Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 258

Systems Group, Department of Computer Science, ETH Zurich

Running Linux binaries over Barrelfish using a Library OS

by

Marc Tanner

Supervised by

Roni Häcki
Lukas Humbel
Dr. David Cock
Prof. Dr. Timothy Roscoe

March 2019–August 2019

Informatik
Computer Science

Abstract

The Graphene library operating system (OS) moves much of the functionality provided by the Linux kernel into a library linked into the application's own address space. Instead of depending on a large array of system calls, the library only relies on a narrow Platform Adaption Layer (PAL) providing high level abstractions like virtual memory, threads and I/O streams. Advantages include host OS independence and application mobility.

This work describes the porting of the Graphene PAL to the Barrelfish research OS, enabling it to run unmodified, dynamically linked, Linux binaries making use of typical Unix features such as pipes, signals and sockets. Challenges encountered while implementing the PAL interfaces, traditionally backed by a monolithic kernel, on top of Barrelfish's multikernel architecture are investigated.

Acknowledgments

I would like to express my gratitude towards professor Roscoe for giving me the opportunity to pursue my interests in operating systems. I am especially thankful to my supervisors Ronni and Lukas as well as the rest of the Barrelfish team for their feedback, guidance and support during our weekly meetings. Thanks to all members of ETH Zurich's Systems Group and fellow students for the intriguing discussions and welcoming work environment.

Finally, I would like to thank my family and friends for their understanding, patience and continued support during my studies.

Contents

Chapter 1

Introduction

So-called library operating systems (OS) approaches to running applications have seen much recent interest in the research community. In these systems, exemplified by Drawbridge[43] and Bascule[9] at Microsoft, BSD "Rump Kernels"[29], and Graphene at Stony Brook University, much of an existing OS is compiled into a library which is then linked against an application written for that OS. The library itself requires not the usual array of system calls but a small number of API calls, which are implemented by a small portability layer for a given host OS. The advantages are that programs written for one OS can be easily executed on another, migrated or persisted on the same OS, or be subject to interposition techniques for replay and debugging. The decoupling of the library OS from its host interface enables independent evolution and provides security isolation with lower overheads than traditional virtual machines.

Graphene[52] is a library OS implementing a Linux personality capable of running multi-process applications. This work describes the porting of the Graphene Platform Adaption Layer (PAL), providing high-level abstractions like virtual memory, threads and I/O streams, to the Barrelfish[8] research OS. As a result, dynamically linked Linux binaries can be executed unmodified on Barrelfish. Supported features include POSIX threads, pipes, signals and sockets. Unlike existing PAL hosts such as Linux and FreeBSD, Barrelfish is not architectured like a typical Unix system based on a monolithic kernel. Instead, it minimizes shared state using the multikernel design. Discussed challenges include bootstrapping of the library OS, the construction of a unified virtual address space comprised of both file backed and anonymous pages, virtualization of resources required for thread local storage and support for multi-process abstractions.

The work is structured as follows: chapter 2 provides the necessary background information about Barrelfish. The Graphene library OS is introduced in chapter 3 where host OS independent functionality is discussed. Chapter 4 describes the PAL used by Graphene to interface with the host OS. Chapter 5 investigates how the PAL is implemented on Linux. The porting of the PAL to Barrelfish is detailed in chapter 6, which is subsequently evaluated in chapter 7. I contextualize my and survey related work in chapter 8. Finally, I summarize my contributions and conclude with ideas for future work in chapter 9.

Chapter 2

Barrelfish

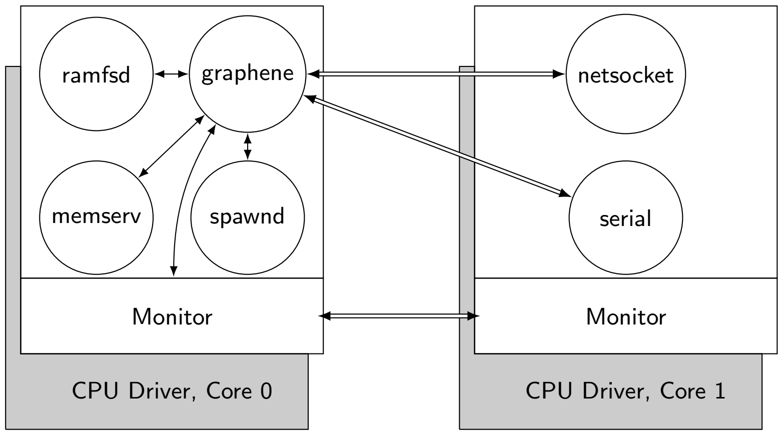

Figure 2.1: Barrelfish architecture overview

Barrelfish is a scalable research operating system based on the multikernel[8] architecture. It treats the local machine as a distributed system[11] of heterogeneous components, linked by a complex interconnect network. Instead of relying on cache-coherent shared memory, it distributes or replicates OS state using explicit message passing.

Similar to an exokernel[21], physical resource management is delegated to system run-time library in user space, whereas the the kernel only enforces protection and authorization. As in a microkernel[35] unprivileged user space processes provide core system services (e.g. file system and network access) and device drivers.

Figure 2.1 depicts the overall system architecture, circles represent independent user space processes, arrows denote communication channels used by the Graphene loader. Individual components are described in subsequent sections.

2.1 CPU Driver and Monitor

To emphasis its simplicity the privileged kernel is referred to as *CPU driver*. Each core runs an independent instance, thereby allowing specialization and seamless support of heterogeneous systems. The CPU driver follows the stateless[22] kernel approach, it is single threaded, non-preemptive, non-reentrant, has no blocking system calls and performs no dynamic memory allocation[44]. It provides minimal hardware abstraction, enforces protection and performs authorization of physical resources using capabilities[5]. Page faults, traps, exceptions and device interrupts are delivered to user space using upcalls[14]. Dispatchers are scheduled and fast local message passing among them is performed. Kernel services are requested through a system call interface.

The *monitor* is a specially trusted user space process, complementing the CPU driver on each core to perform potentially long running operations. The monitors form an inter-core network used to provide low-level OS functionality typically found in a monolithic kernel. OS state is replicated using distributed algorithms to maintain consistency across core boundaries[5]. Each user level process is provided with a connection to its local monitor which is subsequently used to bootstrap inter-core communication.

2.2 Capabilities and Virtual Memory Management

Access control to kernel objects and physical resources is enforced using capabilities. Like seL4[33], Barrelfish employs a partitioned capability system in which the typed memory of a capability is only accessible to the local CPU driver. User space uses capability references, rooted in a dispatcher specific capability space, to perform capability invocations through a system call interface.

Barrelfish applications construct their own virtual address space through secure page table manipulations by means of capability invocations. A variant of self-paging[26] is used: the CPU driver reflects page faults back to the affected dispatcher via upcalls. Above this kernel interface the system library `libbarrelfish` provides convenient high-level abstractions similar to those found in the Mach memory system[45]. The main components are[24]:

- *Shadow page tables* keep track of the virtual address space: all mappings, including meta data and the used capabilities.
- *Virtual regions* represent a contiguous region of virtual memory.
- *Memory objects* back virtual regions by handling all their page faults.
- *Virtual space* structure, keeping track of a sorted list of virtual regions.

This separation allows a great deal of flexibility, different types of memory objects can employ different physical memory allocation strategies e.g. to exploit NUMA effects or use huge pages.

2.3 Domains and Dispatchers

A process context in Barrelfish is represented by a *domain*, which bundles a number of core bound *dispatchers*. The latter of which are scheduled by the CPU driver in the form of a dispatcher control block (DCB). The first part of the DCB is private to the CPU driver and contains references to the capability space, root page table and scheduler information[5]. The second part of the DCB is also accessible from user space and contains upcall entry points to deliver page faults, traps, inter-domain messages and scheduler activations. A domain's address space spans core boundaries, but the capability space is local to its dispatcher.

2.4 User Level Threading

Based on the dispatcher abstraction of the CPU driver a form of scheduler activation[2] is used to implement user level threading. A dispatcher can either be *enabled*, meaning it executes application code, or *disabled* when executing code within the threading library itself. The user level scheduler is non-reentrant: when disabled upcalls are blocked. Both states have an associated save area in the shared part of the DCB where upon preemption of the dispatcher the register state can be saved. Resuming a disabled dispatcher, restores the previous execution state and code resumes within the threading library as if no preemption happened. When resuming an enabled dispatcher, the CPU driver changes the dispatcher state to disabled and then performs an upcall into the user level scheduler. It can now decide to resume the same execution state or save it to a thread control block (TCB) and restore a different thread. The TCB is a purely user space structure of which the CPU driver is not aware. Besides register state it also contains references to the associated dispatcher, regular thread stack, exception stack, exception handler and thread local storage (TLS) data.

For later sections it is crucial to understand the different stacks and register states involved in handling a page fault:

1. The CPU driver catches a page fault and upcalls into the entry point registered in the DCB.

2. The system runtime library executes on a dedicated dispatcher stack.

3. The thread exception handler is called on the thread exception stack and given access to the register state at the time of the page fault.

4. The exception handler returns and the possibly modified register state is restored. Typically execution resumes on the regular thread stack.

Notice that the exception handling mechanism is non-reentrant, meaning the exception handler must not generated further faults.

