Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 45b

Systems Group, Department of Computer Science, ETH Zurich

A session control interface for a Multikernel

by

Raphael Fuchs

Supervised by

Prof. Dr. Timothy Roscoe, Simon Peter

August 31, 2012

inf | Informatik
Computer Science

Abstract

This thesis introduces a terminal subsystem for the Barrelfish multikernel. We demonstrate how character-based input-output streams can be implemented in a message-passing system.

Additionally, we introduce the notion of a session, show how sessions can be protected from each other by using capabilities and describe how they interact with the terminal subsystem. On top of that, we present mechanisms to start applications as part of a session, detaching them from a session or aborting an application.

In a case study, we port OpenSSH to Barrelfish demonstrating command-line interaction with the OS over a network connection.

Acknowledgments

I would like to thank Prof. Roscoe for giving me the opportunity to work on the Barrelfish operating system. His valuable insights helped me not to get lost in technical details but rather understand the wider picture.

The weekly meetings with Simon Peter motivated me to try out new ideas and his assistance and guidance were key factors in the completion of this thesis.

The feedback on my early design as well as on my code submissions from Andrew Baumann, Kornilios Kourtis and the other people from the Barrelfish mailing list helped to enhance the design and code considerably. I enjoyed writing my thesis in this environment.

Special thanks go to Lauren Pupillo, Jonas Moosheer and Antoine Kaufmann for reviewing my thesis and providing feedback.

Furthermore, I would like to thank my parents for their support over all the years.

Contents

Chapter 1

Introduction

1.1 Motivation

The increasing number of cores in commodity computers and the move towards more diverse systems in terms of memory hierarchies, interconnects and instruction sets poses major challenges to operating system design. *Andrew Baumann et. al.* proposed in 2009 [3] a new OS structure, the *multikernel* to cope with these challenges. Additionally, they present *Barrelfish*, an implementation of the multikernel concept.

At present, human users interact with the Barrelfish OS via a rudimentary shell and a serial console. While this "just works", it lacks flexibility and many basic features. More importantly, Barrelfish does not yet, in contrast to all major operating systems, support the concept of a *user session*.

This thesis improves on this and makes the following contributions:

- We present a general framework for character-based I/O streams and adapt the current system to use it for command-line interaction over the serial line.
- We introduce the notion of a session for the Barrelfish multikernel and show how protection between multiple concurrent sessions is achieved.
- We provide mechanisms for starting and terminating applications from the command line as well as attaching an application to and detaching an application from a session.
- We demonstrate command-line interaction over the network.

1.2 Overview

This thesis is structured as follows. The next chapter introduces relevant systems and concepts that are used throughout this thesis, and chapter 3 surveys related work. In chapter 4 we present the framework for character-based I/O streams and define the concept of a session. How these concepts are implemented and the changes to the rest of the system is presented in chapter 5. Chapter 6 provides a case study of how to use the framework over a network connection, while we evaluate our design and implementation by comparing it to a UNIX-like system in chapter 7. Finally, chapter 8 concludes and gives directions for future work.

Chapter 2

Background

2.1 Barrelfish

Barrelfish is a research operating system built from scratch and developed collaboratively by researchers from ETH Zurich and Microsoft Research. Since modern multi-core computers increasingly resemble a networked system [4], it is structured ground-up as a distributed system. The design follows that of a *multikernel* [3] with independent cores that communicate using messages and share no memory. The intention is to build a system that scales well as the number of cores increases and can exploit heterogeneous hardware resources.

2.1.1 CPU driver

Each core runs its own architecture-dependent kernel, called *CPU driver*. The CPU drivers do not share state with other CPU drivers nor do they communicate with each other. Among other things, it is responsible for enforcing protection, performing authorization and scheduling processes. Services from the CPU driver are invoked via the architecture specific system call mechanism.

2.1.2 Monitor

Each core also runs a distinguished user-mode process called *monitor*. It is responsible for all inter-core coordination and sets up interprocess communication. The monitors collectively keep replicated data structures such as the address space mappings globally consistent.

2.1.3 User domains and dispatchers

User level processes are called *user domains* or just *domains*. A user domain can span multiple cores and consists of typically one dispatcher per core it runs on [28]. A dispatcher is the unit of kernel scheduling and is an implementation of scheduler activations [1], i.e. it can schedule and run its own threads.

2.1.4 Inter-dispatcher communication

Communication in Barrelfish is not between processes but dispatchers. The kinds of messages allowed between two dispatchers is defined in an interface specification. The Flounder interface definition language is used for this purpose. At compile time this interface is translated into C code stubs, which the application programmer can use to send messages.

A server dispatcher can export an interface by sending a message to its local monitor, which responds by sending a unique interface reference (iref) back. This iref is usually registered at the central name service. After the client dispatcher looked up the iref of the service it is interested in, it uses the iref to bind to the interface [2].

2.1.5 Capabilities

Barrelfish uses capabilities [5] in order to provide access control to kernel objects. In a nutshell, a capability is a token that authorizes anyone possessing it to perform a specific set of actions on the kernel object associated with the capability. Only the kernel can directly access and modify the actual capability, user domains work with capability references.

The capability system of Barrelfish is modeled on that of seL4 [14]. In that model capabilities are held in a guarded capability table, which is similar to a guarded page table [16]. The nodes of the table are called CNodes and are themselves capabilities. A CNode has a fixed number of slots to hold other capabilities. A reference to the root CNode is held in the dispatcher. All CNodes together form the capability address space (CSpace).

2.1.6 Octopus

Recently, Barrelfish was extended with *Octopus*, a coordination service for processes or activities. It provides a key-value store and an API to set and retrieve records with the intention that applications can share a small amount

of configuration data. Octopus does not yet support access control for records. It is integrated into the *System Knowledge Base* (SKB) [30].

2.2 Terminals

There used to be a time when users interacted with computers using hardware terminals connected over the serial line. The computers were mostly mainframes that supported multiple concurrent users working at different hardware terminals. The VT100 from Digital Equipment Corporation was a prominent example of such a hardware terminal. It contained many control sequences that did not display a character on the screen but performed a specific action such as clearing the screen or moving the cursor. For example the sequence `ESC [ 3 D` moved the cursor three positions to the left. Such sequences travelled in the same byte stream over the serial line as printable characters did [8].

While hardware terminals are hardly used today, terminal emulators like xterm [7] or gnome-terminal [18] still emulate the exact same control sequences as the VT100 and its successors VT102 and VT220 did.

Chapter 3

Related Work

In the first section of this chapter we describe how other operating systems handle terminal I/O, followed by a section investigating how sessions are implemented.

3.1 Terminal I/O

The POSIX standard specifies the "General Terminal Interface" which most UNIX-like operating systems (Linux, FreeBSD, Mac OS X, Oracle Solaris) implement in various degrees of completeness [13].

The standard distinguishes canonical and non-canonical input processing. In canonical mode, input is processed in lines and a read request will return at most one line. Moreover, the operating system echoes the characters as they are typed by the user, so that he sees what he types. The user is able to correct typing errors and there exist special characters to erase the last word (typically `Ctrl+W`) or the entire line (typically `Ctrl+U`).

In non-canonical mode, input is not assembled into lines and special character processing is disabled. Editors often run in non-canonical mode. Since special characters like `Ctrl+W` are not processed by the operating system, the editor can assign its own meaning to them.

All the configuration options for terminal I/O are contained in the `termios` structure. We can get the `termios` structure containing the current settings using the function `tcgetattr`, modify the settings in the structure and apply the settings with a call to `tcsetattr`. The POSIX standard specifies more than 50 configuration options, and many implementations add about 30 non-standard options on top of that. [27, Chapter 18]

Linux, FreeBSD, Mac OS X and Oracle Solaris implement all of the above mentioned character processing in a module called terminal *line discipline*. [19, Chapter 10] In those operating systems there also exist line disciplines for the Point-to-Point Protocol (PPP) [26] and for using IP over a serial line. FreeBSD additionally has a netgraph line discipline and Mac OS X has one for serial tablets.

