Bachelor's Thesis Nr. 138b

Systems Group, Department of Computer Science, ETH Zurich

Running Linux Binaries over Barrelfish using a LibraryOS

by

Yves Bieri

Supervised by

Prof. Timothy Roscoe,
Dr. Kornilios Kourtis,
Gerd Zellweger,
Simon Gerber

March 2015 – September 2015

inf | Informatik
Computer Science

Abstract

Today's operating systems (OS) have to keep up with the quick evolution of both hardware and software. They face the challenge of optimally running applications on the underlying hardware. Due to extremely fast development of the hardware market, it appears to be an insurmountable job for an operating system to keep up and run the application optimally on the built-in hardware.

Library operating systems present a promising approach to solve this problem. A library OS moves the implementation and management of system objects like virtual memory to application space, while letting the guest OS handle hardware resource protection. Creating a library out of an OS kernel allows applications to choose a library that implements system objects in an optimal way. The library OS communicates with the guest OS kernel using an abstract binary interface. The library OS can be run on any host OS, as long as there exists a host OS specific implementation of this small interface.

This thesis focuses on implementing this interface, called Platform Adaptation Layer (PAL), for the Graphene library OS on the Barrelfish research operating system. Graphene encapsulates the Linux OS into a library. We discuss the major challenges of implementing the PAL on Barrelfish and discover that due to the lack of dynamic linking in Barrelfish, Graphene cannot yet be completely implemented. Nevertheless, we were able to investigate the performance of PAL's event signaling. The key conclusion of this thesis is, that signaling can be up to an orders of magnitude slower on Barrelfish than on Linux. With some modifications to the timeslice length of the Barrelfish scheduler nearly the same performance can be achieved as on Linux.

Acknowledgements

I want to thank Prof. Timothy Roscoe for giving me the opportunity to work on Barrelfish; Dr. Kornilios Kourtis, Gerd Zellweger and Simon Gerber for their valuable insights and answers to my questions. Further I would like to thank the other people of the ETH Zurich Systems Group for their assistance and the anonymous reviewers for their constructive comments.

Contents

1 Introduction **3**

2 Related Work **5**
- 2.1 Graphene 5
  - 2.1.1 Architecture 5
  - 2.1.2 Multi-process Applications 7
- 2.2 Barrelfish 8
  - 2.2.1 Multikernel Architecture 9
  - 2.2.2 Memory Management and Capabilities 10
  - 2.2.3 Filesystem 10
- 2.3 Drawbridge 11
  - 2.3.1 Design & Architecture 11

3 Design & Implementation **13**
- 3.1 System Structure 13
- 3.2 Platform Abstraction Layer (PAL) 15
  - 3.2.1 PAL_HANDLE 15
  - 3.2.2 Threading 15
  - 3.2.3 File 16
  - 3.2.4 Memory 18
  - 3.2.5 Signals 19

4 Benchmark **25**
- 4.1 Implementation 25
- 4.2 Results 26

5 Conclusion **29**
- 5.1 Future Work 29

Bibliography **31**

1 Introduction

New hardware is released at an enormous rate so that programs can run faster and faster. But not only the hardware in a computer determines how fast a program will run. It is also highly dependent on the operating system in use. The operating system can be a limiting factor for the performance of a running application. Some OS provide high level abstractions like files, processes etc. to the application. Those abstractions cannot be modified and thus cannot be optimized for each single program you want to execute.

In 1995, Engler, Kaashoek and O'Toole Jr. [1] developed a new OS architecture, the exokernel, that tries to solve this problem by moving the hardware resources to a library OS. A library OS (libOS) is an OS collapsed into an application library. The library OS uses its own implementation of system objects like virtual memory by using hardware resources provided by a small interface. Through its own implementation of those system objects, not only can they be adapted to and optimized for a specific application but also to provide security isolation. Applications running on top of a libOS gain the advantages of virtualization. Processes running in different virtual spaces cannot interfere with each other. AlibOS can achieve virtual machine like security isolation but with much less overhead than a classic virtual machine, thanks to not having to duplicate features like hardware management for host and guest OS.

This thesis focuses on implementing a Linux library OS on top of the Barrelfish [2] operating system. Barrelfish is a research operating system developed by the Systems Group of ETH Zurich together with Microsoft Research. OSes optimize for the most common hardware, but as hardware becomes more diverse, optimization gets more and more difficult. Barrelfish tries to solve this problem using an implementation of a multikernel. A multikernel treats a single machine with multiple cores as a distributed network. By doing so, an OS scales better, as expensive locking of shared data structures can be avoided. Instead the different cores in the multikernel network communicate via message-passing. For native applications Barrelfish already uses a libOS but it cannot yet run Linux applications.

2 Related Work

2.1 Graphene

Graphene is a library OS developed by Tsai et al. [3]. A library OS is designed to run a single-process application in a virtual machine (VM) like environment. To achieve this, an already existing operating system kernel is refactored into an application library. The functions of this library implement OS system calls by mapping the abstract programming interface (API) of the guest OS to the host kernel.

The Graphene library OS was first implemented on top of a Linux host. Previous to Graphene, most library OSes were designed for single-process applications (Tsai et al., 2014). Many applications, like shell scripts or Makefiles, use fork, execve or similar syscalls. Those require multi-process support. The goal of Graphene is to extend the libOS functionality to multi-process applications while keeping memory and performance overheads to a minimum. To execute a multi-process application, Graphene assigns each process a new libOS instance. Those instances collaborate to support multi-process functionality, but appear as a single shared OS to the application. The different instances communicate by using remote procedure calls (RPCs) over byte streams (similar to Linux pipes).