The threading library also provides low-level synchronization primitives like semaphores, mutexes and conditional variables as well as higher-level abstraction such as waitsets and deferred events.

2.5 Thread Local Storage

Thread Local Storage (TLS) provides fast access to thread specific data. C applications can declare a variable to be thread local by specifying the storage class `__thread` (a language extension supported by e.g. GCC[1]) which was then standardized in C11 as `_Thread_local`. A common POSIX example of a variable which could be declared that way is `errno`.

The x86-64 ELF TLS ABI[19] specifies that such variables are located relative to the thread pointer denoted by the `%fs` segment register. Barrelfish - like Linux - uses variant II of the TLS specification, which mandates that the thread pointer points to the thread control block and TLS variables are accessed using a negative offset.

One peculiarity of the segment registers is that until the Intel Ivy Bridge CPU models they could only be set in privileged mode. The full 64-bit virtual address can be set using a Model Specific Register (MSR). Alternatively, the 32-bit segment base address can be set by manipulating entries of the Local Descriptor Table (LDT). The table can subsequently be indexed using a 16-bit segment selector which can be set in unprivileged mode.

The latter method is currently used by Barrelfish. Upon thread creation a system call is performed to allocate a LDT entry pointing to the thread control block. When performing a thread switch in user space, the segment selector is adjusted accordingly. This avoids a system call which would somewhat defeat the purpose of user level threading. During a context switch to another dispatcher the complete LDT base address is swapped out.

Notice that this setup dictates the location of the TCB which must necessarily be allocated in the lower 32-bit of the virtual address space.

2.6 Inter-dispatcher Communication

Barrelfish implements a number of core system services as independent user space servers exposing a remote procedure call (RPC) interface. Conceptually services are published to a name server where clients can query for and subsequently bind to them. Flounder[6], an interface definition language, is used to specify the API in terms of typed messages. Based on the definition a stub compiler generates plumbing code to marshal, dispatch, fragment and reassemble messages.

The stubs provide a common interface independent of the underlying inter connect driver (ICD) backend. On the same core Local Message Passing (LMP) is employed to transfer small messages in hardware registers using a fast path in the CPU driver akin to concepts found in L4 IPC[36]. Cross-core communication uses a cache-coherent shared memory region to transfer cache-line sized frames without kernel involvement. The technique is similar to user level RPC[12] and in Barrelfish referred to as Userlevel Message Passing (UMP).

Figure 2.1 distinguishes them as follows: LMP channels on the same core are denoted by normal arrows, whereas the core spanning UMP channels are depicted using double arrows.

[1] `http://gcc.gnu.org/onlinedocs/gcc/Thread-Local.html`

2.7 File System

Barrelfish applications can access the hierarchical file system by means of a generic Virtual File System (VFS) API which is implemented using RPC calls to file system servers. Available backends include support for NFS, FAT32 and ramdisks. The API exposes only very limited meta data: directories and files are distinguished, the latter has an associated size property. A permission model is currently not implemented. In contrast to a typical Unix environment neither hard nor symbolic links are supported and a rename primitive is missing.

2.8 Networking

Barrelfish uses user space network drivers to set up the queuing interface of modern network cards. Applications either manipulate these hardware queues directly, by running their own network stack on a per flow-basis, or talk to an additional user space service which performs the necessary multiplexing on their behalf[5].

The net socket server performs this role by exposing a callback based RPC interface. Clients can create either UDP or TCP sockets and will be notified whenever data has been received or has successfully been transmitted. This callback based interface fits the general event-based programming model favored by Barrelfish, but proves to be challenging to fit into a more traditional BSD socket API.

The actual network protocol implementation are provided by the lwIP TCP/IP stack[20] which depending on the configuration provides 3 different programming interfaces[2]:

- The *raw* API is event based, registered callback functions are directly invoked from within the core IP stack. The application code and the IP stack run in the same thread. This supports zero-copy operations where the caller has to ensure that the data remains available until indicated by a completion event. It completely bypasses the OS support layer, but only one thread can run the packet processing loop and perform network operations.

- The *netconn* API, also referred to as sequential-style. It provides blocking calls, facilitating a more traditional socket like usage. It introduces a dedicated TCP/IP thread which runs the event-based protocol stack, while application code, issuing blocking calls, must run in different execution contexts. It preserves zero-copy operations, but introduces more abstractions and thread synchronisation primitives relying on an OS support layer.

- The *sockets* API provides a compatibility layer for BSD sockets as e.g. specified in POSIX. It is implemented on top of the 'netconn' layer. As a result multiple threads can issue socket calls, albeit with higher overhead due to the required synchronization.

[2] `http://www.nongnu.org/lwip/2_1_x/group__api.html`

To further complicate matters Barrelfish source contains multiple versions of the lwIP stack with different supported modes of operation.

The older and deprecated Barrelfish network architecture is based on lwIP version 1.1.3 with modifications to talk to a common network service (netd) for port allocation and ARP request lookup. These changes allow multiple network aware applications to work concurrently. The original design is described in a distributed systems lab project[47]. The lwIP stack is operated in OS mode i.e. using a Barrelfish specific adaption layer[3] providing: semaphores, mutexes, mailboxes and a thread abstraction. Although it has some limitations (non-existing timeout handling for semaphores, mailboxes always store only a single message) it empowers all 3 lwIP API variants.

The newer Barrelfish network architecture uses lwIP version 2.0.2 without the OS adaption layer, meaning the lwIP stack operates in mainloop mode and only supports the raw, event-based API. This fits the general Barrelfish programming model, but precludes multi-threaded and blocking access to the same network stream. Instead of performing inter domain communication within lwIP, a new library (libnet) was introduced to handle network initialization and polling of lwIP events.

2.9 Existing POSIX compatibility

The C runtime library used in Barrelfish, derived from FreeBSD libc, in combination with `libposixcompat` provides basic POSIX functionality. File operations are mapped to the VFS interface, sockets are implemented using the older lwIP socket API. `mmap()` only handles the most basic cases, major limitations include: only anonymous i.e. no file backed memory mappings, no overlapping mappings, no partial unmappings. Dynamic loading of shared objects, signals, fork and the exec families of functions are not supported to name just a few omissions.

In contrast to Graphene's library OS approach, applications need to be linked against these compatibility libraries and can not be used unmodified.

[3]`http://www.nongnu.org/lwip/2_1_x/group__sys__os.html`

Chapter 3

Graphene Library OS

Graphene[52, 51] provides a Linux kernel personality in form of a shared library which can be pre-loaded into an application address space. It implements the Linux syscall interface, special devices and pseudo file systems using a narrow Platform Adaption Layer (PAL). Moving the state from the kernel into the application's address space facilitates migration techniques and provides host OS independence. Such a library OS process, with only a limited host OS interface, is referred to as *picoprocess*.

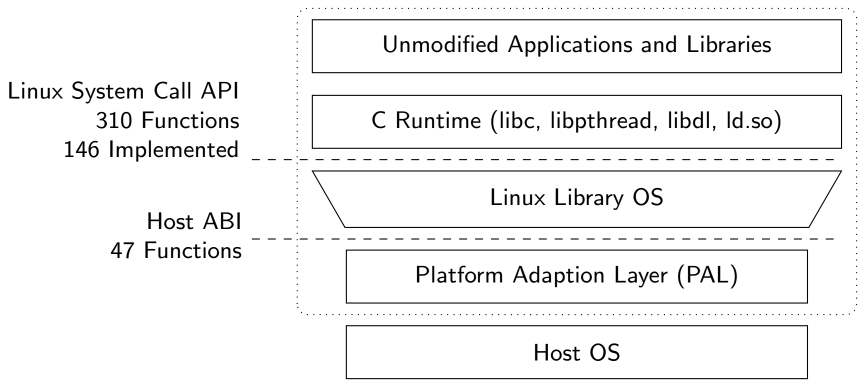
Linux System Call API
310 Functions
146 Implemented
Host ABI
47 Functions
Unmodified Applications and Libraries
C Runtime (libc, libpthread, libdl, ld.so)
Linux Library OS
Platform Adaption Layer (PAL)
Host OS

Figure 3.1: Graphene architecture overview.

Figure 3.1 illustrates the basic Graphene architecture. Core system libraries like the C runtime (libc), threading library (libpthread) ELF linker and loader (ld.so), dynamic library linker (libdl) have been modified to perform function calls into the library OS instead of system calls to the Linux kernel. Let us exemplify the whole sequence:

1. An unmodified dynamically linked application calls `malloc()`.
2. The call is resolved to the modified C library whose allocator requests memory through the `brk()` or `mmap()` syscalls.
3. The library OS implements the syscalls using the `DkVirtualMemoryAlloc()` PAL call.
4. The PAL call is satisfied by the native memory system of the host OS.

3.1 Resource Handles

In Unix systems file descriptors are employed by user space to reference kernel objects. Within Graphene these objects and their respective state resides within the library OS in the same address space. File descriptors are used to index a lookup table to access generic handles, implemented as tagged unions.

The shared part contains a reference to the underlying PAL resource as well as type, locking and reference counting information. The type specific section contains the necessary bookkeeping data needed to provide the corresponding POSIX interface. As an example: an opened file handle keeps track of its original host path, file type, file size, the current file offset and a small data cache around it.