3.2 Session

In Unix-like systems, a session is defined as a collection of one or more process groups. A new session is established by a call to `setsid`. The calling process becomes the session leader of the new session and it is the only member of the session. Moreover, a session can have a controlling terminal. [27, Chapter 9]

Chapter 4

Approach

This chapter starts with the goals that guided the design of the proposed terminal subsystem followed by a general overview, describing the user domains and libraries that are involved as well as the flow of characters between them. Furthermore, we introduce the notion of a session and investigate what role a session plays in the terminal subsystem.

4.1 Design goals

4.1.1 Independence of device drivers

The terminal subsystem is designed in such a way that user domains reading characters need not to know where they came from. They could originate from the keyboard driver or arrive over the serial line, to name just two examples. Similarly, writing characters is independent of the destination of the characters, they could be drawn on the screen by a VGA driver or travel over the network to a remote host.

4.1.2 Compatibility with C standard input and output

The standard I/O library of the C programming language provides programmers with a set of higher-level functions compared to UNIX `read` and `write` for doing input and output. Especially the functions for formatted I/O, `printf` and `scanf`, are in ubiquitous use in C programs. The terminal subsystem should provide blocking terminal I/O functions that the overarching C library can use to implement its standard I/O library.

4.1.3 Flexibility

Since terminals are not exactly high-bandwidth devices, the design focuses more on flexibility than high performance. Flexibility allows to cope with the wide range of different terminals and terminal emulators that exist.

4.2 General overview

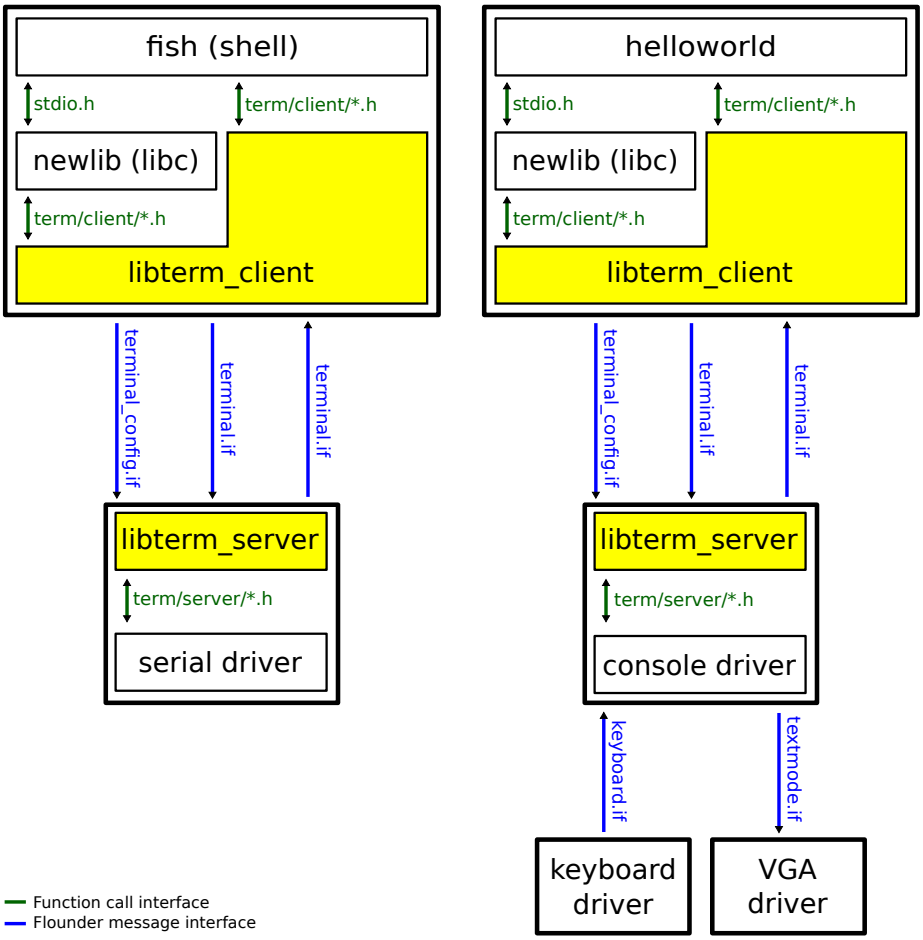

Figure 4.1: General overview of the terminal subsystem

Figure 4.1 depicts the general overview of the terminal subsystem. Black bold boxed indicate user domains while the boxes inside represent the user code and libraries that are relevant for the discussion. Communication across user domain is done using flounder messages which is represented in the figure by blue arrows and blue text indicating the name of the flounder interface. Communication

within a domain is done using a function call interface which is depicted in the figure as green arrows and the corresponding header file that contains the function prototypes.

User domains like the shell or a "Hello World" program can read and write characters in a way that is independent of the device driver that handles the low-level interaction with the hardware. To this end, we introduce the flounder interface `terminal.if` that is responsible for an unidirectional character stream. As listing 4.1 shows, it consists of a single message type that is used to send or receive a buffer of characters. We will refer to user domains like our shell *fish* or the "Hello World" program as *terminal clients*.

```
interface terminal "Unidirectional character stream." {

    /**
     * \brief Input or output of a character buffer.
     *
     * \param buffer Buffer holding characters.
     * \param length Amount of characters in the buffer.
     */
    message characters(char buffer[length]);
};
```

Listing 4.1: Flounder interface `terminal.if`

Terminal servers, on the other hand, are domains like the serial driver or the console driver that export the generic interfaces for the terminal clients. Each terminal server exports the interface `terminal.if` once for the incoming character stream from the terminal client and once for the outgoing character stream to the terminal client. Additionally, it exports the interface `terminal_config.if`, providing the terminal client with the means to send configuration messages to the terminal server. Terminal servers handle the received messages at these interfaces in a device driver dependent way.

The serial driver forwards characters received from the serial line to the terminal client. Likewise, it sends characters received from the client via the serial line.

While we will describe how a console driver fits into the terminal subsystem, this thesis does not provide an implementation for such a driver. We imagine that the console driver emulates a hardware terminal, e.g. a DEC VT200. It receives the keys, the user pressed on the keyboard, in the form of scancodes or hardware independent keycodes over the flounder interface `keyboard.if`. Subsequently, it maps them onto characters or control sequences and sends them to the terminal client. Output from the terminal client is drawn on the screen by the console driver. To this end, it can make use of a textmode VGA driver via a flounder interface `textmode.if`. Control sequences like the one mentioned

in the background chapter to move the cursor three positions to the left are translated by the console driver into the VGA command to reposition the cursor. Instead of using a VGA driver for the output, the console driver could make use of a framebuffer via the existing flounder interface `fb.if`.

We consider character streams end-to-end: from the terminal or terminal emulator that generates the stream until the terminal client that consumes it. The character stream is not modified nor interpreted in-between. The environment variable `TERM` in the terminal client indicates the type of the attached terminal and defines how the received character stream should be interpreted. Likewise the terminal client generates an output character stream that is tailored for the type of the attached terminal and is transported unmodified to the terminal or terminal emulator. The serial driver serves as a good illustration. It is located between the terminal client and the actual terminal that is attached via the serial line. Therefore, it does not modify the character stream but only maps between the two different transport mechanisms: serial communication and flounder messages. Moreover, the serial driver shows that the concept of a terminal server that we introduces is not to be confused with a terminal or terminal emulator. While the serial driver is a terminal server since it exports the generic terminal interface for the terminal clients, it is not a terminal or terminal emulator.

We introduce the library `libterm_client` that handles connection setup of a terminal client to a terminal server and handles much of the character processing providing the application programmer with a set of higher level APIs. Section 4.6 describes the API provided by this library in more detail. Moreover, the library `libterm_server`, described in section 4.7, provides higher level APIs that help the development of drivers that act as terminal servers.