2.1.1 Architecture

Figure 1 shows the architecture of Graphene. An instance of Graphene runs an unmodified host binary and additional libraries, needed for the binary's execution. The execution takes place within a picoprocess. A picoprocess is ”(...) a native-code execution abstraction that is secured via hardware memory isolation and a very narrow system-call interface, akin to a streamlined hardware virtual machine” [4]. Graphene modifies the standard GNU libc to redirect system calls to a shared library named libLinux.so and then uses a Platform Adaptation Layer (PAL) to expose a host binary interface comprised of 43 methods for managing functionality like virtual memory, networking and a file system to libLinux.

When an application calls one of these functions (e.g through a malloc call) the PAL translates the call to the abstract binary interface (ABI) into the corresponding system calls for the host kernel.

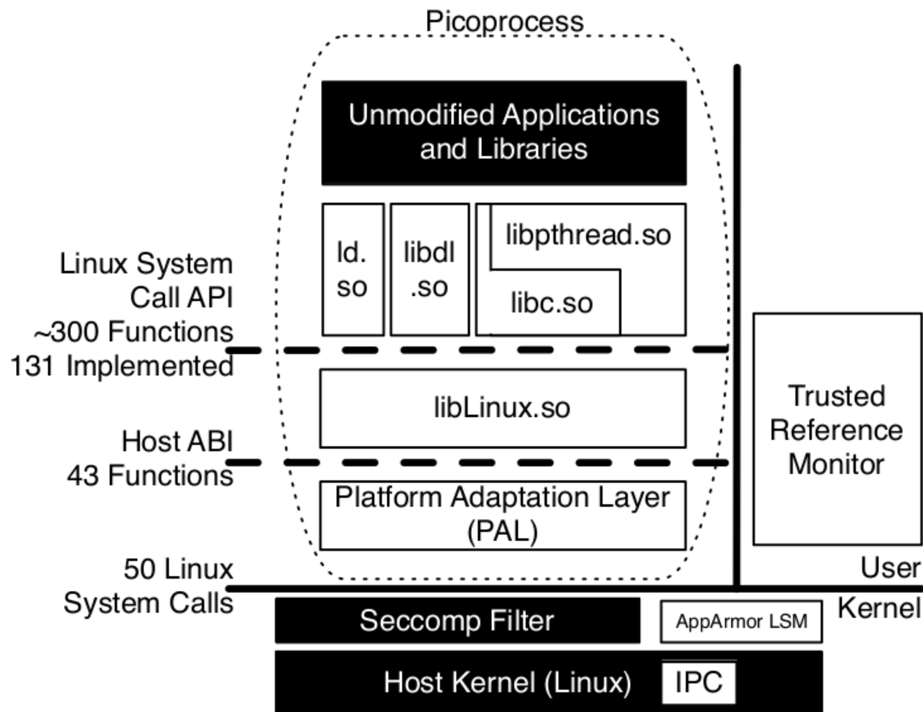

Figure 1: Graphene Architecture

Let us analyse this more thoroughly with an example. Let's assume an application wants to allocate a buffer on the heap. This is done using the malloc function:

```c
int * buf = malloc(sizeof(int)*10);
```

The malloc call is provided by the adapted Graphene libc. Upon calling `malloc()`, the Graphene libc calls libLinux, which knows how to translate the malloc request to a PAL call. Malloc corresponds to the PAL call `DkVirtualMemoryAlloc()`. In the PAL, `DkVirtualMemoryAlloc()` now requests memory from the host (Linux) by issuing an mmap syscall.

The PAL exposes a generic set of ABIs. At the moment, the target host OS is Linux. However, Graphene can be ported to another host OS by implementing the PAL ABIs using the new host's system calls. Table 1 gives an overview of the ABIs that have to be implemented on a host for Graphene to work. They consist of some ABIs proposed by the Drawbridge [5] library OS plus some additional ABIs introduced in Graphene to handle multi-process applications.

Adopted from Drawbridge		
Class	**ABIs**	**Description**
Memory	3	Allocate and protect virtual memory.
Scheduling	12	Threads and synchronization.
Files & Streams	12	Files inside a `chroot`-style jails and byte streams among picoprocesses.
Process	2	Create a child picoprocess, and exit self.
Misc	4	Get random bits, time of day, etc.
Added by Graphene		
Class	**ABIs**	**Description**
Segments	1	Manage x86 segment registers for TLS.
Exceptions	2	Handle hardware exceptions.
Streams	3	Share stream handles and rename files.
Bulk IPC	3	Exchange copy-on-write pages.
Sandboxes	1	Move into a new sandbox, closing handles to other picoprocesses.

Table 1: Host ABI functions implemented in Graphene, taken from [3]

2.1.2 Multi-process Applications

To support multi-process applications, Graphene sets up a new libOS instance for each process. These different ”picoprocesses” work together to implement multi-process functionality like signals or fork. If process 1 wants to signal process 2, it uses a remote procedure call to the second libOS instance over a byte stream, as shown in Figure 2. The second libOS then reacts to the signal by calling the corresponding signal handler.

Graphene handles the RPC using a so called interprocess communication (IPC) helper thread in each picoprocess. The IPC helper listens to all connected streams and then can respond to remote requests or send himself events to other external processes. The RPCs are not only used for distributed processes like execve, fork and signal, but also to keep a consistent namespace across picoprocesses. All libOS instances have the same set of used variable names, threads IDs etc. This model has the additional benefit of simple security isolation by being able to block all RPCs to a picoprocess.