3.2 File System

Much like the Linux kernel itself, Graphene also features a common Virtual File System (VFS) interface abstracting different backends behind a common API. The VFS contains a directory cache for metadata using positive and negative entries to minimize PAL calls. Batched I/O operations of the `readv()` and `writev()` syscalls are implemented at this level.

For actual data storage a ”chroot” style file system is used to expose a host directory within the file system accessible to the picoprocess. The typical I/O operations are mapped to their PAL stream counterparts. In order to reduce the latency of repeated `read()` and `write()` syscalls with small buffers, a memory mapped based caching mechanism is used around the current file position.

Applications also expect the presence of a number of character devices and pseudo file systems emulated by the library OS:

- `/proc` with process information. The per-process directories contain symlinks pointing to the root directory (`root`), the current working directory (`cwd`) and the binary being executed (`exe`). The `maps` file exposes the virtual address space layout. Finally, the `fd` subdirectory maps file descriptors to their referenced objects.
- `/proc/cpuinfo` to advertise CPU capabilities and `/proc/meminfo` to expose the configured memory quota.
- `/dev/tty` representing the interactive console.
- `/dev/random` and `/dev/urandom` as random sources. The former requests randomness from the host, while the latter is only seeded through the PAL and then uses repeated hashing to produce output without ever blocking its caller.
- `/dev/zero` as an infinite stream of zero bytes.
- `/dev/null` as a data sink, where read attempts always report end of file.

3.3 Virtual Memory

The Linux kernel provides a virtual address space abstraction consisting of page aligned virtual memory areas (VMA) which can be individually allocated, disposed or replaced. The `mmap()`, `mprotect()`, and `munmap()` system calls are implemented by calling their counterparts within the PAL.

The `brk()` and `sbrk()` system calls, manipulating the "program break" at the end of the data segment of the loaded executable, are served from a pre-allocated memory region. The standard C library allocator, which makes use of these lower level interfaces to provide `malloc()` and `free()`, is used unmodified.

As an optimization the Linux kernel loads the vDSO, a virtual dynamic shared object, into each process address space to provide syscalls (e.g. `gettimeofday()`) which do not need elevated privileges. This mechanism is currently not supported within Graphene. Instead, the affected system calls are handled like all others: implemented in the library OS based on PAL functionality.

The library OS itself uses slab allocators for various types of internal objects. Address space layout randomization can optionally be enabled, to randomize base addresses of returned allocations.

The current virtual address space layout is tracked using a sorted double linked list of VMAs recording the base address, size, protection flags and backing file handle (if any) of each contiguous memory area. Section 3.5 will describe how this data is used to implement checkpointing and migration.

3.4 Threads

Applications typically use the POSIX threading (pthread) library to create threads and related synchronization primitives like mutexes, conditional variables, reader/writer locks and barriers. Linux uses a 1:1 scheduling scheme where the user space `pthread_t` control block refers to a schedulable kernel object using a thread ID. This thread control block (TCB) structure also contains the necessary data for thread local storage (TLS) as specified in the x86-64 ELF TLS ABI[19]. The TCB was extended with additional references to cover the bookkeeping needs of the library OS.

As was explained in section 2.5, setting the `%fs` segment register to an arbitrary virtual address holding the TCB requires a privileged instruction. In Linux the TCB can either be specified as an argument to `clone()` during thread creation, or later by invoking the `arch_prctl()` syscall. Both the segment register manipulation and thread creation are performed using appropriate PAL calls.

The pthread synchronization primitives are all implemented on top of the `futex()` system call. It allows to block the calling thread until a supplied memory location has the desired value. A corresponding wake up mechanism can be used to notify blocking threads. Within the library OS this functionality is implemented using PAL synchronization events.

3.5 Processes

Unix systems create new processes using `fork()` by duplicating the calling process. For the virtual address space this is typically implemented using copy-on-write of the underlying physical pages. However, Graphene does not require a shared memory environment. Instead, it creates a new pristine process, checkpoints the parent and migrates all process state to the child via a PAL RPC stream. This snapshot includes all virtual memory areas, process handles i.e. file descriptors and signal handlers.

`execve()` is used to replace the currently running binary image with a new program in a fresh virtual address space. Unless file descriptors are marked as close-on-exec, they remain open and accessible to the new application. This is implemented in the library OS by creating a new process and selectively migrating the desired file descriptors. Other transferred state includes the environment variables and program arguments.

To implement multi process functionality like signal delivery and exit notification, each library OS uses a dedicated inter process helper thread which coordinates OS state using a mixture of point-to-point and broadcast RPC streams. Global resources such as the process identifier namespace and interfaces like `kill()` and `waitpid()` are provided by exchanging messages over these channels.

3.6 Inter Process and Network Communication

The library OS supports Unix domain, TCP stream and UDP datagram sockets as well as pipes by relying on byte stream abstractions provided by the PAL. This emphasises the use of high-level abstractions: the network stack is part of the host OS and the library OS has no access to its internal state.

Chapter 4

PAL Host ABI

The Platform Adaption Layer (PAL) defines a host neutral application binary interface (ABI) providing high-level abstraction like processes, threads, virtual memory and I/O streams suitable to serve as a base for the library OS.

It is an extension of the Drawbridge ABI[43] used to host a library OS version of Windows 7. The most prominent new interfaces concern the support of multi-process applications. Extensions related to exception handling and thread local storage were already present, although in slightly different form, in Bascule[9].

As guiding principles the PAL ABI should provide generic, host independent, stateless, narrow interfaces to aid portability, reduce attack surface and facilitate migration[51].

The PAL is responsible to translate from the x86-64 System V ABI used by Linux to the host calling convention. It contains an executable and linkable format (ELF) loader which loads application, library OS and PAL into a common address space, performs symbol resolution and dynamic relocation before jumping to the application entry point.

4.1 Resource Handle

Analogous to the library OS (see section 3.1) the PAL also uses a tagged union as a common handle referencing host OS resources. Apart from the type information the definition of this `PAL_HANDLE` is host dependent and provided by concrete PAL implementations.

The common PAL code defines a global process control block structure (`PAL_CONTROL`) capturing global state such as: the host type, process id, CPU capabilities, memory quotas, profiling information and handles to the parent process, main thread and broadcast stream. These are all initialized before starting execution of the library OS.

4.2 I/O Streams

I/O streams provide a common byte stream abstraction for files, RPC communication and TCP as well as UDP sockets. The `DkStreamOpen()` PAL call defines an unified resource identifier (URI) scheme to refer to different kinds of host OS objects:

- `file:path` where path is either absolute or relative to the current working directory. As a convention, paths ending in a trailing slash refer to directory streams.
- `pipe:name` is used to initiate a connection to a previously published RPC server `pipe.srv:name` with matching name.
- The same principle is also applied to TCP (`tcp:address:port`) and UDP (`udp:address:port`) clients. The corresponding servers use the same addressing format, but with the `tcp.srv` and `udp.srv` URI scheme, respectively.
- `dev:tty` referring to a character device representing the active console.
- The broadcast stream, connecting all picoprocess running in the same sandbox, has no designated URI but is initialized automatically in the PAL startup code.

All streams support I/O operations using `DkStreamRead()` and `DkStreamWrite()`. Both calls take position arguments denoting absolute file offsets. This is an example where the PAL ABI minimizes internal state and instead externalizes it to the library OS. Non-file streams simply ignore the position argument. The return value indicates the number of processed bytes.

Streams are accompanied with meta data like: the type, name, blocking behavior, readable or writable state and access modes. Sockets expose certain TCP options (keepalive, cork, nodelay) and expose the send/receive buffer sizes and timeouts.

Server streams support a blocking `DkStreamWaitForClient()` call which suspends execution until a connecting client appears. It returns a handle representing the newly established connection.

File related streams support operations to truncate or extend (`DkStreamSetLength()`) and persist (`DkStreamFlush()`) data as well as manipulate their naming within the file system (`DkStreamDelete()` and `DkStreamChangeName()`). The Unix permission model based on owner, group and others with read, write and execute permissions is exposed through calls for metadata queries.

An array of handles can be polled for activity using the `DkObjectsWaitAny()` call which blocks until one of the observed handles indicates a state change, in which case it is returned, or the given timeout expired.

4.3 Virtual Memory

The PAL provides a virtual address space abstraction consisting of page sized regions which are either backed by file content or anonymous memory. Individual pages can either be inaccessible or a combination of readable, writable and executable. Existing memory areas can be superseded by new overlapping mappings.

`DkVirtualMemoryAlloc()` allocates a page aligned, contiguous, anonymous memory region either at a provided fixed address or lets the host OS decide its location. Protection flags specify the aforementioned access modes, either when creating the mapping or afterwards using `DkVirtualMemoryProtect()`. A memory range can be returned to the host OS with `DkVirtualMemoryFree()`.

Analogous to the anonymous mappings, `DkStreamMap()` can be used to establish file backed memory regions. The interface takes an additional opened stream handle, file offset and optional copy-on-write flag. The latter specifies that any changes are kept private to the current picoprocess and not shared nor represented in the underlying file. `DkStreamUnmap()` invalidates an existing file based memory area.