4.3 ID capability

Before we define what a session is, let us introduce a new capability type called ID capability. The ID capability is a non-forgeable, system-wide unique ID. Every domain can generate as many ID capabilities as it pleases, each of which is distinct from any other ID capability created by the same or any other domain in the system. One can, however, not create an ID capability with a specific ID. Therefore, if a domain creates an ID capability only the domain itself and the domain it directly or indirectly passed this capability hold an ID capability with this specific ID.

4.4 Session

A *session* is a collection of user domains sharing a terminal for input and output and is represented by an ID capability. Session membership is determined by holding the session ID capability. To create a new session, we create a new ID capability and pass it on to all domains that should participate in the session. A domain that is part of a session is called *session domain*. Any domain can be part of at most one session at a time. Examples of session domains include the shell, the utility program `ls` or an editor.

Daemons on the other hand, are domains that run independent of a session and are not associated with a terminal. Examples of daemons include device drivers, the monitor or the system knowledge base (skb). We envision that daemons should use the system log to perform output and that anything written to the standard output is discarded. Since we do not yet have a system log in Barrelfish, everything written to the standard output is routed through `sys_print`, a system call to print a debug message.

Whether a domain is a session domain or a daemon is determined solely by how the domain is started. If it is started as part of a session, for example from a shell that is itself part of a session, it will be a session domain otherwise it will be a daemon. A session domain can always detach itself from the session by calling the function `daemonize()`, which is part of `libterm_client`.

4.4.1 Protection

Sessions are protected from each other, i.e. only domains belonging to a specific session can read to and write from the terminal associated with the session. To this end, the interface references for the incoming and outgoing character streams as well as the iref for the configuration interface of a terminal server are not registered at the name service since doing so would allow any domain knowing their name to retrieve the irefs. Instead, any terminal server exports an additional interface `terminal_session.if`. The iref of this interface is typically registered at the name service with a publicly known name. For example, the first instance of the serial driver registers the iref of its terminal session interface with the name `serial0.terminal` and the first instance of the console driver uses the name `console0.terminal`.

To associate a session with a terminal, the terminal session interface provides the rpc message type `session_associate_with_terminal` (see listing 4.2). It expects the session ID capability as input and returns the interface references of the incoming and outgoing character streams as well as of the configuration interface, which a terminal client can subsequently use to communicate with the

terminal server. As long as the terminal is associated with a session further calls to `session_associate_with_terminal` will return an error and no valid irefs. This ensures that only one session is associated with the terminal server at any time and that domains not belonging to a session are not able to obtain the irefs to communicate with the terminal server.

```
interface terminal_session "Terminal Session Interface" {

    /**
     * \brief Associate a terminal with a session.
     *
     * \param session_id ID capability representing the session.
     * \param in_iref    Interface reference to be used for incoming characters
     *                   as seen by the terminal client.
     * \param out_iref   Interface reference to be used for outgoing characters
     *                   as seen by the terminal client.
     * \param conf_iref  Interface reference to be used for configuration
     *                   messages.
     * \param err        SYS_ERR_OK if successful
     *                   TERM_ERR_TERMINAL_IN_USE if terminal is already part
     *                                            of another session
     */
    rpc session_associate_with_terminal(in cap session_id, out iref in_iref,
                                        out iref out_iref, out iref conf_iref,
                                        out errval err);
};
```

Listing 4.2: Flounder interface `terminal_session.if`

4.5 Startup process

In the general overview we have looked at the case of a terminal client that has an established connection to the terminal server and in the last section we have learned that the terminal servers only register the interface reference of their session interface at the name service. In this section we give a more complete overview of the steps and the additional domains that are involved. We use the serial driver as an example for a terminal server but the steps are analogous for any other terminal server.

Figure 4.2 illustrates all the relevant actions that typically happen during system startup until our shell *fish* is running and ready to receive input from the user.

When the serial driver starts up, it exports the terminal session interface and registers the assigned interface reference at the name service with the name `serial0.terminal`. It does not yet export any of the other terminal interfaces

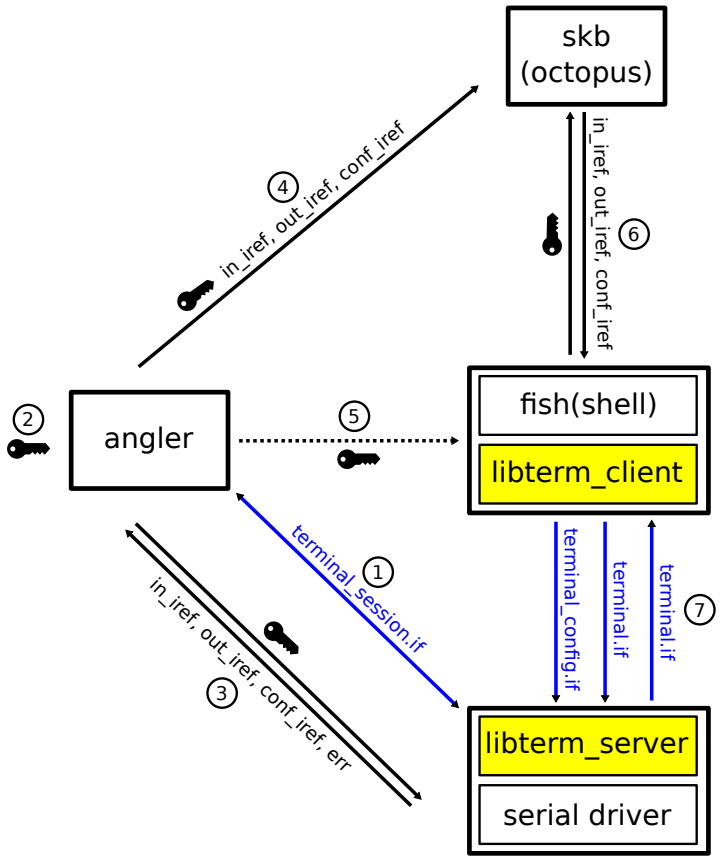

Figure 4.2: Startup process step by step

(step 1).

Angler is the session initialization manager and serves a similar purpose as getty [11] does in a UNIX system. It must be started with two arguments. The first argument specifies which terminal server we want to use for our session and the second, the type of the terminal. Let us assume that the characters we receive over the serial line originate from the terminal emulator `xterm`, then the two arguments for angler are `serial0.terminal` and `xterm`.

At startup angler creates a new ID capability to represent the session it is about to establish (step 2). In figure 4.2 the session ID capability is depicted with a key symbol. Thereafter, it looks up the interface reference of the terminal server supplied as the first argument to angler at the name service. After the binding of this interface completes, it sends the message `session_associate_with_terminal` (see listing 4.2) to the terminal server supplying the ID capability as an argument. If the terminal server is already associated with a session it replies with the error code `TERM_ERR_TERMINAL_IN_USE`. Otherwise, it exports

the three interfaces for the incoming character stream, the outgoing character stream and the configuration interface and replies those irefs together with the error code back to angler (step 3).

We use Octopus to store the state associated with a particular session. Since some of this state, namely the interface references used for the communication between the terminal client and the terminal server, should only be accessible by domains that are part of this particular session, we extend Octopus to support access control for records. Angler stores the three interface references it received from the terminal server at Octopus using the ID capability as access control (step 4).

The session is now completely initialized and as a next step angler spawns the shell (step 5). It sets the environment variable `TERM` to `xterm` and inherits the session ID capability allowing the shell to look up the state associated with the session.

After fish starts up but before the `main()` function runs, it checks whether or not it was passed a session ID capability; if this is not the case it is a daemon and steps 6 and 7 from figure 4.2 are not executed. Otherwise it uses the session ID capability to look up the state associated with the session at Octopus. In particular it retrieves the interface references it will use to communicate to the terminal server (step 6). Having received the interface references it binds to all three interfaces and is now able to do terminal I/O (step 7).