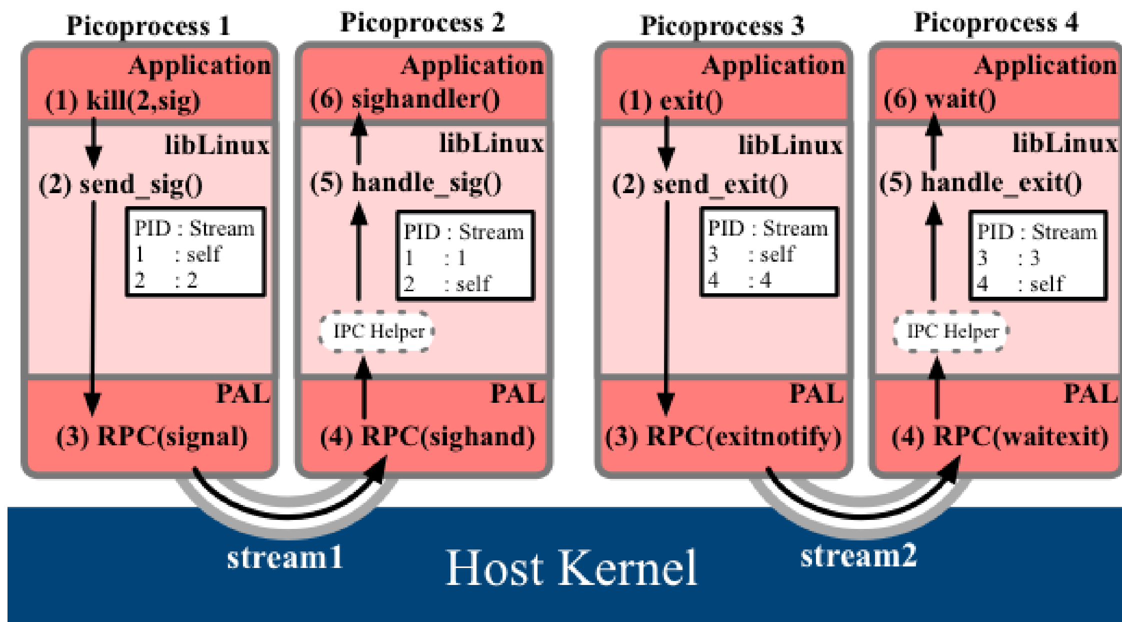

Figure 2: Signaling and Exit in Graphene. Taken from [6]

2.2 Barrelfish

In this section we will take a look at some of the design decisions and implementations in the Barrelfish [2] research operating system that are relevant for this thesis. This section will communicate the basic architecture of Barrelfish, how the operating system handles memory managment using capabilities and briefly touch on the underlying file system. The multikernel is based on three principles (Baumann et al., 2009):

1. Make all inter-core communication explicit.
2. Make OS structure hardware-neutral.
3. View state as replicated instead of shared.

The inter-core communication using messages becomes more efficient compared to shared memory, the more cores are involved. Shared memory between multiple cores induce a huge overhead by having to locking data structures. By using explicit messages, a multikernel OS can use optimization algorithms known from networking, like pipelining or batching, to achieve better performance than shared memory could.

The benefit of making the operating system hardware-neutral is that the code base does not have to be extensively adjusted for each new hardware base. Especially the efficiency of

inter-process communication is extremely dependant on the hardware. By transferring this functionality to the software level, the speed of a multikernel is now much less dependant on eventually changing hardware.

Operating systems like Windows or Linux use shared data structures to keep state. Barrelfish on the other hand replicates the state as much as needed and then keeps it consistent using explicit messages. This reduced the overhead of shared memory synchronization and the latency by putting data is closer to the process that accesses it frequently.

By applying those three principles, a multikernel OS aims to improve performance on systems with many cores.

2.2.1 Multikernel Architecture

The Barrelfish architecture is designed based on the multikernel model shown in Figure 3. The multikernel model treats the operating system as a distibuted network of cores. The different cores have no shared memory; inter-core communication is handled using message passing.

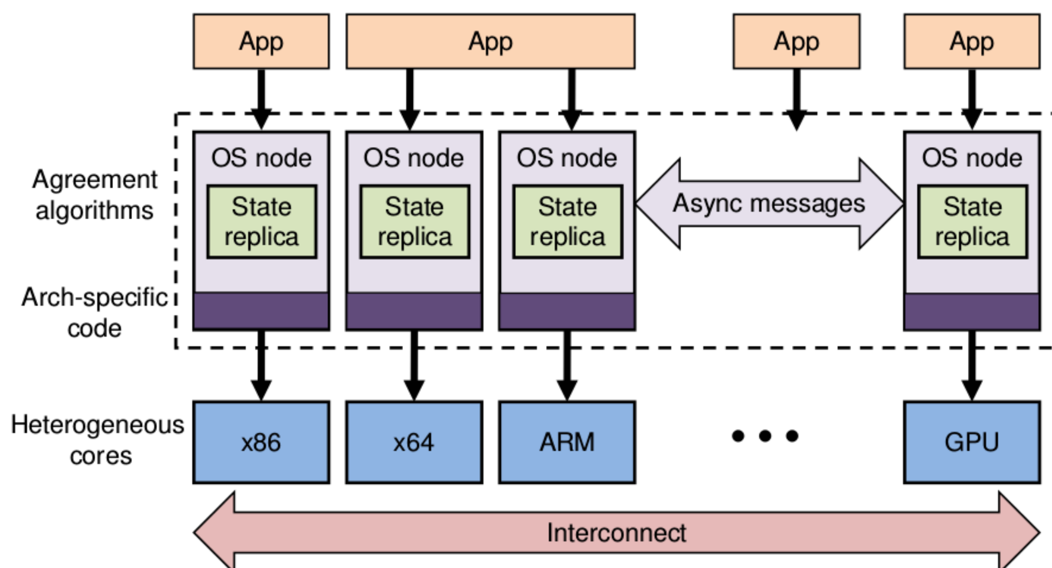

Figure 3: Multikernel architecture. Taken from [2]