4.4 Threads

The PAL defines a thread abstraction with dedicated stack and register state, but shared address space. `DkThreadCreate()` initiates a new thread, starts execution of the given entry point and returns a thread handle. A thread can voluntarily give up its associated scheduler time slice by calling `DkThreadYieldExecution()`. Similarly, `DkThreadDelayExecution()` suspends execution of the calling thread for a specified time frame. `DkThreadResume()` restores the execution state of a given thread. The lifetime of the calling thread can be terminated using `DkThreadExit()`.

A number of thread synchronization primitives are also specified by the PAL. Critical sections can be protected using mutually exclusive locking. `DkMutexCreate()` returns a new mutex handle, `DkObjectsWaitAny()` acquires the lock, `DkMutexRelease()` releases it. To enforce dependency relationships among threads two different event types are defined:

- *Synchronization* events are used to model producer/consumer relationships. When signaled, exactly one waiting thread is released and the event is automatically cleared.
- *Notification* events are used to represent one-off occurrences, once set they remain signaled until explicitly cleared.

Both event types can be signaled using `DkEventSet()` and reset with `DkEventClear()`. As with mutexes, calling threads can block for an event by means of `DkObjectsWaitAny()`.

The `%fs` and `%gs` segment selector registers, used to refer to the thread control block and thread local storage data, can be manipulated using the `DkSegmentRegister()` call.

4.5 Processes

The process abstraction supported by the PAL does not assume any implicitly shared state between parent and child. Instead `DkProcessCreate()` creates an independent picoprocess with a new instance of the library OS and PAL. The executable to launch can be specified using a file URI with an accompanying arguments array. The caller receives a RPC handle set up as a communication channel between parent and child. `DkProcessExit()` terminates all threads, ends the process lifetime and returns the given exit status to the host environment.

The Graphene architecture minimizes state in the host OS, by moving it into the user space library OS. However, in order to provide a POSIX environment certain I/O streams, which can not be recreated independently, need to be shared across process boundaries. Examples include opened file streams whose underlying file has subsequently been deleted and bound network sockets. For this reason, the PAL defines the `DkSendHandle()` call to transfer a PAL handle over an established RPC channel from which it can be recovered using `DkReceiveHandle()`. By sharing a handle the primitive exposes host OS state to another picoprocess.

As an optional optimization for inter-process communication, the PAL specifies a fast bulk transfer scheme based on copy-on-write mappings of pages. To avoid the latency of a RPC channel, the host OS is instructed to map the same physical pages into different processes, but keep their changes private by duplicating the pages upon modification.

4.6 Exception Handling

The PAL provides a mechanism to react to hardware exceptions (division by zero, illegal instruction, memory faults), external signals (quit, suspend, resume) and internal failures. `DkSetExceptionHandler()` allows the library OS to register an exception handler for each of these events. The handler function will be upcalled from the host OS with the register state causing the failure. The control flow of the exception handler must end with a call to `DkExceptionReturn()` which destroys the exception stack frame and resumes execution by restoring the possibly modified register state.

4.7 Sandboxing

The PAL defines a sandbox abstraction used to isolate library OS instances. During application startup a static security policy can be specified using a manifest file which white lists accessible directories, hosts and port ranges. The PAL relies on the host OS to enforce isolation policies of the file system, RPC streams and network sockets. Picoprocesses running in the same sandbox can share resources, but cross-sandbox communication is blocked.

4.8 Miscellaneous

The `DkSystemTimeQuery()` PAL call returns the system time in microseconds since the Unix Epoch. Access to a cryptographically secure random number generator is provided via `DkRandomBitsRead()` which might block until enough entropy has been collected. Other PAL interfaces query CPU feature flags (`DkCpuIdRetrieve()`) and configured memory quota available to the picoprocess (`DkMemoryAvailableQuota()`).

Chapter 5

Linux PAL Host

This chapter summarizes how the PAL ABI is implemented on Linux, resulting in a Linux-on-Linux setting with reduced host kernel exposure. The same basic concepts also apply for the FreeBSD host or any other Unix-like system with a monolithic, privileged kernel where state is centralized in a common address space and all required primitives are available with the desired semantics. As a result of the similarities between the PAL abstractions and the native system interface, most PAL calls are thin wrappers around the corresponding host syscalls.

5.1 Virtual Memory

This is exemplified by the virtual memory related interfaces: `mmap()` is used to allocate memory, `munmap()` frees it and `mprotect()` changes the access rights. Conveniently these interfaces provide the required semantics: they operate on page aligned memory regions, support both anonymous and file based mappings, allow arbitrary overlappings. The PAL calls therefore only transform the provided PAL flags and returned error codes.

5.2 I/O Streams

The implementation of the file based I/O stream interface is only slightly more involved. The relevant section of the `PAL_HANDLE` structure stores: the file descriptor referencing the host file, the original file name used to create the handle and the current file position. File I/O operations compare the given file offset to the cached value, adjust it if needed using `lseek()`, invoke the desired `read()` or `write()` syscall and then advance the file position according to the return value.

Other file related stream PAL calls, like set length (`ftruncate()`), flush (`fsync()`) change name (`rename()`), delete (`unlink()`) or meta data related functionality (`fstat()`), are implemented as pass through to the corresponding syscall. By default all file descriptors are marked as close-on-exec to avoid their leakage into child processes.

The RPC streams are by default backed by Unix domain sockets, but can alternatively also use pipes. The network streams are served using the socket

API. The broadcast stream is implemented using a multicast IP socket.

Waiting for events on various types of PAL handles is implemented by passing the underlying file descriptors to `poll()`. Non-blocking semantics of the stream interfaces is provided by setting the `O_NONBLOCK` property of the file descriptor using `fcntl()`.

5.3 Threads and Processes

The PAL also maintains a thread control block (TCB), unlike the one shared by the C runtime and library OS it is referred to by the `%gs` segment register. The TCB references the regular thread stack as well as the alternate signal stack used to execute signal handlers. Thread creation itself is performed by invoking `clone()` with appropriate flags to share the virtual address space and file descriptor table. The synchronization primitives are implemented using a combination of atomic instructions and `futex()` calls.

Process creation itself is performed using a combination of `vfork()` and `execve()`, communication between parent and child is established using a combination of pipes and Unix domain sockets. The latter is also used to implement the `DkSendHandle()` and `DkReceiveHandle()` calls, by means of file descriptor passing using `SCM_RIGHTS` and `sendmsg()`.

As an optimization, the bulk IPC mechanism is implemented using a kernel module, exposing a `/dev/gipc` device with a `ioctl()` interface, enabling sharing of physical pages using a copy-on-write.

5.4 Miscellaneous

Manipulation of the `%fs` segment registers is supported using `arch_prctl()`, `%gs` is reserved for use by the PAL TCB. The `dev:tty` character device is mapped to the standard input/output file descriptors. `/dev/urandom` serves as source of randomness. Queries for the system time are satisfied using either `clock_gettime()` (if available via the vDSO) or `gettimeofday()`. The exception interfaces are implemented by registering corresponding signal handlers which in turn call the PAL exception handlers.

5.5 Security Isolation

The sandbox properties are established using Linux host features. The seccomp-bpf syscall filtering mechanism is used to limit host exposure to the about 50 system calls used by the PAL implementation. To support static binaries, or more generally executables with hard coded syscall instructions, the non-whitelisted syscalls are reflected back into user space to their corresponding implementation within the library OS.

The seccomp-bpf policy has only access to scalar syscall arguments passed in registers, pointer arguments can not be dereferenced for inspection. An extension of the AppArmor Linux Security Module in combination with a trusted user space process acting as a reference monitor is used to enforce security policies by validating arguments to syscalls like `open()`, `connect()` or `bind()`. This restricts host access and isolates independent picoprocesses.

Chapter 6

Barrelfish PAL Host

This chapter documents the porting efforts of the platform adaption layer (PAL) to the Barrelfish multikernel operating system architecture. Challenges experienced while mapping PAL interfaces and their - often implicitly assumed - underlying POSIX semantics are discussed.

6.1 Bootstrapping

While Barrelfish uses the same calling conventions and executable format (ELF) as Linux, it does currently not support dynamic linking of shared libraries. Instead, Barrelfish applications are statically linked and include all their dependencies.

A first attempt tried to leverage earlier work on dynamic linking for Barrelfish[31]. The idea was to build the PAL as a shared library and use native Barrelfish system components to load the environment needed to run the library OS. This approach would require build system modifications to produce shared versions of all involved Barrelfish libraries. Due to the excessiveness of the changes and the relative immaturity of the dynamic linking code, it was ultimately abandoned.

Instead the decision was taken, to reuse the same ELF loader relied up on by the Linux PAL host implementation. The main advantage being, that the GNU C library derived code has already experienced widespread usage and provides robust symbol resolution and dynamic relocation support. Together with the rest of the PAL it is compiled into a static library archive and linked into a regular Barrelfish application, functioning as a PAL loader. To resolve symbol names of the PAL ABI itself, the fall back mechanism of the runtime loader was extended to consult a lookup table with addresses populated at build time by the static linker.

6.2 File Streams

The PAL handle for file abstractions is comprised of: a Barrelfish VFS handle, cached file position and the absolute path name used to open the file.