If fish starts another domain the process is analogous to angler starting fish, i.e. it spawns the new domain, sets the environment variable `TERM` and inherits the session ID capability. The new domain subsequently connects to the terminal server. Each domain which is part of a particular session has a connection to the terminal server associated with the session. So there is a many-to-one relationship between terminal clients and terminal server.

4.6 `libterm_client`

The two main responsibilities of `libterm_client` are the connection setup to the terminal server and all the character processing. It provides applications with simple functions to read and write characters. To meet our design goal of compatibility with the C standard I/O library we provide blocking read and write functions.

Native Barrelfish applications are typically written in an event-driven style and most of our native APIs are non-blocking. Apart from the blocking API, `libterm_client` also features a non-blocking API.

4.6.1 Blocking API

The function `term_client_blocking_init` (see listing 4.3) is used to initialise `libterm_client`. It takes a pointer to a `term_client` structure as well as the session ID capability as arguments. The function initializes the `term_client` structure to their default values and performs step 6 and 7 from figure 4.2, i.e. it looks up the interface references associated with the session at Octopus and connects to the terminal server. The `term_client` structure contains all the state associated with the terminal client library and is passed as first argument to all functions of the library.

```
/**
 * \brief Initialize a connection to a terminal server and block until
 *        connection is established.
 *
 * \param client     Terminal client state, initialized by function to default
 *                   values.
 * \param session_id The session the domain is part of.
 *
 * Dispatches the monitor waitset until all the bindings to the terminal server
 * are established.
 */
errval_t term_client_blocking_init(struct term_client *client,
                                   struct capref session_id);
```

Listing 4.3: Terminal client library - blocking API: initialization

If a domain is part of a session, we automatically call `term_client_blocking_init` during domain initialization. Therefore, user code does not have to initialize the terminal client library explicitly, it can use functions from the blocking API as well as the functions from the C standard I/O library right away.

To tear down the connection to the terminal server, the function `term_client_blocking_exit` is used, which is shown in listing 4.4. Like the initialization, tear down is automatically handled during domain tear down.

```
/**
 * \brief Tear down connection to terminal server.
 *
 * \param client Terminal client state.
 *
 * Dispatches the control waitset until the message is sent.
 */
void term_client_blocking_exit(struct term_client *client);
```

Listing 4.4: Terminal client library - blocking API: tear down

The blocking API features the boolean settings echo and line mode. There exist functions to turn them on and off and to query the current setting. If echo is enabled, the characters the user types at the terminal are echoed back and the user sees what he just typed. If line mode is enabled, a read from a terminal returns at most one line. Both settings are turned on by default.

In order to do an blocking read from a terminal we provide the function `term_client_blocking_read` (see listing 4.5). It takes a buffer that will hold the characters read from the terminal once the function returns as an argument. Additionally, the function takes the maximum number of characters that we want to read as an argument as well as a pointer to a variable that will hold the number of characters that were actually read at the time the function returns. The read will return if either the maximum number of characters that the user requested are read or line mode is enabled and the end of the line was reached or an error occurred.

```
/**
 * \brief Blocking read from a terminal.
 *
 * \param client  Terminal client state.
 * \param data    Buffer to hold read characters.
 * \param length  The number of characters to read.
 * \param read    Number of characters read. This might be less than length if
 *                line_mode is enabled and the end of line was reached or if an
 *                error occurred.
 *
 * \return SYS_ERR_OK if successful.
 *         TERM_ERR_IO if an I/O error occurred.
 *
 * Dispatches the read waitset if no data is available.
 */
errval_t term_client_blocking_read(struct term_client *client, char *data,
                                   size_t length, size_t *read);
```

Listing 4.5: Terminal client library - blocking API: read

Similar to the read function, we also provide a blocking write function (see listing 4.6). It takes a buffer holding the characters to be written, the number of characters in the buffer and a pointer to a variable holding the actual number of characters written once the function returns as arguments. The actual number of characters that were written by a call to `term_client_blocking_write` only differs from the number of characters the user wanted it to write if an error occurs. In this case, the former number indicates how many characters already were written before the error occurred.

Listing 4.7 contains the function prototype that is used to send a configuration

```
/**
 * \brief Blocking write to a terminal.
 *
 * \param client  Terminal client state.
 * \param data    Buffer holding characters to write.
 * \param length  The number of characters to write.
 * \param written Number of characters written. This might be less than length
 *                if an error occurred.
 *
 * \return SYS_ERR_OK if successful.
 *         TERM_ERR_IO if an I/O error occurred.
 *
 * Dispatches the write waitset until data is sent.
 */
errval_t term_client_blocking_write(struct term_client *client,
                                    const char *data, size_t length,
                                    size_t *written);
```

Listing 4.6: Terminal client library - blocking API: write

message to the terminal server. The application programmer has to specify which configuration option he wants to send and supply and optional string argument. This method of sending a configuration message can be used for example to change the baud rate of the UART driver.

```
/**
 * \brief Send a configuration command to the terminal server.
 *
 * \param client Terminal client state.
 * \param opt    Configuration option.
 * \param arg    Optional argument.
 *
 * \return SYS_ERR_OK if successful.
 *         TERM_ERR_UNKNOWN_CONFIG_OPT if option is unknown.
 *
 * Dispatches the config waitset until configuration message is sent.
 */
errval_t term_client_blocking_config(struct term_client *client,
                                     terminal_config_option_t opt, char *arg);
```

Listing 4.7: Terminal client library - blocking API: config

However, the above way of sending a configuration message is not used for cursor movements, changing the screen color or clearing the whole screen. For all of these operations there exists escape sequences that we send in-band from the terminal client to the terminal server. Sending them out-of-band could lead to unexpected behavior if the ordering is not preserved. To illustrate this let us

assume an editor wants to highlight a specific word in red. It therefore sends the command to change the color to red, sends the characters that make up the word and then sends the command to change the color back. If the first command arrives at the terminal after some characters of the word already arrived, the word would be only partially colored.

4.6.2 Non-blocking API

In a nutshell, the non-blocking API provides non-blocking sending of characters or configuration messages and callbacks for any receiving event.

Before we explain the API in more detail, let us give some insights into the internals of `libterm_client`. The terminal client library uses three waitsets, one for the incoming character stream, one for the outgoing character stream and one for the configuration interface. During initialization with the function `term_client_blocking_init`, these waitsets are set to newly allocated waitsets. The blocking API dispatches those waitsets until the required messages arrive or are sent.

One can change each of these three waitsets. Listing 4.8 shows the function that is used to change the waitset for the incoming character stream. The functions to change the waitset for the outgoing character stream and for the configuration interface are analogous.

```
/**
 * \brief Change the waitset used for incoming characters.
 *
 * \param client  Terminal client state.
 * \param read_ws New waitset to use.
 *
 * \return SYS_ERR_OK if successful.
 *         TERM_ERR_CHANGE_WAITSET on error.
 */
errval_t term_client_change_read_waitset(struct term_client *client,
                                         struct waitset *read_ws);
```

Listing 4.8: Terminal client library - non-blocking API: change read waitset

Changing the read waitset is not enough for non-blocking input. One also has to specify the callback that should be called, once characters arrive. This can be done with the function `term_client_set_chars_handler` (see listing 4.9). The character callback function that is registered is called each time characters arrive on the underlying flounder interface. It is the responsibility of the application programmer to dispatch events on the read waitset in order that the characters callback function is invoked.

```
/**
 * \brief Set handler that is called when new characters arrive.
 *
 * \param client   Terminal client state.
 * \param chars_cb Characters handler.
 */
void term_client_set_chars_handler(struct term_client *client,
                                   term_characters_handler_fn *chars_cb);
```

Listing 4.9: Terminal client library - non-blocking API: set character callback

The boolean settings echo and line mode do not apply for the non-blocking API, their current setting is simply ignored. These options are convenient for the blocking API but do not fit well into the event-driven, non-blocking style of the non-blocking API.

