Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Bachelor's Thesis Nr. 137b

Systems Group, Department of Computer Science, ETH Zurich

Dynamic Linking and Loading in Barrelfish

by

David Keller

Supervised by

Prof. Timothy Roscoe
Reto Achermann
Simon Gerber

March 2015 – August 2015

Informatik
Computer Science

# Abstract

Dynamic linking and loading capabilities are an integral part of most modern operating systems. They make the system more modular, enable independent updates of system resources and client applications and save space.

This work describes the design and implementation of dynamic linking and loading in Barrelfish – a multikernel research operating system – using its native APIs and build toolchain.

The evaluation shows that space can be saved for executables compared to a static linked binary if more than 37 applications share the same set of libraries. A threshold easily met if mandatory system libraries are shared.

# Acknowledgments

I want to thank Prof. Timothy Roscoe for letting me write this bachelor thesis under his supervision. I learned a lot during this time. I also want to thank the assigned assistants Simon Gerber and Reto Achermann for their support. They always had time to answer my questions during and outside weekly meetings and they provided me valuable feedback on earlier versions of this work. Furthermore I want to thank David Cock for his feedback on the draft and the discussion about the design.

Additionally, I want to thank Zeno Koller, Martina Steinhauer and Mathias Daube for corrections on the drafts and Fabienne Christen for her unwavering mental support while creating this work.

Last but not least I want to thank my family for supporting me throughout all the years leading up to this work.

# Contents

# Chapter 1

# Introduction

Today, most operating systems, such as Linux, OS X, FreeBSD and Windows, provide a way to use so-called shared libraries (also dynamic libraries or DLLs). These libraries contain functionality which is reused by many different applications at runtime. Shared libraries get linked into the executable implicitly at program startup – dynamic linking – or they can be loaded explicitly inside the application – dynamic loading [37, 38]. This way of linking libraries brings many benefits over statically linked libraries besides the reuse of code achieved with both ways of linking.

The three main advantages of shared libraries compared to statically linked libraries are the following:

1. The shared library has to be stored just once on a machine and is not part of every program binary that links to it. This saves disk space since the executable files are smaller in size.

2. Bugfixes and changes in shared libraries just need a recompilation and redistribution of the updated library. Every application that links to it gets the benefit of the update directly without any action needed by the author of the dependent application.

3. With virtual memory, a shared library only needs to be loaded once into physical memory and the read only parts then can be shared by all applications using this library. This saves memory. Sharing of writable sections is possible, but normally not intended.

There are also drawbacks of using shared libraries. The main disadvantages are described by the umbrella term "Dependency hell" [35]. First, one program can depend on many shared libraries. It might be necessary to install them in a specific order. If there is no such thing as a packet manager, this needs to be done manually and can be tedious. Second, if there exists a long chain of dependencies, program startup can take significantly longer than for a statically linked program. Third, there might exist conflicting dependencies. Meaning, one application depends on a certain version of a shared library and another depends on the same library, but on a newer or older version with changed application binary interface (ABI) or application programming interface (API). The system might have a versioning solution to cope with this challenge. But then one loses the benefit of having just one library installed and needs to store and load different versions of the library. The benefit of security updates to shared libraries does not apply automatically to all clients. Finally, one needs to load complete shared libraries into memory, even though the application calls just one function of it. When using static libraries however, only the needed object files get linked into the executable. This might be amortized if multiple applications access the same library and its read only sections are shared in physical memory.

Barrelfish, a multikernel research operating system, does not support dynamic linking and loading capabilities and shared libraries at the time. This work gives Barrelfish the capability of distributing and updating system libraries with the operating system and building and distributing applications independently. It offers all the advantages (and disadvantages) described earlier in this chapter.

This work starts with background about dynamic linking and loading by showing how this capability is implemented in different operating systems. This background chapter also provides information about standards in dynamic loading and describes a binary file format that is used by many open source operating systems for executables and shared libraries. Finally there is an introduction to Barrelfish, the research operating systems extended with this work. The design chapter describes what the dynamic linking and loading in Barrelfish should look like and the implementation chapters shows how it is implemented. The implementation is evaluated by comparing the size of binary files and the speed of specific functions. In the chapter about future work missing features in the implementation and possible extensions are discussed.

# Chapter 2

# Background

The implementation of dynamic linking and loading in modern operating systems differs in many ways between operating systems. This chapter starts with the general infrastructure every dynamic linker and loader needs and provides. Then an overview of binary formats used by the discussed operating systems is given, followed by the description of the Portable Operating System Interface (POSIX) a standard for dynamic loading. Afterwards the implementations in Linux, FreeBSD and Windows are explored. These three operating system provide a broad view of the landscape of todays operating systems. They all have different histories and depending on that have different implementations of dynamic linking and loading capabilities. At the end of this chapter the base of this work – Barrelfish and its static linking capabilities – is described.