Such a handle is allocated during the file open call which also needs to support the equivalent of `O_CREAT|O_EXCL` semantics to ensure that the new file did not previously exist. Contrary to that, the Barrelfish `vfs_create()` API succeeds even when the file is already present. Instead, existence is tested by an attempt to open the file: if successful, the exclusive creation request fails. However, this procedure suffers from a time-of-check to time-of-use (TOCTOU) race condition. Between the two RPC calls a third party could successfully create the file, resulting in a PAL handle referencing an existing file. Changing the Barrelfish VFS create API to fail for files which are already present, would result in the desired POSIX behavior.

The read and write I/O operations are mapped to their Barrelfish counterparts. In case the given absolute file offset does not match the cached file position, the VFS handle is repositioned to the desired location. Then the I/O operation is performed and the cached file position updated accordingly. The same technique is used in the Linux host PAL implementation to avoid unnecessary seeks for consecutive I/O.

The Barrelfish VFS API currently lacks a rename primitive. For files the following fall back was implemented:

1. Create or open destination file.
2. Truncate destination file to zero length.
3. Copy content from source to destination.
4. Delete the source file.

However, in case an error occurs this procedure damages an already existing destination file. It also does not guarantee any atomicity as required by POSIX. To fix this, the Barrelfish VFS API should be extended with a rename operation, supported by each backend.

Another important behavioral difference concerns the lifetime of file handles. In a POSIX environment an opened file descriptor remains valid when its underlying path resource is unlinked, while with the current Barrelfish implementation the removal invalidates the file handle and subsequent I/O operations fail.

Meta data queries report the type of a path element as either a directory or file, the latter also has an associated size property. Due to the lack of a permission model everything is reported as world readable, writable and executable.

Other stream related PAL functionality to flush, truncate and close files is provided using equivalent Barrelfish VFS calls.

6.3 Virtual Memory Management

Implementing the virtual memory related PAL calls, using Barrelfish primitives, is complicated by the requirement that both anonymous and file backed memory regions can replace pre-existing memory mappings. In fact this is not a seldom used corner case, but used by the ELF loader during picoprocess initialization. It allocates a contiguous, anonymous virtual memory area and then proceeds to map individual, file backed ELF sections at fixed offsets over it.

Section 2.2 introduced the memory subsystem of Barrelfish and explained how different types of memory objects are used to back virtual regions. More concretely, `memobj_anon` handles page faults of an anonymous memory region by mapping in frame capabilities on demand. Similarly, `memobj_vfs` serves page faults of file mappings by wrapping an anonymous memory object and filling its pages with data read from a VFS handle. With the current implementation existing virtual regions and their backing memory objects can not be resized.

The challenge is how to combine these existing abstractions to build an unified virtual address space with the desired properties. There are multiple plausible options:

- A new memory object type associated with an all-encompassing virtual region. The distinction between anonymous and file backed memory, frame capability management and other metadata bookkeeping would be handled within the memory object. This design ignores much of the existing high-level abstractions of the Barrelfish memory system.
- Splitting of existing virtual regions. When dealing with a request for an overlapped mapping with different backing type, the existing virtual region and associated memory object would be replaced by at most three new ones. One preceding the mapping point, one representing the new mapping and one covering the remainder of the original region. In order for the pre-existing parts to remain unchanged, the data would have to be copied over. Alternatively the ownership of the underlying frames would have to be transferred to the new memory objects.
- Use page sized virtual regions. By breaking up larger allocation and mapping requests into the smallest possible allocation unit, subsequent replacement or invalidation of sub regions becomes trivial. This approach reuses Barrelfish's existing high-level memory allocation functions. However, it has considerable meta data overhead: each page sized chunk allocates an associated virtual region with accompanying memory object. Due to the existence of many virtual regions it also puts strain on the virtual space structure. The sorted double linked list whose algorithms are linear in the number of allocations i.e. pages can become a bottleneck.

The approach based on a new memory object type seemed like a layering violation. Splitting and copying existing regions was disregarded because of its complexity and concerns regarding thread safety. Due to its simplicity, the option using page-sized virtual regions was adopted and any performance optimizations were postponed.

Section 2.4 covered the exception mechanism used to handle page faults in the self-paging paradigm used by Barrelfish. While POSIX defines `sig_atomic_t` and the concept of async signal safety as a list of functions which can safely be executed in a signal handler, the PAL leaves the exact exception handling context unspecified. In any case, because page faults and other exceptions use the same non-reentrant delivery mechanism, the registered PAL handlers must not cause page faults as the resulting double fault would cause immediate thread termination. Possible counter measures include:

- Allow nested exceptions / upcalls as in Psyche[39]. Upon entering the runtime library from the CPU driver, the exception handler would be executed on a new stack frame. Making the upcall delivery mechanism reentrant is the most complex, but also most general approach.

- Decouple page faults from other exceptions. This could for example be implemented using a dedicated stack for page fault handlers. Although this option is not a general solution for all exceptions types, it will nonetheless enable most typical signal handlers.

- Avoid page faults. Pre-fault all memory regions immediately after allocation. This is consistent with other high-level memory allocation functions in Barrelfish and matches the behavior of the limited `mmap()` support in the existing POSIX compatibility library. Like the previous approach it is not a universal solution, but sufficient for the most common case.

A fully reentrant refactoring was deemed too invasive, the decoupling not general enough for the effort. Hence, I accepted the negative performance implication and settled for the pre-faulting strategy.

6.4 Network Sockets

The main architectural decision concerns the logical place of the network stack: should it be part of the application address space like in an exokernel or exposed through a shared system service as in a microkernel?

The PAL stream API defines a blocking interface which is a natural fit for the BSD socket API as demonstrated by the Linux PAL host implementation. Hence, a first attempt operated the lwIP network stack in system mode and used the socket API directly (see section 2.8). As a pre-requisite for the system mode in the newer lwIP version, the OS specific support layer needed to be ported from the legacy lwIP code base. It provides a thread abstraction and synchronization primitives like semaphores, mutexes and mailboxes for cross-thread message exchanges. The Barrelfish network library (libnet) was changed to start the lwIP stack in system mode, using a dedicated network thread instead of periodically polling an event loop. While the socket API matches the basic PAL requirements, moving the network stack into the application complicates sharing of resources in a multi-process setting. It is unclear how the PAL send/receive handle operations for network sockets, as required by a forking server, would be implemented.

The other option is to use the network socket server, a user space service, which multiplexes the networking hardware and provides a callback based RPC interface. Internally it uses the lwIP network stack in raw mode. The main challenge is to wrap the event based API to provide the blocking semantics required by the PAL interfaces. The sending part is unproblematic and works as follows:

1. Set socket status to *sending* and initiate send operation.

2. Enter an event dispatch loop for the waitset handling the RPC communication until the socket status changes.

3. The on-sent handler resets the socket status to *ready*.

4. Return to caller of PAL function.

More challenging is the receiving part. When serving a read call, a similar looping construct is used to wait until data becomes available. However, data can arrive even when no consumer is immediately ready to read it. Hence, it needs to be buffered within the PAL handle representing the socket. Unfortunately the net socket server interface currently provides no rate limiting mechanism, resulting in potentially unlimited buffer growth. Incoming data can also not simply be discarded, because it was already ACKed on the TCP protocol level, meaning the sender will not retransmit.

6.5 RPC Streams

RPC streams are implemented by exposing a Flounder interface featuring a write method to transfer a byte array. Using the existing Barrelfish IDC infrastructure works for communication channels within a single as well as between different dispatchers.

When opening a server pipe stream to which clients can subsequently connect to, the Flounder interface is published to the Barrelfish name server. A mapping is maintained between the published Flounder interface reference identifier (`iref_t`) and the PAL pipe number specified in the URI scheme (`pipe.srv:`). The translation from the latter to the former needs to be accessible and consistent across dispatcher boundaries. Either the file system or the name server itself can be used to look up the interface identfier necessary to bootstrap the communication channel.

The PAL handle representing the server pipe maintains a backlog of connected clients. If said backlog is empty, `DkStreamWaitForClient()` will enter an event dispatch loop until the connected callback is invoked, indicating that a client appeared.

On the client side of such a pipe handle, incoming data is buffered using a linked list of data chunks from which subsequent read attempts are served. The blocking behavior when the pipe buffer is full is implemented using a combination of failing further write requests and notifying the caller once more space becomes available. Non-blocking operation mode is implemented by first querying the `can_send()` method of the underling Flounder channel.

Broadcast streams are provided by `graphened`, a shared system service which exposes a RPC interface with which each library OS instance registers itself during PAL initialization. Incoming messages are forwarded to all other connected clients. Upon process termination the broadcast channel is teared down.

6.6 Threads

Most of the thread related PAL interfaces can be mapped fairly easily to corresponding Barrelfish threading functionality. Examples include PAL calls to yield, resume, delay and exit a thread.