2.2.2 Memory Management and Capabilities

Also in a multikernel OS there are resources that have to be kept consistent over the whole system, for example physical memory. The OS has to prevent that there exists multiple virtual mappings to the same physical region. For this purpose, Barrelfish introduces capabilities to keep its physical memory allocation across multiple cores consistent. A capability can be described as a user level reference to a region of physical memory. All the memory management is done through user level system calls that manipulate capabilities. Each user process has to handle its own virtual memory management. If a process wants to map virtual memory to a physical memory region it request capabilities. As a result, it is assigned some RAM capabilities. It then retypes these RAM capabilities to page table capabilities which allow it to create a new page table. Afterwards, to create a mapping the process retypes some RAM capabilities to mappable frame capabilities, which can be inserted in the newly created page table to establish a new physical-virtual memory mapping.

2.2.3 Filesystem

Barrelfish uses a virtual file system (VFS) API on top of a network file system (NFS). NFS is based on the client/server approach and let's the application also update non-local network files. The VFS on top just allows applications to access an underlying file system in a uniform way.

2.3 Drawbridge

Drawbridge is a library OS developed by the Stony Brook University in collaboration with Microsoft Research. It builds a prototype of a Windows 7 library OS that is able to run applications like Excel or PowerPoint. Drawbridge is the first libOS built from a commercial operating system. It aims to achieve security isolation and rapid system evolution with an order of magnitude smaller overhead than a virtual machine would have.

2.3.1 Design & Architecture

Drawbridge categorises the services of Windows 7 into three categories. It packs application services like frameworks and language runtime into the libOS and leaves hardware services like device drivers, and user services like the GUI to the host OS. It then implements an ABI using a platform adaptation layer and security monitor, so that the library OS can communicate with the host OS services. While a Hyper-V VM of Windows 7 uses around 512 MB of RAM and 5 GB of disk space, the API of drawbridge only needs 16 MB RAM and 64 MB of disk. The Drawbridge library OS is only about $1/50^{th}$ the size of Windows 7, but is able to provide execute a broad variety of applications.

3 Design & Implementation

This chapter will focus on design and implementation choices made to implement the functionality of the PAL on Barrelfish. The first section outlines the challenges encountered compiling the Graphene libOS together with a Linux binary for Barrelfish, while the second section focuses on the specific implementation of certain PAL ABI functions.

3.1 System Structure

In Figure 1 we showed how an application on Linux is executed using the PAL and different shared libraries. Compared to Linux Barrelfish does not yet support dynamic linking, thus cannot load shared libraries at runtime. The PAL contains a function that specifies how symbols of the binary are resolved at runtime to find the entry point of the host ABI. The idea behind a library OS is that it can load an application during runtime and that they therefore do not have to be statically compiled together. This imposes a big problem for Barrelfish.

In a first approach we decided to take the class responsible for symbol resolving from Linux and hoped that it would also resolve the symbols for Barrelfish, even without dynamic linking support. This approach did not yield the desired result and therefore we had to search for another solution. We decided to go with the approach shown in Figure 4.

We noticed that it will be nearly impossible to use the libOS without dynamic linking and decided to exclude it for the time being and to build a statically linked Barrelfish application consisting of the PAL, a slightly adapted Linux binary and libbarrelfish. In the Linux binary we ourself do the translation the library OS normally would do. So instead of using functions like malloc we use the already resolved function `DkVirtualMemoryAlloc()` our platform adaption layer does understand. With this approach we still have to modify the Linux binary, but by implementing the PAL functions for Barrelfish we build a foundation to facilitate running the whole Graphene libOS, once Barrelfish supports dynamic linking and we are still able to run test applications on the PAL. This allows us to draw conclusions about the performance of Graphene on Barrelfish even without a full library OS being in place.

The static linking of the modified Linux binary with the PAL and libbarelfish was not as trivial as it seemed at first glance. With dynamic linking Barrelfish could ask for the entry

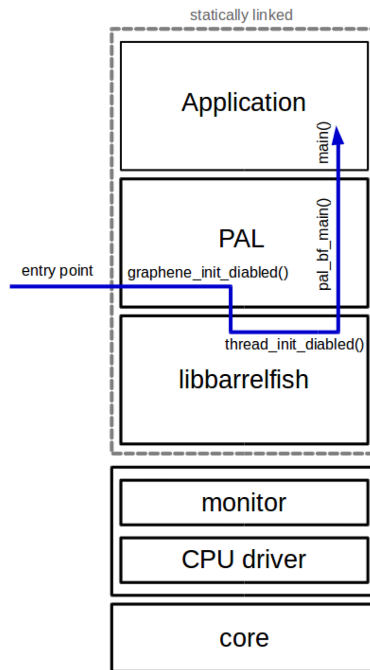

Figure 4: Statical Linking of Application with PAL and libbarelfish

point of PAL which then would ask the application for its entry point. With static linking we have to explicitly state the name of the entry function and cannot just ask the dynamic linker to resolve the entry point symbol. Barrelfish usually would spawn a main_thread in which it calls the main function of the application. But this is not what we want. Instead we want Barrelfish to run the PAL first and let the PAL then call `main()` of the binary.