## 2.1 Infrastructure

A dynamic linking and loading infrastructure needs a format for shared libraries. All the operating systems discussed here, use the format they use for executables also for shared libraries with additional data that is provided to make loading and linking at runtime possible. This makes it easier for operating system developers, because they need to write only one runtime loader for executables and shared libraries [40]. One main difference compared to an executable is the fact that shared libraries need to be aware of the fact that they might not be always loaded at the same virtual address. Some formats solve this problem with position independent code, others rewrite the absolute addresses if the intended addresses are not available. Fixed unchangeable addresses for every library would be hard to maintain. In such an environment a – most likely central – authority has to define at which particular address a library is loaded and the authority is well advised to leave some spare space after the library for future versions. Otherwise every update that increases the size of a library has to be assigned another address leading to fragmentation of the address space and the number of libraries that can exist is limited to the number of slots available in the address space.

The operating systems provide a program linker to create shared libraries and to link them against executables. This ensures that the runtime linker is aware which shared libraries to load at program startup. The runtime linker is the key component for shared libraries. It runs at program startup, searches for the required shared libraries, does relocations and symbol resolution and thus enables that every part of the program can call the functions and access the data it needs. In more detail the runtime linker needs to do the following steps in this specific order whenever it loads a new shared object:

1. Load the defined shared object and collect its dependencies.
2. Load all the dependent shared libraries recursively.
3. Create a breath-first dependency graph for symbol resolution.
4. Relocate all loaded shared objects.

Step 3 might be done together with 2, because the load order normally reflects a breadth-first order of the dependencies.

Since shared libraries are often intended to be used by different programs and to be updated independent of them, the operating systems have global directories where the runtime linker searches for depending libraries. Additionally the environment and shared library can provide more search paths. The operating system also handles sharing of read only parts in physical memory via virtual memory.

## 2.2 Binary formats

### 2.2.1 ELF

Linux and FreeBSD as well as Barrelfish use the Executable and Linkable Format (ELF) for their executables and shared libraries. Thus, we start with an overview of ELF and how its specification defines shared libraries independent from the implementations in different operating systems.

ELF is a container format for a lot of different binary formats. Figure 2.1 show how an ELF file is structured. It offers two views of its contents.

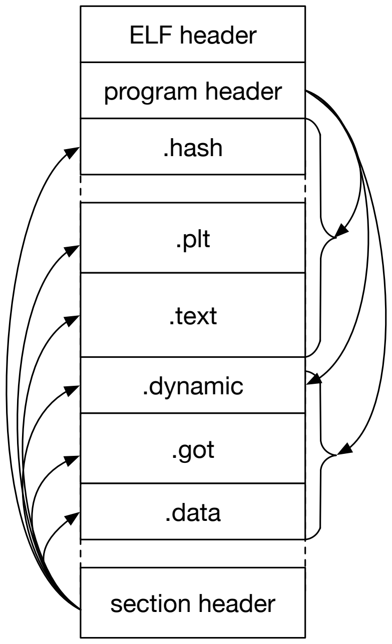

Figure 2.1: Structure of an ELF file

One view is based on sections defined by the section header. They are used to group code (`.text`), global variables (`.data`, . . . ) and more together and they are used by the program linker to merge these sections when combining multiple object files. Segments, on the other hand, are defined in the program header table and used by the dynamic linker and loader. It points to segments containing multiple section that need to be loaded into memory and also shows where the dynamic section is. The dynamic section contains all information the dynamic linker needs for running. For all possible entries see [17].

There exist two essential tables in every dynamically linked executable and shared library to access global data and functions (i.e. every shared object has its own set of tables). The tables help to limit the number of runtime relocations needed and the places where they have to be done. More precisely, relocations are only necessary in the global offset table (GOT). This keeps the `.text` section and procedure linkage table (PLT) unmodified (i.e. relocation free) and these sections can be shared between different processes via virtual memory. The assembly code accesses these tables relative to the current instruction position, thus they are not dependent on where in memory they get loaded. Only the offset to the tables is required to stay the same.

## Global Offset Table (GOT)

The global offset table (GOT) enables relative addressing by storing absolute addresses to globally accessible data and functions. All entries are relocated at runtime before anything is called in the shared object unless they are function calls, which might be relocated lazily (see PLT). Global variables are accessed by loading the address of the variable from the GOT entry and then accessing the data at this address.