The more interesting parts concern thread creation and in particular handling of thread local storage. For the latter Linux and the PAL ABI expect to set the thread pointer represented by the `%fs` segment register to an arbitrary 64-bit virtual address. However, as was discussed in section 2.5, Barrelfish uses the same segment register for its own thread control block (TCB) and requires it to be located in the lower 4GB of virtual address space. There exist multiple options to resolve these conflicts, let us first discuss different allocation strategies for the TCB:

- Force the allocation of the TCB of the Graphene library OS to the lower 4GB of virtual address space. This would make it compatible with the existing, local descriptor table based, segment management code of Barrelfish. The memory allocation function of the PAL would be extended with a new flag, similar to `MAP_32BIT` as supported by `mmap()` on Linux.
- Change Barrelfish's threading code to support arbitrarily located thread control blocks. The architecture specific register state of both the dispatcher and user level thread structure would hold a full 64-bit virtual address, instead of a 16-bit segment selector. The CPU driver would save/restore its value whenever scheduling a different dispatcher. In the most general case a new system call, similar to Linux's `arch_prctl()`, allows user space to update the segment register upon thread switching. If the CPU supports the FSGSBASE extension, the syscall overhead can be avoided by manipulating the registers directly using unprivileged instructions.

Instead of extending the PAL interface, the more general approach of eliminating existing TCB allocation restrictions in the Barrelfish code was pursued. The new system call exposes segment register state by accessing a corresponding model specific register (MSR) interface. During initialization the CPU driver enables the FSGSBASE extension by manipulating the corresponding bit in the CR4 control register.

While this allows usage of arbitrarily located thread control blocks, it does not resolve the conflict that both the Linux and Barrelfish environments demand control over the `%fs` register. The following alternatives were considered to remedy the situation:

- Use *distinct* segment registers as thread pointer. Because the goal is to run unmodified Linux binaries, Barrelfish would need to leave `%fs` untouched. Existing usage in `curdispatcher()` and `thread_self()` would have to migrate to `%gs`. While the current Barrelfish code base does not use thread local storage, these changes would also diverge from the x86-64 ELF TLS ABI, requiring support from the compiler toolchain.

  The resulting setting would be similar to that taken advantage of by Wine: Windows and Linux use different TLS ABIs, meaning each environment can use its native TCB access mechanism without causing a conflict.
- *Virtualize* the `%fs` segment register. While there is only one physical resource, make sure that both the Barrelfish and Linux environments always run in a context with the segment register set to their expected value.

Using a different TLS ABI was dismissed as being too invasive. Instead, the correct segment register state is restored whenever crossing the PAL boundary separating the Linux and Barrelfish environments. The Barrelfish PAL introduces its own thread control block, keeping track of the user level thread, its exception stack and the current value of the `%fs` segment base address as seen by the Linux library OS and Barrelfish, respectively.

```
struct pal_tcb {
    struct pal_tcb *self;
    int (*entry)(void *param);
    void *param;
    void *exception_stack;
    void *fsbase_linux;
    void *fsbase_barrelfish;
    PAL_HANDLE thread;
};
```

Listing 6.1: Barrelfish PAL thread control block.

The PAL TCB is initialized within the newly created thread context and is pointed to by the `%gs` register, which is reserved for this purpose and must remain unchanged during the thread's lifetime. The initial `%fs` value, as assigned by Barrelfish's threading library, is recorded. The exception stack allocated and the exception handler registered. Finally, control is passed to the caller supplied thread entry point.

`DkSegmentRegister()` simply returns or updates the `fsbase_linux` field of the PAL TCB and rejects any modification of `%gs` which is also unused by Barrelfish.

Whenever entering (leaving) the PAL the TCB is fetched through `%gs`, and the Barrelfish (Linux) `%fs` segment register value is restored. More concretely, this "world switching" takes place within preprocessor macros used by the generic portion of the PAL.

As for the thread synchronization primitives: PAL mutexes are directly mapped to their Barrelfish equivalents, while synchronization and notification events are provided using a combination of atomic instructions and conditional variables. Atomic integers are used to track the number of waiters and whether the event is signaled or not. Synchronization events release one waiter at a time by signaling the underlying conditional variable, while notification events invoke its broadcast operation.

6.7 Processes

Barrelfish spawns new domains using a RPC call to a process manager service. During this invocation program arguments, environment variables and capabilities can be passed to the new domain. Otherwise no implicit state is shared. The `DkProcessCreate()` PAL call is implemented by spawning the Barrelfish application `graphene` which operates as a PAL loader, as described in the bootstrapping section. The path denoting the Linux binary to execute is prepended to the actual program arguments.

The PAL calls to share handles across process boundaries are challenging to implement in the existing Barrelfish setting. Unlike in other PAL hosts there exists no monolithic kernel which centralises state and can mediate access to resources by copying or reference counting corresponding kernel objects. The Barrelfish analogon to the file descriptor passing of Unix systems is the capability transfer. However, user space system APIs like the VFS or the socket server interface use their own, non-capability based, resource handles.

Two plausible options to remedy the situations are:

- Extend each system service with an API to share a resource handle between domains.
- Independently recreate an equivalent resource handle in the receiving domain. However, because the sender is not obligated to close its handle after sharing it, this requires that all associated state is maintained by the caller.

The second approach is used for file streams. The sender first persists all pending changes by flushing the handle, then transmits the full path over the stream. The receiver proceeds to create a PAL handle by opening the same file. This suffers from an obvious race condition and fails for files that have since been unlinked, which was a motivating example for the send/receive interface. The common case works because each read/write call takes an absolute file position maintained by the library OS.

For network sockets no such workaround is possible. The network socket server would require the aforementioned interface extension to make the socket available to another domain. RPC streams can be rebound in the receiving domain, but the way the existing Flounder mechanism is used only works for point-to-point communication. An additional intermediate layer would be necessary in case the sender keeps its handle alive.

6.8 Exception Handling

Recall how exception handling works in Barrelfish (see section 2.4). The CPU driver upcalls into libbarrelfish where execution starts on the dispatcher stack, before calling the thread exception handler on the thread exception stack. During thread initialization the PAL registers such a Barrelfish exception handler which presents a different PAL entry point. The execution context switches from the Linux application into the Barrelfish system library. As such the segment register virtualization needs to take place. The dispatcher context has its segment register saved in the DCB. Within the exception handler the PAL TCB can be recovered from the `%gs` register value as recorded at the time of the exception. Therefore, the thread pointer of the Linux environment can be restored before invoking the function registered with `DkSetExceptionHandler()`.

To support the `DkExceptionReturn()` PAL call, libbarrelfish was extended with a new function which clears the thread's exception state and resumes the dispatcher with the given register state.

As was previously discussed in the context of virtual memory, the Barrelfish exception mechanism is non-reentrant. Meaning that exception handlers must not cause further exceptions. In particular, nested signal handlers, registered with `sigaction()` and the `SA_NODEFER` flag, can not be supported using the provided PAL implementation.

6.9 Miscellaneous

The `dev:tty` device is implemented using a global lock around the read/write RPC interface provided by the serial sub system[23].

Because Barrelfish currently does not provide a native random source, the `rdrand` instruction is used directly to generate random bytes. Failing that, or during development and debugging, a constant byte stream is returned. In any case, the current implementation is not suitable for serious cryptographic applications.

The time stamp counter is used as a time source. During initialization the CPU driver measures the number of cycles per millisecond and exposes the data through a syscall. This information is subsequently used to convert the elapsed cycles to a time interval.

Chapter 7

Evaluation

In this chapter the introduced overhead of PAL calls compared to their underlying native Barrelfish interfaces is measured. Challenges, resulting from Graphene's architecture, in providing the expected POSIX semantics of shared resources in a multi-process setting are discussed.

7.1 Performance

A number of PAL functions, exemplified by the file I/O operations, merely pass on incoming arguments to the corresponding native Barrelfish interfaces. Hence, the performance primarily depends on existing Barrelfish system components. However, the virtualization of the `%fs` segment register, as discussed in section 6.6, introduces a constant overhead whenever crossing PAL boundaries.

In order to quantify the performance implications of this "world switching", a number of micro benchmarks were carried out on a dual-socket Intel Xeon E5-2670 v2 @ 2.50GHz "Ivy Bridge" machine with 10 cores per socket and 256GB of main memory. The existing Barrelfish benchmarking infrastructure was used. The presented cycle counts are averages over 10^6 iterations as reported by the `rdtsc` instruction. Each benchmark was repeated 10 times, demonstrating stability by yielding the same results.

Recall the two possibilities to set a the full 64-bit virtual address of a segment register:

- Using the unprivileged `wrfsbase` instruction, available in Intel CPUs starting from the Ivy Bridge microarchitecture.
- By issuing a system call which uses the privileged `wrmsr` instruction to change the corresponding model specific register (MSR).

To provide more context, the raw syscall overhead and micro benchmark of the `wrmsr` instruction, as measured by the CPU driver during initialization, is also reported in table 7.1.

Method	Cycle count
Unprivileged instruction (`wrfsbase`)	35
Set segment register syscall	248
No operation (NOP) syscall	129
Privileged instruction (`wrmsr`)	134

Table 7.1: Micro benchmarks manipulating segment registers.

The unprivileged instruction is approximately 7 times faster than the MSR based system call. It is a worthwhile optimization and a good match for Barrelfish's user level threading model, featuring fast thread switching. Basing the `%fs` segment register virtualization on it, keeps the constant cost added to each PAL function invocation below Barrelfish's syscall overhead.

Interestingly, the summation of the results for the NOP syscall and the `wrmsr` instruction does not provide a lower bound for the measurements of the set segment register syscall which essentially just combines the two. One possible explanation is that the MSR micro benchmark is ran during CPU driver initialization, while the others are initiated from user space once the whole OS is operational.