Non-blocking write is provided by the function shown in listing 4.10, which enqueues a buffer of characters for sending. Compared to its blocking counterpart it might return `TERM_ERR_TX_BUSY` if a previously written buffer is not yet sent via the flounder interface. One can specify a continuation that is called once the write completes.

```
/**
 * \brief Non-blocking write to a terminal.
 *
 * \param client  Terminal client state.
 * \param data    Buffer holding characters to write.
 * \param length  The number of characters to write.
 * \param cont    Continuation invoked once the write completes.
 *
 * \return SYS_ERR_OK if successful.
 *         TERM_ERR_TX_BUSY if another message is buffered but not yet sent.
 *         TERM_ERR_IO if an I/O error occurred.
 */
errval_t term_client_write(struct term_client *client, const char *data,
                           size_t length, struct event_closure cont);
```

Listing 4.10: Terminal client library - non-blocking API: write

4.6.3 Filters

One important feature of `libterm_client` is character processing. It handles the mismatch between the characters the terminal sends and the characters the user applications expect. For example, terminals send the character `\r` in response to the user hitting the return key. Most Unix-like systems, however,

use the character `\n` to indicate a line break. Moreover, to indicate a line break to a terminal one has to send the character sequence `\r\n`.

To handle all this character processing we introduce the notion of a filter. A filter is a function that takes a buffer of characters and applies some character mapping or filtering on it. The filter may add or remove characters entirely, so the number of characters in the buffer might change.

There are three groups of filters. Input filters are applied to the raw character stream received via flounder and before the characters are passed on. The above mentioned mapping of carriage return to newline is implemented as a filter and added by default to the group of input filters. Output filters are applied to the characters the user passes to `libterm_client` and before they are sent via flounder. The mapping of `\n` to `\r\n` is also implemented as a filter and by default part of the group of output filters.

There are also echo filters, which are applied to the raw characters received via flounder before they are echoed back. We will have a closer look at echo filters when we describe the implementation of `libterm_client` in section 5.4.

A sensible set of filters are added by default to the respective filter groups, but the application programmer has full control over which filters he wants to apply. Listing 4.11 shows the function that can be used to add another input filter. All the filters in one group are chained, i.e. the output of the first filter is used as the input of the second filter and so forth until all filters are processed.

```
typedef void term_filter_fn(char **data, size_t *length);

/**
 * \brief Add an input filter.
 *
 * \param client Terminal client state.
 * \param filter Input filter.
 *
 * \return Filter ID.
 */
term_filter_id_t term_client_add_input_filter(struct term_client *client,
                                              term_filter_fn *filter);
```

Listing 4.11: Terminal client library: add input filter

The filter ID that is returned by the function `term_client_add_input_filter` can be used to remove this specific input filter using the function shown in listing 4.12.

Listing 4.11 and 4.12 showed the functions used to manipulate input filters. There exists two analogous pairs of functions to add and remove output or echo

```
/**
 * \brief Remove an input filter.
 *
 * \param client Terminal client state.
 * \param id     Filter ID.
 *
 * \return SYS_ERR_OK if successful.
 *         TERM_ERR_FILTER_NOT_FOUND if there was no filter with this ID.
 */
errval_t term_client_remove_input_filter(struct term_client *client,
                                         term_filter_id_t id);
```

Listing 4.12: Terminal client library: remove input filter

filters.

4.6.4 Character triggers and domain control

In order to enable a user to control domains, we introduce the notion of character triggers. The idea of a character trigger is simple: it is a character combined with a closure. When the character appears in the input stream, the corresponding closure is called.

Listing 4.13 shows an example of a character trigger. We first define the function `term_trigger_int_handler` that prints a message and then exits. Furthermore, we specify that this function should be called, if the user types `Ctrl+C`.

```
static void term_trigger_int_handler(void *arg)
{
    fprintf(stderr, "User hit Ctrl+C.\n");
    exit(EXIT_FAILURE);
}

struct term_trigger term_trigger_int = {
    .closure = {
        .handler = term_trigger_int_handler,
        .arg = NULL,
    },
    .trigger_character = CTRL('C'),
}
```

Listing 4.13: Terminal client library: example trigger for `Ctrl+C`

As with filters, the application programmer can add and remove triggers. Listing 4.14 shows the function that can be used to add a new trigger.

The above shown character trigger for `Ctrl+C` is added by default allowing the

```
/**
 * \brief Add a character trigger.
 *
 * \param client  Terminal client state.
 * \param trigger Trigger to add.
 *
 * \return Trigger ID.
 */
term_trigger_id_t term_client_add_trigger(struct term_client *client,
                                          struct term_trigger trigger);
```

Listing 4.14: Terminal client library: add character trigger

user to terminate a running domain. If the application programmer wants to disable this behavior, he simply removes the corresponding character trigger.

Apart from the default trigger for `Ctrl+C`, there is a default trigger for `Ctrl+\`, which also terminates the running domain. This trigger, however, is special as it cannot be removed by the application programmer.

In accordance with our design goal "flexibility" the concept of filters and triggers are designed as a flexible way to manipulate a character stream and associate an action with a specific character.

4.7 libterm_server

The goal of the library `libterm_server` is to facilitate the development of device drivers that act as terminal servers. It provides device driver programmers with the ability to register callbacks when a new session is established, when characters arrive from the terminal client or when a configuration message arrives. Moreover, it features a function to send characters to a terminal client.

The library `libterm_server` maintains the notion of the current terminal client, which is the session domain that is currently running in the "foreground". Terminal input is always sent to the current terminal client. Additionally the library manages the steps 1 and 3 from figure 4.2, i.e. exports the session interface and if a new session is established, exports the remaining three interface and sends the interface references back to the domain that initiated the new session.

4.8 Alternative designs

The design presented in this chapter follows the vertically structured approach. We also considered a service oriented design with a central terminal service, but did not find a significant advantage over the vertically structured approach. Moreover, such a central service would raise further questions concerning how to deal with multiple terminals: Should there be one instance of the terminal service per terminal or should the central terminal service be able to handle multiple terminals? Also, it introduces another layer of message passing and possibly buffering. Overall, we concluded that the vertically structured approach would be easier to provide.

Chapter 5

Implementation

In this chapter we take a closure look at the implementation and describe the changes that were necessary to the rest of the system.

5.1 ID capability

The ID capability, which we introduced in section 4.3, is created in the CPU driver. It supports a single invocation called *identify* to retrieve the system-wide unique ID. This ID consists of the concatenation of the core ID representing the core on which the ID capability was created and a core-local ID. The former is a unique number that identifies the CPU driver. The latter is a strictly monotonically increasing number that we increment each time a new ID capability is created.

By using the core ID as part of the system-wide ID we enable each CPU driver to generate the ID capability without the need to run an agreement protocol while still guaranteeing system-wide uniqueness.

5.1.1 Creating capabilities at runtime

In the capability model of seL4 all capabilities are either created at startup or retyped from existing capabilities. For the set of capabilities that they support: untyped memory, virtual memory, thread control block, CNode, endpoint and asynchronous endpoint this makes sense. All of these are memory related and if one could just create a capability the memory region that the capability references could be accessed without being authorized, i.e. without holding a untyped memory capability to the corresponding region [6].

The capability model of Barrelfish being based on the one of seL4 did not allow the creation of capabilities at runtime either. The ID capability, however, is special since it does not reference a memory region, all the state associated with the capability fits into the `capability` structure, and the user should be able to create them at runtime. To this end, we extend the capability model of Barrelfish to allow the fabrication of certain types of capabilities at runtime. We add a new invocation *create* alongside copy, mint, retype, delete and revoke for CNodes that creates a new capability in a specified slot.

Currently, only the ID capability can be created at runtime. If one tries to create any other capability type, the error `SYS_ERR_TYPE_NOT_CREATABLE` is returned.

5.2 Inheritance of capabilities and passing capabilities as arguments

We learned in the chapter describing the approach that fish inherits the session ID capability from angler. Barrelfish has support for spawning a new domain and passing a capability containing the file descriptors of the current domain on to the new domain. For the session ID capability a similar inheritance mechanism is needed.