We define the entry point of our statically linked application to be in the PAL. From there we call the function `graphene_init_disabled()` which changes the main_thread to be `pal_main_thread()` and then calls `thread_init_disabled()` located in libbarrelfish, to set up Barrelfish's threading. Now Barrelfish does not start its usual main thread, that calls the applications main function, but executes our custom `pal_main_thread()`. This thread calls `pal_bf_main()`, the main function of Barrelfish's PAL implementation, where some initialization, argument parsing and preparation for the application execution takes place. After all this setup we can finally call the `main()` function of the modified Linux binary, which will now execute over the PAL and not over Barrelfish directly.

3.2 Platform Abstraction Layer (PAL)

This section focuses on the specific implementation of certain PAL ABI functions, mainly focusing on the implementation of threading, memory management and signaling.

3.2.1 PAL_HANDLE

A PAL_HANDLE is a union of structs used to build different handle types needed by the PAL. The union consists of a header that specifies the type of the handle. Further, the union contains a struct for each handle type needed by the PAL, e.g. there exist structs for files, threads, events and more. These type structs each contain a header and some type specific fields. The thread struct for example additionally contains a thread ID and a pointer to the thread, while the file struct contains the absolute pathname of the file it refers to. When a function creates a PAL_HANDLE it has to specify its type and the fields in the struct of that type. If a function receives such a handle as an argument, it can find out its type and has then access to all the information specified in this type's struct. PAL_HANDLEs provide a simple way to pass a lot of content to a function using only one argument that contains all important information of a given type.

3.2.2 Threading

For the threading in Graphene five methods were implemented in a pretty straight forward manner using the existing threading support available in Barrelfish. We will now elaborate on our design choices.

```
1 int _DkThreadCreate (PAL_HANDLE * handle, int (*callback)
     (void *), const void * param, int flags)
```

`_DkThreadCreate()` takes a thread handle, where we will store the thread pointer and thread ID, a function that will be executed by the thread and the arguments with which the function will be called as arguments. Our implementation uses Barrelfish's `thread_create_varstack()` method. The method creates a Barrelfish thrad and returns a pointer to the new thread. This pointer together with the thread ID provided by Barrelfish are then stored in the thread handle and returned to the callee.

```
void _DkThreadExit (void)
int _DkThreadResume (PAL_HANDLE threadHandle)
void _DkThreadYieldExecution (void)
```

Thread exit is implemented by making `_DkThreadExit()` a wrapper which calls Barrelfish's `thread_exit()`. Also `_DkThreadYieldExecution()` and `_DkThreadResumeExecution()` simply call Barrelfish's `thread_yield()` and `thread_resume()` respectively.

```
int _DkThreadDelayExecution (unsigned long * duration)
```

To implement `_DkThreadDelayExecution()` I decided to use `barrelfish_usleep()` from libbarrelfish's deferred.c. Graphene expects you to specify the delay in microseconds and also Barrelfish's usleep expects the time in microseconds so the delay can be passed to the `barrelfish_usleep()` method without conversion.

3.2.3 File

To implement file handles in Graphene we decided to use the already existing virtual file system (vfs) from Barrelfish.

```
static int file_open (PAL_HANDLE * handle, const char *
    type, const char * uri, int access, int share, int
    create, int options)
```

`File_open()` takes the uri of the file to open and flags, that specify properties like whether it already exists or should be created as arguments, as well as an output filehandle. Barrelfish's `file_open()` expects the full absolute path of the file so we have to prepend the path to the filename. This is done by using the memmove command to shift the filename to the right and to prepend the path. Providing the absolute path of the file will let Barrelfish recognize it. Afterwards we check the create flag. If it is set, the implementation uses the `vfs_create()` method to create a `vfs_handle_t` with the absolute pathname. Else the file already exists and we can use `vfs_open` to open the file to the specified vfs handle. Finally we build our output handle. As it is a filehandle we set the vfs handle and the path of the opened file and return the handle. The implementation is pretty straight forward. By using file system functions of Barrelfish, one has to check some flags and after opening or creating the file build a new Graphene filehandle.

```
static int file_read (PAL_HANDLE handle, int offset, int
    cnt, void * buffer)
```

The read function takes as arguments a Graphene filehandle, that wraps the Barrelfish file system information of the file, an offset, from which we will start to read together with a buffer, where the file is stored and an integer that specifies how many chars should be read from the buffer. We will be using the `vfs_handle_t` wrapped by the Graphene handle. First we call `vfs_seek()` with the `vfs_handle_t` and the offset, to find the position in the handle. Then we can call `vfs_read` with our vfs handle. This will write the read data to the buffer we received as argument. The return value of the read function is the actual number of characters read. With this it is possible to verify if all what we wanted to read actually could be read.

```
static int file_close (PAL_HANDLE handle)
static int file_delete (PAL_HANDLE handle, int access)
```

The functions `file_close()` and `file_delete()` are wrappers for the Barrelfish functions `vfs_close()` and `vfs_delete()` respectively. In the delete method we also get a boolean variable, that tells us whether we have the right to access the file or not. If we do not have access we naturally will not try to delete the file, but report this back to the user instead.

```
static int file_map (PAL_HANDLE handle, void ** addr, int
    prot, int offset, int size)
```

The implementation of `file_map()` is also interesting. This function creates a mapping in the virtual space of the calling process. As arguments `file_map()` receives a Graphene filehandle, a starting address, an offset, where the mapping should be placed and the size of the mapping. The size we get does not have to be a multiple of the BASE_PAGE_SIZE, so we round it up to the next nearest multiple. Next off we are going to use Barrelfish's `vspace_map_file()` function and pass it all the above arguments and additionally a vregion and a memobj as output handles. Finally we use the `vregion_get_base_addr()` from Barrelfish to get the output address of file_map. The address points now to the beginning of the new mapping.