More generally, performance optimizations have so far been of secondary concern. The virtual memory related interfaces in particular, have proven to be a bottleneck. Initial investigations revealed, that the dominating cost is not due to the page-sized mapping approach described in section 6.3, but because of the naive frame allocation strategy involving expensive capability retype operations.

7.2 POSIX Compatibility

The previous chapter already mentioned cases where the presented Barrelfish PAL host implementation fails to provide the required POSIX behavior. Examples included the VFS interfaces (missing rename primitive, lack of atomicity, invalidation of open file handles upon deletion) and the limitations of the exception mechanism. While these issues could be resolved by adapting the relevant Barrelfish system components, in this section a more fundamental issue related to Graphene's current implementation of the multi-process abstractions shall be discussed.

Listing 7.1 shows the C source code of a minimal example, illustrating behavior related to the file offsets of shared file descriptions. For brevity, header includes and error checking has been omitted.

```
int main(void)
{
  const char *msg;

  int fd = open("demo", O_WRONLY|O_TRUNC|O_CREAT, 0600);

  if (fork() > 0) {
    wait(NULL);
    msg = "Parent\n";
  } else {
    msg = "Child\n";
  }

  write(fd, msg, strlen(msg));
}
```

Listing 7.1: Behavior of a shared file description.

The program first creates a file, then forks off a child and lets it write through the inherited file descriptor. After the child has terminated, the parent also writes to the file. On a conforming system, the resulting file content is therefore:

```
Child
Parent
```

This works because both process specific file descriptors point to the same open file description, holding the file offset and status flags, within the system-wide table of open files. Hence, when the child writes its message, the file offset is advanced and the subsequent write of the parent is placed after it.

Now recall that in Graphene's design the state of the open file description, including the current file offset, is maintained independently in each library OS instance. Because this state is currently not replicated among different pico-processes, the file offset as seen by the parent remains unchanged by the child's write. Consequently, the parent overwrites the previous message, resulting in a file storing just:

```
Parent
```

The example illustrates how the POSIX interfaces encourage state centralisation and complicate the distributed library OS design of Graphene.

Chapter 8

Related Work

The idea of putting more OS functionality into application libraries goes back to Anderson[1]. Initially such systems like the Exokernel[21] aimed to provide performance advantages by exposing low level hardware interfaces.

Roscoe et al.[48] questioned hardware based virtual machine interfaces and argued for high-level virtualization abstractions.

The notion of a *picoprocess* as an isolated entity mapping a high-level guest API to a narrow (syscall) interface originated from attempts to run applications in a web context. Xax[18] achieved OS independence by specifying an ABI implemented by a platform abstraction layer. Embassy[27] called a similar concept client execution interface (CEI), which was subsequently employed to run single-process POSIX applications in a minimal picoprocess[28].

Drawbridge[43] refactored Windows 7 into a library operating system capable of running major applications. Benefits included improved application mobility and isolation with lower overhead than traditional virtual machines. The introduced Drawbridge ABI provides a small set of OS abstractions: threads, virtual memory and I/O streams.

Bascule[9] provides portable and composable extensions (e.g. tracing and checkpointing) for library operating systems based on interposition techniques of the Drawbridge ABI. The Bascule ABI made all I/O operations asynchronous and introduced new calls for exception handling and thread local storage. The latter provides a way to allocate memory for TLS at thread creation time, thereby avoiding the a system call required in the general solution when crossing PAL boundaries. To showcase that extensions are both independent from the library- as well as from the host operating system, a Linux library OS personality was implemented and the Bascule ABI was ported to Barrelfish.

Graphene[52] demonstrated that the library OS approach can also be adapted to multi-process applications by implementing a distributed POSIX implementation. The PAL ABI (see chapter 4) is adapted from Drawbridge with extensions for library OS state coordination and sandboxing mechanisms. Other changes related to exception handling and TLS setup were already present in Bascule.

Haven[10] and Graphene-SGX[53] leverage the Intel SGX hardware and library operating systems to shield unmodified applications from untrusted hosts. Haven is based on Drawbridge and uses a library version of Windows 8, while Graphene-SGX is a SGX host implementation of the Graphene PAL, thereby enabling a Linux execution environment.

Plan 9[42] only provides a narrow syscall interface which shares similarities with the Graphene ABI. The byte stream abstraction used by the 9P network protocol, exposed through the file system, to talk to server processes, resembles the RPC stream used in Graphene to coordinate library OS state.

Unikernels[37, 38] combine single-purpose applications with OS functionality in a single address space image expected to run on hypervisors or bare metal. Because there is no need for protection, syscalls become function calls with no change in privilege level. Neither page table switches nor TLB flushes are necessary. This specialization allows the omission of unneeded functionality and enables whole system optimizations leading to fast boot times and low memory footprints. OSv[32] and HermiTux[41] are Linux compatible Unikernels employing binary rewriting and dynamic library substitution techniques. Unikernel Linux[46] is an attempt to turn Linux itself into a Unikernel.

User Mode Linux (UML)[16] runs Linux as a regular user space process. However, unlike with a picoprocess there is no deduplication of host OS functionality. Instead UML is implemented as an additional architecture of the Linux kernel using host system calls. It has been characterized as an "alternative approach to paravirtualization[4]".

Rumpkernels[29, 30] enable reuse of compartmentalized drivers of a monolithic kernel by means of a hypercall interface. As an example the NetBSD originated rump kernel hypercall interface[40] covers: virtual memory, files and I/O, exception handling, access to clock and randomness sources as well as thread creation, scheduling and synchronization primitives. As such it is similar to the picoprocess ABIs used by Drawbridge, Bascule and Graphene. By reusing the syscall interface (driver) it is possible to run POSIX-like applications on platforms implementing the hypercall interface. However, compared to Graphene limitations include: only anonymous mmap, no signals and no multi process support.

Containers[50] virtualize core kernel data structures to provide lightweight isolation among processes. Unlike with library operating systems, the OS personality does not reside within the application address space. The exposed syscall interface is still huge, presents an attack surface and achieves no host OS independence.

Google's gVisor[25] improves container security by implementing the Linux syscall interface in a "user space kernel" (Sentry) thereby limiting host kernel exposure. Depending on the mode of operation either Linux virtualization (KVM) or process tracing (ptrace) techniques are used for system call redirection. Compared to Graphene the implementation leverages a high-level language (Go), includes a user space network stack and delegates the acquisition of file handles to a separated process (Gofer) accessible over the 9P protocol. Instead of depending on a platform neutral adaption layer, host interaction is performed directly using a reduced set of Linux syscalls.

K42[34] is a scalable operating system using object orientation to decompose OS functionality and reduce state sharing. The Linux personality[3] of K42 is mostly implemented in userspace. As in Graphene, the syscall invocation within glibc is redirected to call into library functions residing in the application's own address space. In a critique of fork[7] the authors reflect on their experience supporting it in K42, how it affects OS architecture and encourages centralization of state.

NetBSD[49], FreeBSD[17] and Solaris/illumos have emulation layers supporting unmodified Linux binaries. These operating systems share the typical Unix architecture of a monolithic privileged kernel. The concepts are mostly the same as on Linux but the implementations differ slightly. Linux emulation is provided by using a different syscall table and a translation layer fixing up the calling convention (e.g. whether syscall arguments are passed on the stack or in registers), errno values and signal machinery. The Linux ABI is mapped to corresponding native system calls.

The first version of the Windows Subsystem for Linux[54] uses picoprocesses to virtualize the Linux syscall ABI on top of the Windows NT kernel to run unmodified Linux binaries.

Wine[55] (for Windows PE binaries) and Darling[15] (for macOS Mach-O executables) are other projects providing a familiar application execution environment on foreign operating systems. While they provide certain kernel functionality in user space services, they do neither refactor an existing code base into a library nor rely on a narrow PAL ABI. Instead, they have to undertake massive porting efforts to re implement a vast API.

Most closely related to my work is a previous Bachelor Thesis[13] which attempted to port Graphene to Barrelfish. It cited the lack of dynamic linking in Barrelfish as a reason why Linux binaries could not be run unmodified on Barrelfish. The presented work circumvents this obstacle by reusing an ELF loader within the PAL.

Chapter 9

Conclusion & Future Work

In its current form the presented work is able to run simple, unmodified, dynamically linked, single-process Linux applications on Barrelfish using the Graphene library OS. Supported features include threads, signals and I/O operations on pipes, sockets and files. Major challenges during the porting effort included: bootstrapping due to lack of native dynamic linking support, TLS resource handling and the construction of a unified virtual address space comprised of both anonymous and file backed memory regions. These were overcome by reusing an ELF loader with a fallback for symbol resolution based on a static lookup table embedded into the PAL library, virtualizing the `%fs` segment register by performing a "world switch" whenever crossing PAL boundaries and splitting up larger memory management tasks into operations on page sized allocations.

During development it became clear that the existing Barrelfish paradigms are often not a good match for POSIX - and by extension PAL - interfaces. Barrelfish's network socket server is event and callback based, whereas the BSD socket API provides blocking calls. The current VFS implementation does not cope with file handles whose underlying file system objects have been removed and lacks rename and hard link primitives as well as a permission model. Barrelfish's memory system enables a great deal of flexibility, but the provided memory objects as high-level user space abstractions do not easily mix anonymous and file backed memory regions. The self-paging and non-reentrant exception mechanism puts restriction on the code of exception handlers and precludes nested signal handlers.