To this end, we generalized the mechanism to inherit capabilities when a new domain is spawned. We introduce the function `spawn_program_with_caps` (see listing 5.1). It takes a CNode containing all the capabilities that the newly spawned domain will inherit as an argument. The layout of this CNode is convention. Currently, the first slot is assigned to the capability used to inherit the file descriptors while the second slot is assigned to the session ID capability. The spawn daemon of Barrelfish knows about the layout of this CNode as well as the designated slot in the initial capability space that each of those capability possesses. It walks the CNode and copies any of the capabilities that are present to the corresponding slots in the capability space of the newly spawned domain.

Apart from inheriting capabilities we also need a way to pass an arbitrary list of capabilities to a domain. The recently introduced device manager *Kaluga* [30] is an example of a domain that needs to pass a list of capabilities to the device drivers it starts. The spawn daemon does not need to know about these capabilities nor do they require a designated slot in the initial capability space of every domain. Which capabilities are passed in which order is an agreement between spawner and spawnee. To this end, the function `spawn_program_with_caps` has another CNode argument. If a domain wants to pass certain capabilities to another domain it creates a CNode and places them into it and supplies the CNode as an argument to `spawn_program_with_caps`. The spawn daemon makes

```
/**
 * \param inheritcn_cap Cap to a CNode containing capabilities to be inherited.
 * \param argcn_cap     Cap to a CNode containing capabilities passed as
 *                      arguments.
 */
errval_t spawn_program_with_caps(coreid_t coreid, const char *path,
                                 char *const argv[], char *const envp[],
                                 struct capref inheritcn_cap,
                                 struct capref argcn_cap, spawn_flags_t flags,
                                 domainid_t *ret_domainid);
```

Listing 5.1: Spawn a program inheriting capabilities and passing capabilities as arguments

the whole CNode available to the spawnee in the form of a CNode if spawner and spawnee are on the same core or in form of a ForeignCNode otherwise.

Except the two arguments discussed, all arguments of `spawn_program_with_caps` are the same as when spawning a domain without inheriting or passing capabilities.

5.3 Access control for Octopus

As mentioned before, we extended Octopus with a simple access control mechanism using the ID capability as access token. One can store a set of attributes together with an ID capability. Only domains holding the capability can subsequently retrieve this attributes. The ID capability servers as the name of the record as well as the access token.

Listing 5.2 shows the function that is used to store a record at Octopus. The attribute argument could for example look as follows, which are the attributes angler uses to store the state of the session.

```
{ session_iref: 20, in_iref: 21, out_iref: 22, conf_iref: 23 }
```

The ID capability is transfered together with the attributes to the Octopus server, where the identify invocation of the ID capability is used to form the name of the record.

Listing 5.3 contains the function used to retrieve a record that was previous stored together with an ID capability.

```
/**
 * \brief Sets a record using the ID capability as the name/key of the record.
 *
 * \param idcap      ID capability used as the name/key of the record.
 * \param attributes Attributes of the record.
 * \param ...        Additional arguments to format the attributes using
 *                   vsprintf.
 */
errval_t oct_set_with_idcap(struct capref idcap, const char *attributes, ...);
```

Listing 5.2: Octopus - set a record using an ID capability as access control

```
/**
 * \brief Gets one record using the ID capability as the key/name.
 *
 * \param[out] data Record returned by the server.
 * \param[in] idcap ID capability used as the key/name of the record.
 */
errval_t oct_get_with_idcap(char **data, struct capref idcap);
```

Listing 5.3: Octopus - retrieve a record using an ID capability as access token

5.4 A peek inside `libterm_client`

We already described the API that `libterm_client` provides to an application programmer. In this section we have a closer look at the implementation of the library.

Figure 5.1 depicts the flow of characters within the library for the blocking API. The yellow box in the figure indicates the boundaries of `libterm_client`. It uses flounder messages to communicate to the terminal server below and provides the function call API described earlier above.

If a buffer of characters arrives over the underlying flounder interface we scan it for characters for which triggers are registered. For each occurrence of the trigger character, the corresponding closure is called. Subsequently, the filters from the input filter group are applied and the resulting characters are buffered in the input buffer. This buffering is necessary in the blocking API, since the application programmer might want to read less characters than arrived in a single flounder message. However, at most one flounder message is buffered. Subsequent calls to `term_client_blocking_read` are served from the input buffer and only if it is empty we dispatch the read waitset to retrieve another flounder message.

If echo is enabled, we apply the echo filters to a copy of the character buffer right after it arrives from flounder and send it as output to the terminal server. The group of echo filters contains by default a filter that displays ASCII control

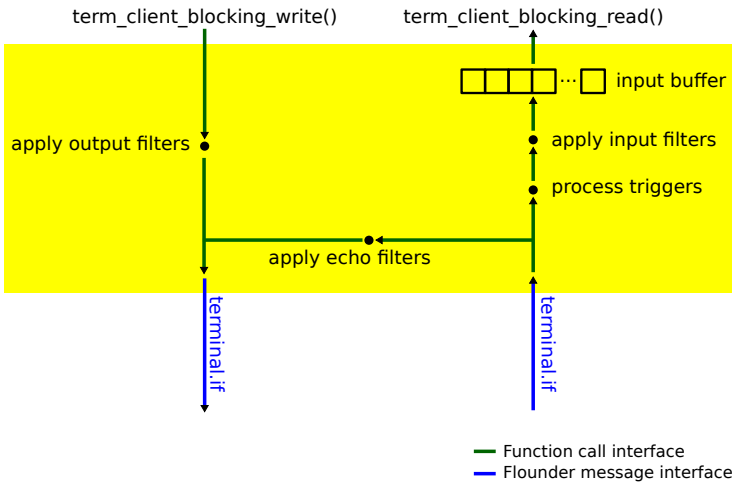

Figure 5.1: Flow of characters within `libterm_client`

characters using printable characters. For example `Ctrl+A` is displayed as `^A` and `Ctrl+C` as `^C`. This filter corresponds to the `ECHOCTL` option of a POSIX system [27].

Blocking output is straightforward: the output filters are applied, the character buffer is enqueued for transmission and we dispatch the write waitset until the send completed.

For the non-blocking API we do not dispatch any waitsets. If a buffer of characters arrives via flounder the triggers are processed and the input filters are applied as in the blocking case. However, instead of buffering the flounder message we directly call the characters callback. If the application programmer calls the blocking write, we apply the output filters and enqueue the message for transmission to the terminal server.

5.5 A peek inside `libterm_server`

As described earlier, `libterm_server` maintains the notion of the current session domain and sends terminal input to this domain. To this end, it maintains a stack of the terminal clients that are attached to this server. If a new terminal client connects, it is pushed onto the stack. When a terminal client disconnects because it terminated, the client is popped from the stack. Whenever new characters are available to the terminal server, for example because they arrived over the serial line, they are sent to the client residing at the top of the stack.

5.5.1 Communication between terminal client and server

We decided to use three flounder interfaces for the communication between terminal client and server. In particular, we have one interface for outgoing and one for incoming characters. One could also think of a single interface that contains all message types. In fact, an earlier design and implementation did exactly that. There is, however, a problem with this approach. To illustrate it, let us assume we want to send some characters and use the blocking write do to this. As outlined earlier, we dispatch events until the message is sent via flounder but we only want to dispatch events that are associated with the sending. Using a single interface we cannot guarantee that. So while we are blocking for the send to complete, there could be events for incoming character and we would handle this events as well. This poses two problems. One the one hand, there could be many incoming messages and we never get a chance to send the message resulting in a livelock. On the other hand, it would require more complicated buffering of the characters that arrive since we can not guarantee that we only receive another message once we finished delivering the data of the old one.

Using separate interfaces for incoming and outgoing characters allows to provide proper back pressure to the terminal servers and keeps the buffering much simpler.