3.2.4 Memory

In this chapter we will describe how we implemented allocating and freeing memory for Graphene on Barrelfish.

```
int _DkVirtualMemoryAlloc (void ** paddr, int i_size, int
    alloc_type, int prot)
```

We start discussing the implementation of memory with the memory allocation. The function gets called with paddr, that specifies at what address the memory will be allocated. If it is NULL we can choose ourself where to allocate it. In case it is not NULL we can use the Barrelfish function `vspace_map_anon_fixed`. This function will create and map an anonymous memory region at a fixed address. If no address was specified, meaning we can allocate the memory at any free region of large enough size, we can use Barrelfish's `vspace_map_anon_attr()` method. This method will automatically create and map an anonymous memory object for us. Independent of whether you use the position dependent or independent version, as output arguments you will receive a memobj and a vregion. Note that currently the implementation differs from mmap with MAP_FIXED used in Linux, in that it will not overwrite pre-existing overlapping parts of the requested mapping when a fixed address is given.

Barrelfish expects a size of power of two for the requested region. The size specified in the function call does not have to fulfill this limitation though. So we round up the size to the next power of two before allocation the vspace. As Graphene expects a memory capability of the requested and not rounded up size, we use a recursion to fill a memobj with the a number of frames specified by the size argument. The recursion adds the highest power of two below the size to the memobj and then adjusts the size accordingly for the next iteration. This is shown in the code snippet below

```
do {
        size_t alloc_bytes = 1ULL << log2floor(size);
        err = frame_alloc(&frame, alloc_bytes, &retbytes
            );
        // add frame to memobj at correct offset
        err = memobj->f.fill(memobj, offset, frame,
            alloc_bytes);
        // trigger page fault(s) for this frame
        err = memobj->f.pagefault(memobj, vregion,
            offset, 0);
        offset += alloc_bytes;
        size   -= alloc_bytes;
    } while (size > 0);
```

Code 3.1: Recursion for frame allocation with error handling omitted for brevity

```c
int _DkVirtualMemoryFree (void * addr, int size)
```

To free memory the function `_DkVirtualMemoryFree()` was implemented. The function takes an address, where the memory is allocated and the size of the region that should be freed as arguments. It then will try to find the vregion that contains addr and just unmap that region. The function at the moment does not support partially unmapping a vregion that was created by `_DkVirtualMemoryAlloc()`, but instead prints a warning in these special cases.

```c
bool _DkCheckMemoryMappable (const void * addr, int size)
```

The `_DkCheckMemoryMappable()` function checks, whether at a given address a memory region of the specified size exists. If it exists, the function returns true and false otherwise.

```c
int _DkVirtualMemoryProtect (void * addr, int size, int
    prot)
```

This function receives as input arguments an address and the size of a memory region and an integer, that specifies what protection this region has to have. Prot is a bit field that specifies access types, bit 0 determines read access, bit one write access and bit 2 whether one has execute right.

```c
int _DestroyFailedAlloc(struct vregion* region, struct
    memobj *memobj)
```

If an error is encountered while allocating memory, the function `_DestroyFailedAlloc()` can be used to clean up the incomplete memory allocation. The method is a wrapper for Barrelfish's `vregion_destroy()` and `memobj_destroy_anon()` methods.

3.2.5 Signals

In Graphene, there is a purely software based implementation of signals. A thread can call `_DkObjectsWaitAny()`. This method takes an array of event handles and the size of the array as arguments. Additionally another handle is provided for output. After calling this function the thread yields and waits for one of the events in the array to be triggered. Multiple threads can wait on the same event.

```
1 int _DkEventCreate (PAL_HANDLE * event, bool initialState,
      bool isnotification)
```

A notification event can be created by using the function `DkNotificationEventCreate(0)`. A zero argument means that in the initial state the event is not signaled, while with a 1 as argument it is already signaled. This function will then call the Barrelfish specific implementation of `_DkEventCreate()`. The function creates an event by creating an event handle and specifying it' properties.

```
1 int _DkEventSet (PAL_HANDLE event)
```

An event can be triggered by using the `_DkEventSet()` method. This method takes an event as argument. If the event is a signaling event, the function wakes up the first thread waiting on this event. If the event is a notification event, the function wakes up all threads waiting on this particular event using a broadcast.
In the Linux implementation a futex syscall is used. From the manual [7]:

> "The `futex()` system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address (while the addresses for the same memory in separate processes may not be equal, the kernel maps them internally so the same memory mapped in different locations will correspond for `futex()` calls). This system call is typically used to implement the contended case of a lock in shared memory, as described in futex(7)."

Because a futex call does not exists for Barrelfish, another approach had to be chosen.

The initial idea for the implementation of `_DkObjectsWaitAny()` was to use condition variables. If the method was called it would start up a new helper thread. This helper thread then spawned a new thread for each one of the events in the array handed to `_DkObjectsWaitAny()`. It then would wait on its own condition variable to be signaled. The threads spawned for the individual events would also first wait on a condition variable. This variable would be set by the `EventSet()` method, when an event was triggered. The thread then would signal the condition variable of the helper thread to resume, who would now cancel all the other threads waiting for an event and return, which would result in the calling thread resuming. We noticed that this approach did not work as planned. Barrelfish only allows cancelling of the running thread itself and at the moment has no implementation to cancel other threads.
So we had to look for an alternative implementation and decided to use the waitsets of Barrelfish. On a waitset multiple channels can be registered. When a waitset is notified that on of the channels registred on it received a message, it replies with a previously specified closure function that has to be called next.
So the new implementation of `_DkObjectsWaitAny()` looks as follows. Recall that the method gets called with an array of events, from which one has to be triggered to

