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[18] J. Liedtke. "On Micro-kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 237–250. ISBN: 0-89791-715-4.

[19] Jochen Liedtke. "Improving IPC by kernel design". In: *ACM SIGOPS Operating Systems Review* 27.5 (1993), pp. 175–188.

[20] Barrelfish Project. *Barrelfish Glossary*. Tech. rep. 001. Version 2.0. Systems Group, ETH Zurich, 2013.
`http://www.barrelfish.org/publications/TN-001-Glossary.pdf`.

[21] Barrelfish Project. *Mackerel User Guide*. Tech. rep. 002. Version 1.5. Systems Group, ETH Zurich, 2013.
`http://www.barrelfish.org/publications/TN-002-Mackerel.pdf`.

[22] Nathan Sidwell et al. "Non-stop multi-threaded debugging in gdb". In: *GCC Developers Summit*. Vol. 117. 2008.

[23] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* (2016). Tenth Edition, for gdb version 7.11.50.
`http://sourceware.org/gdb/current/onlinedocs/gdb/`.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Debugging Interface for Barrelfish

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Tanner	Marc

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 1. August 2016	M. Tanner

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.