5.6 Adapting the serial driver

Before we started this thesis, the serial driver exported a single flounder interface `serial.if` that mainly consisted of messages for input and output of buffers of characters. We extended the serial driver to additionally export the terminal interfaces so that it can act as a terminal server. Since `libterm_server` internally handles most of the terminal server specific tasks only small changes were necessary. It basically initializes the library at driver startup by providing callbacks when a new session is established or characters or a configuration message arrives. Whenever characters arrive over the serial line it submits them to `libterm_server` which in turn sends them to the correct terminal client.

The serial driver still exports the "raw interface" `serial.if` alongside the terminal interfaces since not every user domain that wants to use the serial line needs terminal services. One use case is a domain or a library that implements the Point-to-Point Protocol (PPP) [26], which can run among other alternatives over a serial line. Such a domain or library needs access to the bytes received over the serial line but does not need terminal services.

Chapter 6

Case study: OpenSSH

To demonstrate command line interaction over a network connection we ported OpenSSH [21] to Barrelfish. OpenSSH is developed as part of the OpenBSD [20] project. A seperate team takes this code and releases a so called portable version which builds and runs on a variety of POSIX systems. The OpenSSH suite consists of a ssh client, a ssh server and various other utility programs. This port only comprises the ssh server `sshd` as well as `ssh-keygen`, a program to generate and manage keys. In this chapter we describe the port in detail and show how it fits into the terminal subsystem. We start with the design goal of the port, followed with a brief overview of the functioning of the OpenSSH server. After we present the approach taken, we look at the changes we made to Barrelfish and conclude with the limitations of the port.

6.1 Design Goal

The native operating system APIs of Barrelfish do not follow the POSIX standard. Instead, POSIX compatibility is provided as a library called `libposixcompat` on top of the native Barrelfish APIs. This library allows to more easily port applications that were developed for POSIX systems. While `libposixcompat` features a respectable set of POSIX functions, it is far from complete. OpenSSH required many more functions that were not yet present. Each time we could decide to either rewrite OpenSSH to use the native Barrelfish APIs or implement the corresponding functions in `libposixcompat`. Whenever feasible within the scope of this bachelor thesis we added the functions to the POSIX compatibility library. This has the advantage that further POSIX applications can make use of this functions therefore reducing the amount of work required to port such programs. Moreover, it results in less changes to OpenSSH, making it easier to

incorporate future releases.

6.2 The OpenSSH server `sshd`

The OpenSSH server is one implementation of the Secure Shell Protocol [29]. The intention of this protocol is to provide secure remote login and data communication.

By default `sshd` listens on port 22 for new connections from a ssh client. For each accepted connection it forks itself and handles the connection in a separate process. Client and server exchange many messages including the algorithms used for securing the communication and the authentication methods accepted by the server. The client then authenticates itself at the server for example by using a password or public-key cryptography. Subsequently, the client requests a pseudo-terminal as well as a shell. The server, in response, allocates a new pseudo-terminal and starts the shell.

After that, the main responsibility of the server is to encrypt any data received from the master end of the pseudo-terminal and send it over the network to the client as well as decrypt anything received from the client and forward it to the master end of the pseudo-terminal.

6.3 Approach

The portable version of OpenSSH uses autoconf [9] to adjust the software at hand to a wide variety of UNIX-like systems. In order to be able to run the configure script and automatically detect the abundant configuration options that OpenSSH uses we set up a cross-compilation environment.

We used the cross-compilation environment until we succeeded to compile OpenSSH for Barrelfish and then switched to the native Barrelfish build system *Hake* [25], therefore providing proper integration into the Barrelfish build system.

6.4 Changes to Barrelfish

There were numerous changes necessary in order to get OpenSSH to compile and run under Barrelfish. We will only present the most important ones in this thesis.

6.4.1 POSIX headers

POSIX specifies many standard header files and defines the function prototypes and symbolic constants it should contain. Barrelfish did not provide some of these headers. For others, it did not include all the necessary definitions. We added the missing headers and definitions that where required by OpenSSH by copying them from the FreeBSD [10] project and adjusting them where necessary.

6.4.2 Pseudo-terminals

In order to implement pseudo-terminals we rely on the library `libterm_server`. In fact, one could argue that `libterm_server` already is an implementation of a pseudo-terminal. As we described earlier it provides a function to send characters as well as a callback once characters arrive, which could be seen as reading and writing from a pseudo-terminal master. While `libterm_server` provides a non-blocking API, the POSIX `read` and `write` functions typically are blocking. The implementation of the `read` and `write` functions, when called with a file descriptor representing a pseudo-terminal master, essentially just bridges the mismatch in the APIs, i.e. it provides a blocking semantic on top of the non-blocking API.

Listing 6.1 contains the sequence of functions that have to be called according to POSIX to allocate a new pseudo-terminal. We provide an implementation for each of them in `libposixcompat`, which we subsequently describe.

```
int masterfd, slavefd, ret;
char *slavepath = NULL;

masterfd = posix_openpt(O_RDWR|O_NOCTTY);
assert(masterfd != -1);

ret = grantpt(masterfd);
assert(ret != -1);

ret = unlockpt(masterfd);
assert(ret != -1);

slavepath = ptsname(masterfd);
assert(slavepath != NULL);

slavefd = open(slavepath, O_RDWR|O_NOCTTY);
assert(slavefd != -1);
```

Listing 6.1: Opening a pseudo-terminal in POSIX

The implementation of `posix_openpt` allocates a new file descriptor for the

master end of the pseudo terminal and initializes `libterm_server` therefore exporting the terminal session interface. Additionally, it creates the special file `/dev/pts/n` in the file system, which can subsequently be used to open the slave end of the pseudo-terminal. The number `n` is a system-wide unique number that is allocated using the sequential records feature of Octopus. For example, the first pseudo-terminal created in the system uses the name `/dev/pts/0`. The function returns the file descriptor of the master end of the pseudo-terminal.

The functions `grantpt` and `unlockpt` grant access to the pseudo-terminal slave end and unlock it. This is done by changing ownership and mode of `/dev/pts/n`. Since Barrelfish does not support access control at the file system level, the implementation of these two functions is empty.

The function `ptsname` simply returns the path name `/dev/pts/n` of the pseudo-terminal slave that is associated with the master. This name is then passed to `open` which allocates a file descriptor for the slave end.

6.4.3 I/O multiplexing: `select()`

After the connection setup, `sshd` uses the POSIX function `select` to do the I/O multiplexing, i.e. it blocks until any of the network sockets or the pseudo-terminal master file descriptor is ready for reading or writing.

While `libposixcompat` provided an implementation for `select` it had a severe limitation. The set of file descriptors passed to `select` could either only contain network sockets or no network sockets at all. For the ssh server, however, it was essential to be able to use both in a single call to `select`.

Before we explain why this limitation existed, let us give a brief overview of the implementation of `select`. It first checks whether or not any of the file descriptors in the read, write and exception set are ready. If this is the case, we return all the file descriptors that are ready. Otherwise, we create a new waitset and pack everything that interests us in onto that waitset and dispatch the waitset until at least one file descriptor is ready. The problem with mixing network sockets with other kinds of file descriptors was that LWIP [17], the TCP/IP stack that we use for Barrelfish, was not yet adapted to support waitset.

Therefore, we first extended LWIP to support waitsets. In particular we provide a function that allows the programmer to register a waitset for a specific socket. Once this socket has data available for reading, an event is delivered on the waitset. There exists an analogous function for the write case. This functions provided the basis for integrating the LWIP networks sockets into our implementation of `select`.

6.5 Limitations of the port

6.5.1 Only a single connection

The ssh server forks itself to provide support for multiple connections. Barrelfish, however, does not support `fork`. One could either provide an implementation for `fork` or rewrite `sshd` to use multiple threads, one for each connection. Both approaches, however, were out of the scope of this bachelor thesis and we currently only support a single connection.