continue. First off, we check if one of the events already has been triggered, before wait was called. If so we immediately return that event and continue with the program execution. With this we make sure to not wait forever, if the `EvenSet()` method gets called before `_ObjectsWaitAny()`.

```
int _DkObjectsWaitAny (int count, PAL_HANDLE * handleArray
    , int timeout,  PAL_HANDLE * polled)
{
  //check if one event was already signaled
  for(int j = 0; j < count; j++) {
    if (atomic_cmpxchg(&handleArray[j]->event.signaled, 1,
        1) == 1) {
      *polled = handleArray[j];
      return 0;
    }
  }

  //if we have a timeout and add a time event
  if(timeout != NO_TIMEOUT){
      count++;
  }

  //Initialize data structures
  struct ws;
  waitset_init(&ws);
  bool triggered[count];
  memset(triggered, 0, sizeof(bool)*(count));
  struct deferred_event timeout_ev;


  if (timeout != NO_TIMEOUT) {
    deferred_event_init(&timeout_ev);
    int err = deferred_event_register(&timeout_ev, &ws,
        timeout, MKCLOSURE(event_triggered, &triggered[count
        ]));
    count--;
  }

  // add our waitset and callback function to this event
  for (int i = 0; i < count; i++) {
    _DkEventRegister(handleArray[i], &ws, MKCLOSURE(
        event_triggered, &triggered[i]));
```

```
33  }
34
35  if(timeout != NO_TIMEOUT) {
36      count++;
37  }
38
39  //we call event_dispatch until an event has been
      triggered
40  while (!any_triggered(triggered, count)) {
41      event_dispatch(&ws);
42  }
43  int winner = get_triggered(triggered, count);
44
45  if (timeout != NO_TIMEOUT) {
46    count--;
47  }
48
49  for (int i = 0; i < count; i++) {
50    _DkEventRemoveWaitset(handleArray[i], &ws);
51  }
52
53  //return the first signaled event
54  if (winner >= count || winner < 0)  {
55    if(timeout != NO_TIMEOUT) {
56      deferred_event_cancel(&timeout_ev);
57    }
58    *polled = NULL;
59    waitset_destroy(&ws);
60    return -1;
61  } else {
62    *polled = handleArray[winner];
63    waitset_destroy(&ws);
64    return 0;
65  }
66 }
```

Code 3.2: Implementation of signals with error handling omitted for brevity

If this is not the case, we register our waitset on all the events we received as arguments. As `DkObjectsWaitAny()` can be called from multiple threads on the same argument, each event internally has a doubly linked list which stores all the waitsets the event is registered on. The exact use of this functionality will be explained later.
Once we set this all up we continuously call the method `event_dispatch()` as show on lines 40 - 42 in Code 3.2. `Event_dispatch()` cycles through all the events

registered on our waitset and checks if one has been signaled.

Recall, that a waitset notifies you what function to call, once an event registred on it receives a message. In our case, the function `eventtriggered()` gets called. This function sets the entry corresponding to the event received by the waitset to true in the boolean array `triggered`.
The function `any_triggered()`, continuously called in the loop condition checks, whether one of the entries in `triggered` changed from false to true. If so, it returns true and thus we exit the loop.
Now let us have a look at the `EventSet()` method: The method checks, whether the event is a signal or a notification. If it is a signal then it signals the first waitset on which the event is registered and returns. If it is a notification, then it signals all the waitsets it is registered on to continue.
So if the event has not already been signaled it sets the signaled flag to true and then steps through the list of waitsets the event is registred on. All those waitsets get notified that the event has happened by using the Barrelfish method `waitset_chan_trigger_closure()`.
Lets return to the method `_DkObjectsWaitAny()`. An event has been set and we can exit the loop on lines 40 - 42 in code 3.2. Now we call the method `get_triggered`, which steps through the boolean array `triggered` and finds out what event has been triggered.
We remove all waitsets that waited on the triggered event and can now return the event that has been signaled.

This implementation was chosen because it is possible, to call `_DkObjectsWaitAny()` with the same Event multiple times. So more than one thread can wait for a specific event to happen. The wait method also notices, whether the `EventSet()` function has been called before the wait function and thus will not wait forever in case this happens. One drawback is that it takes some time to signal all waitsets an event is registered on. If multiple `_DkObjectsWaitAny()` wait for the same event, a second event that is called after the first one can overtake it if it only has to notify one waitset. This can be similar to a race condition and `_DkObjectsWaitAny()` could return the wrong event as the winner. But as this methods purpose is being a barrier that halts execution until at least one event has been signaled but does not care which event, that this drawback seems acceptable.

4 Benchmark

We benchmarked the PAL's event signaling on Barrelfish and Linux. The tests were run on "babybel1", a 20 core machine from the Systems Group of ETH Zurich. In Table 2 the exact specifications of the machine are shown.

CPU	2x10 Intel(R) Xeon(R) E5-2670 v2
Frequency	2.50 GHz
L1 cach	640KiB
L2 cache	2560KiB
L3 cache	25MiB
RAM	16GiB DIMM DDR3 @1867 MHz

Table 2: Hardware specification of "babybel1", used for benchmarking

The two test environments we will be using are the current version of Barrelfish (date: 30.08 2015) and Ubuntu 14.04.

4.1 Implementation

To compare the performance of event signaling on Barrelfish and on Linux we have written a test that uses the PAL's functions. The PAL then uses the host OS specific functions to execute the test. In a first attempt we let the main thread of the test create two helper threads, that both signal an event each. The main thread then calls `DkObjectsWaitAny()` on the two events to wait for one to be signaled. We measure the time it takes from the beginning of the main thread's execution until one of the two events is signaled. We made sure that there is no overhead, due to a something like a printf call between the start and end of our measurement.
After some test runs and consultation with the developers of Graphene we noticed that Graphene cannot yet wait on multiple events simultaneously on Linux, because the implementation uses futexes, which cannot be asynchronously waited on. Note that the Barrelfish implementation of event signaling is very well capable of waiting on multiple events on the same time. But to compare the implementations on the different hosts we decided to modified our test so that the main thread only creates one helper thread, which will then signal an event.