The first three entries of the GOT have special purposes on the x86-64 architecture. The first entry (i.e. `GOT[0]`) holds the address of the dynamic section. The second and third entry are initialized by the runtime linker to a value that is needed for the call back into the runtime linker and the entry address of the runtime linker, respectively [34]. They are used for lazy relocated function calls via PLT.

## Procedure Linkage Table (PLT)

The procedure linkage table (PLT) has an entry for every global function with a special entry at the beginning of the table. All position independent global function calls are done via the PLT. Because most of the applications have many more function definitions than global data and most of the functions never get called, the relocations for function calls are done lazily by default. This speeds up startup time if many shared libraries are loaded. Every PLT entry has a corresponding GOT entry for relocation, thus the PLT can be mapped into a read only and shared region in memory.

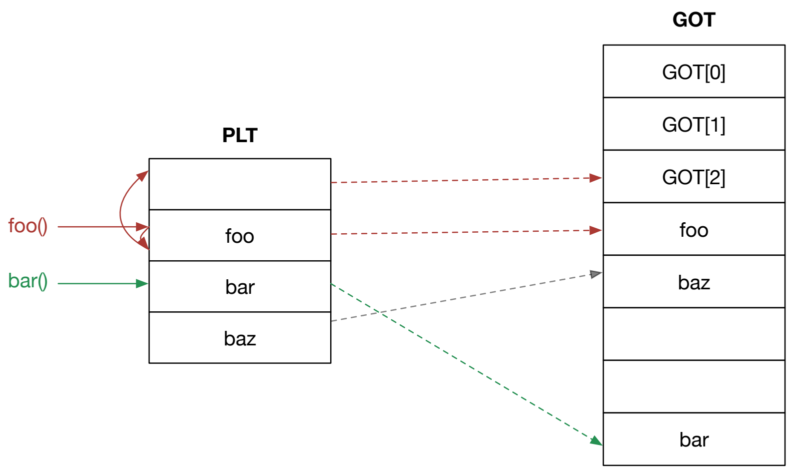

Figure 2.2: How function calls via PLT work. Red: first call. Green: subsequent calls.

Every PLT entry starts with an indirect jump to the address in the corresponding GOT entry (`foo()` and `bar()` in Figure 2.2). If the PLT relocations were done at loading time or the function was already called before, the GOT entry points to the address of the function and it gets called with just one additional jump, compared to a direct function call (green arrows in Figure 2.2). Otherwise the GOT entry just points to the address after the first instruction in the PLT entry. This instruction pushes the offset of the relocation entry to the stack and jumps to the first entry in the PLT.

The first PLT entry pushes the second GOT entry to the stack and moves the control flow to the third GOT entries' address which is initialized to the entry point of the runtime linker (red arrows in Figure 2.2). The runtime linker then relocates the GOT entry for this PLT entry and afterwards calls the function. Later function calls jump directly to the function, because the GOT entry points now to the function.

### 2.2.2 PE

The Portable Executable (PE) format is used in Windows and based on the Unix Common Object File Format (COFF). Like ELF it contains different load sections each having its own permissions. These are referenced by headers in the PE file. The import address table (IAT) is the equivalent to the PLT, but the function addresses are written directly into this table and they can be referenced by name or ordinal (a number). The main difference to the ELF format are relocations. PE has no position independent code. The shared libraries (DLLs) are compiled for a preferred base address. If at runtime this address is already used, a rebase is needed. This rewrites all absolute addresses, which is a huge penalty. If there on the other hand is no rebase needed, it runs significant faster than a position independent ELF file [41].

## 2.3 POSIX - a standard

To make source code more portable between different platforms, a joint group from IEEE and The Open Group defined standard terms, concepts and interfaces, the Portable Operating System Interface (POSIX) standard. The current version is *POSIX.1-2008 2013 Edition*[1] [23] and is implemented in big parts by most of todays operating systems. For dynamic loading, they define an interface and parts of the implementation [20].

[1]Also known as *POSIX.1-2013* [25].

### 2.3.1 Interface

The interface is defined by the four functions shown in Listing 1 as declared in the header file `dlfcn.h` [20].

```c
void *dlopen(const char *file, int mode);
void *dlsym(void *restrict handle, const char *restrict name);
int dlclose(void *handle);
char *dlerror(void);
```

Listing 1: Dynamic linking functions defined by POSIX

This interface can be used to load a shared library into the running program, call a function or access data and unloading the library. Listing 2 show an example using the shared library `libfoo.so` and accesses function `bar()` (error handling is omitted for better readability).

```c
void *handle;
handle = dlopen("libfoo.so", RTLD_LAZY);
int (*bar)(void);
*(void **) (&bar) = dlsym(handle, "bar");
bar();
dlclose(handle);
```

Listing 2: Example using the POSIX interface

#### `dlopen()`

`dlopen()` takes as arguments a filename or filepath of a shared library available to be loaded into the address space of the process and mode flags defining specific behaviors for symbol availability and relocation. On success, the function returns a pointer to a handler that is used in subsequent calls to `dlsym()` and `dlclose()`, otherwise a `NULL` pointer is returned and an error message can be accessed through `dlerror()`.