More importantly POSIX encourages centralisation of state. This was recently discussed in the context of `fork()`[7] but does also apply, albeit to a lesser extend, to the underlying PAL interfaces enabling it in the library OS. Functionality to pass PAL handles between processes is straightforward to provide in a monolithic kernel where state is centralised and kernel objects can easily be reference counted or copied, but more involved in distributed OS services. It is even more complex for state solely kept in the library OS within the application's own address space. A concrete example of this is the file seek position which is currently not shared even though the Graphene authors claim that it "would be a straightforward extension to current RPC mechanisms"[52].

In my opinion the Graphene PAL ABI could benefit from a more detailed and rigorous specification. Often the semantics of certain interfaces is implicitly assumed to match POSIX behavior - conveniently supported by existing host implementations - instead of explicitly defined. Examples include: execution context of PAL exception handlers and their reentrant behavior, atomicity of I/O operations, thread safety requirements and state of file streams whose underlying file system object has been deleted.

Resolving the previously mentioned limitations related to the VFS and multiprocess abstractions as well as performance optimizations of the the virtual memory interfaces is left for future work. The substitution technique used to redirect system calls into the library OS only works for dynamically linked executables. To also support statically linked applications, system call redirection based on binary rewriting or trap reflection into userspace, as suggested by the Bascule[9] authors, would be needed. The sanboxing policy is currently not enforced, meaning independent library OS instances are not isolated and cross-sandbox communication is not prohibited. Future work could explore a setting with improved isolation guarantees where distinct, per-sandbox instances of system services are used.

To conclude, while the presented system is capable of running simple, single-process Linux applications, it is still a long way off a fully conformant distributed POSIX implementation.

Bibliography

[1] T. E. Anderson. “The case for application-specific operating systems”. In: *Proceedings Third Workshop on Workstation Operating Systems.* Apr. 1992, pp. 92–94.

[2] 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: *SIGOPS Oper. Syst. Rev.* 25.5 (Sept. 1991), pp. 95–109.

[3] Jonathan Appavoo, Marc A Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan S Rosenburg, Robert W Wisniewski, and Jimi Xenidis. “Providing a Linux API on the Scalable K42 Kernel.” In: *USENIX Annual Technical Conference, FREENIX Track.* 2003, pp. 323–336.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. “Xen and the Art of Virtualization”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles.* SOSP ’03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177.

[5] Team Barrelfish. *Barrelfish Architecture Overview.* Tech. rep. 000. Version 2.0. Systems Group, ETH Zurich, 2013.

[6] Andrew Baumann. *Inter-dispatcher communication in Barrelfish.* Tech. rep. 011. Version 0.3. Systems Group, ETH Zurich, 2011.

[7] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. “A fork() in the road”. In: *17th Workshop on Hot Topics in Operating Systems.* ACM, May 2019.

[8] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles.* SOSP ’09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44.

[9] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. “Composing OS Extensions Safely and Efficiently with Bascule”. In: *Proceedings of the 8th ACM European Conference on Computer Systems.* EuroSys ’13. Prague, Czech Republic: ACM, 2013, pp. 239–252.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.* OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 267–283.

[11] 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 Conference on Hot Topics in Operating Systems.* HotOS'09. Monte Verit&#224;, Switzerland: USENIX Association, 2009, pp. 12–12.

[12] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "User-level Interprocess Communication for Shared Memory Multiprocessors". In: *ACM Trans. Comput. Syst.* 9.2 (May 1991), pp. 175–198.

[13] Yves Bieri. "Running Linux Binaries over Barrelfish using a LibraryOS". Bachelor's Thesis. ETH Zurich, 2015.

[14] David D. Clark. "The Structuring of Systems Using Upcalls". In: *Proceedings of the Tenth ACM Symposium on Operating Systems Principles.* SOSP '85. Orcas Island, Washington, USA: ACM, 1985, pp. 171–180.

[15] *Darling - macOS translation layer for Linux.* `https://www.darlinghq.org/`. Accessed: August 2019.

[16] Jeff Dike. *A user-mode port of the Linux kernel.*

[17] Roman Divacky. "Linux emulation in FreeBSD". Master's Thesis. 2016.

[18] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. "Leveraging Legacy Code to Deploy Desktop Applications on the Web". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation.* OSDI'08. San Diego, California: USENIX Association, 2008, pp. 339–354.

[19] Ulrich Drepper. *ELF Handling For Thread-Local Storage.* Version 0.21. 2013.

[20] Adam Dunkels. "Design and Implementation of the lwIP TCP/IP Stack". In: *Swedish Institute of Computer Science* 2 (2001), p. 77.

[21] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. "Exokernel: An Operating System Architecture for Application-level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266.

[22] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. "Interface and Execution Models in the Fluke Kernel". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation.* OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 101–115.

[23] Raphael Fuchs. "A session control interface for a Multikernel". Bachelor's Thesis. ETH Zurich, 2012.

[24] Simon Gerber. "Authorization, Protection, and Allocation of Memory in a Large System". PhD thesis. ETH Zurich, 2018.

[25] *gVisor Architecture Guide.* `https://gvisor.dev/docs/architecture_guide/`. Accessed: August 2019.

[26] Steven M. Hand. "Self-paging in the Nemesis Operating System". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation.* OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 73–86.

[27] Jon Howell, Bryan Parno, and John R. Douceur. "Embassies: Radically Refactoring the Web". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation.* NSDI'13. Lombard, IL: USENIX Association, 2013, pp. 529–546.

[28] Jon Howell, Bryan Parno, and John R. Douceur. "How to Run POSIX Apps in a Minimal Picoprocess". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference.* USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 321–332.

[29] Antti Kantee. "Flexible operating system internals: the design and implementation of the anykernel and rump kernels". PhD thesis. Aalto University, 2012.

[30] Antti Kantee and Justin Cormack. "Rump kernels: No os? no problem!" In: *Login: USENIX Magazine* 39.5 (2014).

[31] David Keller. "Dynamic Linking and Loading in Barrelfish". Bachelor's Thesis. ETH Zurich, 2015.

[32] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. "OSv: Optimizing the Operating System for Virtual Machines". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference.* USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 61–72.

[33] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles.* SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220.

[34] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. "K42: Building a Complete Operating System". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006.* EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 133–145.

[35] 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.

[36] Jochen Liedtke. "Improving IPC by Kernel Design". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles.* SOSP '93. Asheville, North Carolina, USA: ACM, 1993, pp. 175–188.

[37] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. “Unikernels: Library Operating Systems for the Cloud”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: ACM, 2013, pp. 461–472.

[38] Anil Madhavapeddy and David J. Scott. “Unikernels: Rise of the Virtual Library Operating System”. In: *Queue* 11.11 (Dec. 2013), 30:30–30:44.

[39] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. “First-class User-level Threads”. In: *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. SOSP ’91. Pacific Grove, California, USA: ACM, 1991, pp. 110–121.

[40] *NetBSD Rump Kernel Hypercall Interface*. `https://man.netbsd.org/cgi-bin/man-cgi?rumpuser`. Accessed: August 2019.

[41] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. “A Binary-compatible Unikernel”. In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Providence, RI, USA: ACM, 2019, pp. 59–73.

[42] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. “Plan 9 from Bell Labs”. In: *In Proceedings of the Summer 1990 UKUUG Conference*. 1990, pp. 1–9.

[43] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. “Rethinking the Library OS from the Top Down”. In: ASPLOS XVI (2011), pp. 291–304.

[44] Barrelfish Project. *Barrelfish Glossary*. Tech. rep. 001. Version 2.0. Systems Group, ETH Zurich, 2013.

[45] Richard Rashid, Avadis Tevanin Jr., Michael Young, David Golub, and Robert Baron. “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”. In: *IEEE Trans. Comput.* 37.8 (Aug. 1988), pp. 896–908.

[46] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. “Unikernels: The Next Stage of Linux’s Dominance”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’19. Bertinoro, Italy: ACM, 2019, pp. 7–13.

[47] Kaveh Razavi. “Barrelfish Networking Architecture”. Distributed Systems Lab Report. ETH Zurich, 2010.

[48] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. “Hype and Virtue”. In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. HOTOS’07. San Diego, CA: USENIX Association, 2007, 4:1–4:6.

[49] Peter Seebach. *Implementing Linux emulation on NetBSD*. `https://www.linux.com/news/implementing-linux-emulation-netbsd`. May 2014.

[50] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 275–287.

[51] Chia-Che Tsai. "A Library Operating System for Compatibility". PhD thesis. State University of New York at Stony Brook, 2017.

[52] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. "Cooperation and Security Isolation of Library OSes for Multi-process Applications". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 9:1–9:14.

[53] Chia-Che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 645–658.

[54] *Windows Subsystem for Linux*. `https://blogs.msdn.microsoft.com/wsl/`. Accessed: August 2019.

[55] *WineHQ - Run Windows applications on Linux, BSD, Solaris and macOS*. `https://www.winehq.org/`. Accessed: August 2019.

ETH

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):

Running Linux binaries over Barrelfish using a Library OS

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)**
Biel, 1. September 2019	[illegible]

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