4.2 Results

Before the helper thread signals the event it waits for a predefined amount of time. On both systems we measure the time until completion, if the helper thread waits for 3'000, 30'000 and 100'000 microseconds respectively before signaling the event. The tests were run multiple thousand times to get meaningful results.
We expect that our tests will run faster on Linux, as it uses the optimized futex system call for its event signaling. The Barrelfish implementation on the other hand uses a presumably slower software based approach.

Run #	Time in μs			
	Linux		Barrelfish	
	average	std. deviation	average	std. deviation.
Run 1	3'253	19	80'014	33
Run 2	30'219	14	80'015	39
Run 3	100'209	10	160'020	84

Table 3: During the first run the helper thread waits 3'000 μs, during the second run 30'000 μs and during the third run 100'000 μs.

We can see in Table 3 that our prediction is true and the signaling on Linux is a lot faster. What is surprising is that Barrelfish takes around 80 ms to finish a run, independent of whether the helper thread waited 3'000 or 30'000 μs before signaling the event. In the third run, when the thread waits for 100'000 μs Barrelfish takes over 160 ms, so around twice as much time as in the previous runs. Linux on the other hand only takes around 200 microseconds longer than the time specified for the helper thread to wait. So in the first run it is 25 times faster than Barrelfish. What took us aback was that runs 1 and 2 take the same time on Barrelfish. This does not seem right. After some investigation we turned our attention to the Barrelfish scheduler. Barrelfish uses rate-based earliest deadline first scheduling [8]. The default timeslice length specified for the scheduler is 80 ms. Now it is obvious why the first two runs both take around 80 ms to complete. They need one timeslice and thus around 80 ms. Naturally it follows why run 3 takes 160 ms to complete. The helper thread there waits 100 ms before signaling the event and thus one timeslice is not enough to complete the task. So this run needs two timeslices and therefore cannot finish in less than 160ms.

We were not really satisfied with this result and decided to change the default timeslice of the Barrelfish scheduler from 80 to 1 ms. The results are shown in Table 4.

Run #	Time in μs			
	Linux		Barrelfish	
	average	std. deviation	average	std. deviation.
Run 1	3’253	19	3’367	506
Run 2	3’0219	14	30’373	493
Run 3	100’209	10	100’432	502

Table 4: During the first run the helper thread waits 3’000 μs, during the second run 30’000 μs and during the third run 100’000 μs. The timeslice length of the scheduler was changed to 1 ms.

As we can see the results changed quite a bi. On average the tests on Barrelfish take around 300 μs longer than the time the helper thread waits before signaling. The standard deviation on the other hand went up to around 500μs, which is half a timeslice. This happens because around half the runs need one more than the minimal amount of timeslices to finish the job. Still, with the changes to the scheduler we see that the Barrelfish implementation of event signaling only takes around 100μs longer than the one from Linux. Considering that our implementation also can wait on multiple events simultaneously we can be quite satisfied with the results we got after changing the length of the scheduler’s timeslice.

5 Conclusion

The goal of this thesis was to implement the Platform Adaptation Layer of the Graphene library OS on Barrelfish. I show how to run applications on the PAL for the Barrelfish OS by introducing an approach that uses static linking together with an altered execution control flow. Directly running unmodified native Linux binaries and receiving the full benefits of a libOS is not yet possible on Barrelfish due to missing support of dynamic linking.

Nevertheless I provide implementations of the most important PAL functions, sticking to the Barrelfish design choices wherever possible and introducing new approaches where needed. I discuss the similarities and differences of the host OS specific PAL implementation comparing Linux and Barrelfish.

The main result of the thesis is a performance analysis on the basis of event signaling. I argue why the benchmarks run up to an order-of-magnitude slower on Barrelfish compared to Linux when using the default Barrelfish scheduler and show, that with a slight modification to the scheduler's timeslice length the benchmarks finish in approximately the same time on both operating systems.

5.1 Future Work

While I have implemented the most important functions of the PAL there still remains some interesting work to be done. The current version of the PAL does not yet support semaphores, byte streams and sockets. In the future we could extend the PAL in a way that allows applications to also use these functions.

Once Barrelfish supports dynamic linking, an interesting next step would be to get the Graphene library OS to run on top of the PAL. This would allow Barrelfish to benefit from the virtualization and platform compatibility a libOS provides.

With a complete implementation of Graphene on Barrelfish more interesting experiments and performance measurements could be conducted. It might be interesting to analyse how Barrelfish performs running an application over the whole libOS compared to running over the PAL. Furthermore these results could be compared to the ones of Graphene running on a Linux OS. Additionally, a full implementation would allow Barrelfish to execute native Linux binaries, providing it a lot of new programs.

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 LibraryOS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Bieri	Yves

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

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

Place, date	**Signature(s)**
Zurich, 02. Sept. 2015	Y. Bieri

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

Bibliography

[1] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[2] 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, pages 29–44, New York, NY, USA, 2009. ACM.

[3] 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, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[4] 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, pages 339–354, Berkeley, CA, USA, 2008. USENIX Association.

[5] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News*, 39(1):291–304, March 2011.

[6] Graphene. `http://graphene.cs.stonybrook.edu/`, 2014. [Online; accessed 18-August-2015].

[7] *futex(2) Linux User's Manual*, September 2015.

[8] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.