6.5.2 Static seed

OpenSSH uses the OpenSSL [22] library for all cryptographic operations. During initialization OpenSSL is fed with seed typically from `/dev/random`. Barrelfish does not yet feature a random number generator and proving it was deemed out of scope. Especially, since recent research has shown that such an implementation is far from trivial [12]. As a kludge, we currently use a static seed.

Chapter 7

Qualitative Evaluation

This chapter performs a qualitative evaluation of our design and implementation by comparing them to UNIX-like systems. We identify the key differences and the motivations for them.

7.1 Monolithic kernel versus microkernel

Most UNIX-like operating systems implement the terminal subsystem in the kernel. Since Barrelfish follows the microkernel approach, the terminal subsystem is implemented in user space in user domains and libraries. We did not encounter any significant difficulties with the approach taken. The only occasion we require support from the CPU driver is to enforce protection between multiple sessions. The capability system of Barrelfish and the ID capability we introduced allowed us to implement the terminal subsystem in user space while still providing protection between sessions.

7.2 Line disciplines

As mentioned in the chapter surveying the related work, UNIX-like systems implement all the terminal related character processing in a module called terminal line disipline. They provide additional line disciplines for the Point-to-Point Protocol (PPP) or for using IP over the serial line (SLIP). We do not support such an overloading of the terminal subsystem and in our view such functionality is better provided by seperate modules. The terminal subsystem that we designed is intended for the communication between a terminal or terminal emulator and a user domain. The Point-to-Point Protocol or serial

IP do not fall into that category. As outlined in section 5.6, the serial driver additionally exports the raw interface. This provides the means to implement protocols like PPP or SLIP in separate modules.

7.3 Filters and character triggers

The concepts of filters and character triggers that we introduce in this thesis and use to implement character processing and domain control is different from UNIX-like systems. In UNIX-like systems there is a set of predefined options that one can enable or disable. For example, the mapping of `\n` to `\r\n` on output is enabled by setting the option `ONLCR`. Our design is more flexible in that the application programmer can add its own filters and triggers without to need to change the terminal subsystem. On the other hand, not every option that UNIX-like systems support can be mapped to the concept of filters and triggers. For example, erasing the last word with `Ctrl+W`. To illustrate why this is not possible let us look at the non-blocking API. Whenever characters arrive over the underlying flounder interface the character callback is called and the characters are passed on. If later the character representing `Ctrl+W` arrives, the previous characters are already passed on, so we cannot erase them.

While it would have been possible to add the feature to erase the last word and in general command line editing similar to the one provided by the canonical mode of UNIX-like systems for the blocking API, we decided not to. On the one hand, we wanted to keep the blocking and the non-blocking API of the terminal client library as consistent as possible. On the other hand, we think that such functionality is better implemented in a separate library. In fact, although UNIX-like systems provide command-line editing via the canonical mode, the importance of this feature declined over time and dedicated libraries like the GNU Readline Library [23] are used instead. The reason for this is that canonical-mode command line editing provides only basic support. For example it is possible to erase the last character or the last word or the entire line but it is not possible to edit a line in the middle. The Readline Library, on the other hand, supports this as well as many more useful features and is therefore preferred by application programmers.

Chapter 8

Conclusion

In this thesis we introduced a framework for handling character-based I/O streams. The framework consists of two flounder interfaces, each handling a unidirectional character stream, as well as one that handles configuration messages. Moreover, the framework features a library that is responsible for character processing and handling of special characters such as `Ctrl+C`. We adapt the serial driver to work with this framework.

Additionally, we define the concept of a session and provide protection between them by using a newly introduced ID capabilty. We provide basic domain control: starting a domain as part of a session, detaching from the session and termination a domain.

As a case study, we ported OpenSSH to Barrelfish and provide a POSIX compatability layer for terminal I/O so that OpenSSH only has to be minimally modified.

8.1 Directions for future work

While the work done as part of this thesis extends and improves Barrelfish in many ways, there are still areas for improvements. The interfaces used to talk to a terminal that we designed as part of this thesis are independent of the terminal. However, the characters, in particular the escape sequences, depend on the terminal. There exist databases like terminfo or the older termcap [24] that specify for every terminal which escape sequences to use for a specific operation, e.g. moving the cursor. Such databases or libraries like ncurses [15] that build on the former databases would provide the means to write applications in a terminal independent way. However, none of them have been ported to Barrelfish yet.

The ID capability with its properties of being system-wide unique and unforgeable is useful for many other IDs apart from the Session ID, for example interface references. While the current implementation guarantees system-wide uniqueness for interface references, they are easy to forge. As a result it is easily possible to connect to any interface exported without being explicitly authorized, e.g. without receiving the interface reference directly or indirectly from the domain that exported the interface.

While we presented basic domain control, there is room for improvement. The current design does for example not support fully-fledged job control with fore- and background jobs as known from UNIX-like operating systems.

Apart from the initialization, an ssh server is essentially an event-driven program. It acts upon data received from the ssh client, from the pseudo terminal master or upon new connection requests. As such it would nicely fit into the native Barrelfish programming environment arguably much better than on top of the POSIX API. Having a native implementation of the secure shell protocol would further allow to integrate capabilities in order to improve security. One could for example use a capability for the host or session keys. Such a capability would only provide invocations for encrypting or decrypting data with the key but no way to access the key itself. So even if the ssh server would be compromised and as long as the kernel is not compromised, the attacker can not get hold of the key.

Bibliography

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991.

[2] Andrew Baumann. Inter-dispatcher communicaion in Barrelfish. Barrelfish Technical Note 011, Systems Group, ETH Zurich, December 2011.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, October 2009.

[4] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[5] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[6] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. *seL4 Reference Manual*. NICTA, 2006. http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf.

[7] Thomas E. Dickey. Xterm - terminal emulator for the x window system. http://invisible-island.net/xterm/.

[8] Digital Equipment Corporation. *VT100 Series Technical Manual*, 2nd edition, 1980. http://vt100.net/docs/vt100-ug/.

[9] Free Software Foundation. *Autoconf*. http://www.gnu.org/software/autoconf/.

[10] FreeBSD. http://www.freebsd.org.

[11] FreeBSD Man Pages - getty(8). http://www.freebsd.org/cgi/man.cgi?getty.

[12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[13] IEEE Computer Society. *Standard for Information Technology – Portable Operating System Interface (POSIX) – Base Specification, Issue 7*. IEEE Std 1003.1-2008.

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, October 2009.

[15] Ncurses library. http://www.gnu.org/software/ncurses/ncurses.html.

[16] Jochen Liedtke. *Page Table Structure For Fine-Grain Virtual Memory*. German National Research Center for Computer Science (GMD), October 1994.

[17] LWIP. A lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/.

[18] Gnome Terminal Manual. http://library.gnome.org/users/gnome-terminal/stable/.

[19] Marshall Kirk McKusick and George V. Neville-Neil. *The Desgin and Implementation of the FreeBSD Operating System*. Addison Wesley, 2005.

[20] OpenBSD. http://www.openbsd.org.

[21] OpenSSH. http://www.openssh.com.

[22] OpenSSL. http://www.openssl.org.

[23] Chet Ramey. *The GNU Readline Library*. http://www.gnu.org/software/autoconf/.

[24] Eric S. Raymond. Termcap/terminfo resources page. http://www.catb.org/terminfo/.

[25] Timothy Roscoe. Hake. Barrelfish Technical Note 003, Systems Group, ETH Zurich, April 2010.

[26] W. Simpson. *The Point-to-Point Protocol (PPP)*, July 1994. RFC 1661.

[27] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison Wesley, second edition, 2005.

[28] Barrelfish Team. Barrelfish architecture overview. Barrelfish Technical Note 000, Systems Group, ETH Zurich, June 2010.

[29] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*, January 2006. RFC 4251.

[30] Gerd Zellweger. Unifying synchronization and events in a multicore operating system. Master's thesis, Systems Group, Department of Computer Science, ETH Zurich, March 2012.