In a special case when `file` is the `NULL` pointer, `dlopen()` returns a handle for the global symbol table, which contains all executable and shared libraries loaded at program startup and through `dlopen()` in global mode. Otherwise just the shared library is searched. The flags passed through mode define how relocation should happen and optionally if the loaded symbols are globally available or not. On subsequent `dlopen()` invocations for the same shared library the same handler is returned, but `mode` is interpreted every time [21].

`dlsym()`

`dlsym()` takes a handle obtained through `dlopen()` and searches for the symbol named `name`. It returns a pointer to the function or data referenced by the passed symbol name. This pointer can be casted to the type of the function or data that is accessed. If the symbol was not found or the handler was not valid, it returns `NULL` and an error message through `dlerror()`.

The constants `RTLD_DEFAULT` and `RTLD_NEXT` are reserved by the standard, but are not standardized. A conversion from a `void *` pointer to a function pointer is not defined in the ISO C standard, but to conform to this standard, this conversation has to work correctly [22].

`dlclose()`

`dlclose()` informs the system that the `handle` is no longer used. Unloading of the shared library by the system is not required, but may be done, except a relocation from another place points to it. This can happen if the shared library was opened in global mode. Relocations are defined such that once carried out, they should never change. The dependency of a relocation is only removed after the unloading of the referencing object. If an error occurs a message is available through `dlerror()` [18].

`dlerror()`

`dlerror()` returns the most recent error in any `dlfcn` as null-terminated string. If no error occurred at all or since the last call to `dlerror()`, `NULL` is returned [19].

### 2.3.2 Symbol resolution

POSIX defines two ways of symbol resolution in the `dlopen()` standard [21].

#### Load order

Load order resolves the symbols in the order of loading. The order starts at the executable followed by its dependencies loaded at program startups and continues with shared libraries loaded later on (e.g. through `dlopen()`). Symbol resolution in the relocation phase and symbol lookup in the global symbol space is done using load ordering.

### Dependency order

Dependency order starts at the given executable and then uses a breadth-first order of all its dependent shared libraries, their dependencies and so on. `dlsym()` uses dependency order for symbol lookup, expect that it searches the global symbol space (obtained by passing `NULL` as `file` to `dlopen()`).

## 2.4 Implementations

### 2.4.1 FreeBSD

Dynamic loading and linking in FreeBSD is implemented by the run-time link-editor (rtld). rtld itself is implemented as a shared library. It gets loaded by the kernel together with any dynamically linked program. The control is then transferred to the rtld which loads all dependent shared libraries and resolves all dynamic symbols. Afterwards, it transfers control to the program itself [3].

#### Data structures

To manage loaded ELF files, rtld uses a struct called `Obj_Entry`, where all information about the executable or shared library are stored and can be queried. It contains information about file system references, mappings, dependencies, pointers to relocations and various booleans that indicate if certain steps are already done or something is available in this object file. This is the main data structure that gets passed around for different actions and in function calls [4].

For dependencies, load order and other related data rtld uses linked lists implemented as structs, containing a pointer to the current object and the next element in the linked list [4].

#### Relocations

There are different relocations that either need to be performed immediately or can be deferred. Mandatory relocations are those in the `.text` section (which normally do not exist) and in the GOT related to global data. On x86 architectures, every PLT entry has a corresponding GOT entry. These GOT entries do not need to be relocated at load time but changed to reflect the offset in memory the new shared object is loaded into. They nevertheless need a sort of relocation, but no symbol lookup is needed, which makes the relocation fast. To make PLT relocations at program runtime possible, the second and third special entry in the GOT need to be set at this time, too. For x86-64 on FreeBSD (called AMD64 in FreeBSD), `GOT[1]` contains a pointer to the `Obj_Entry` of the shared object itself and `GOT[2]` is a pointer to `_rtld_bind_start` — the entry function into rtld. If it is indicated that the PLT relocations should happen at loading and not deferred (e.g. through the runtime variable `LD_BIND_NOW`), this is done as a final step of relocation. Because relocations are architecture specific, their interface is declared in the rtld header file (`rtld.h`) and they are implemented in processor specific relocation files (`ARCH/reloc.c`) that get linked during program linking depending on the target archtitecture.

**Lazy relocations** If not disabled via environment variables, flags or parameters, FreeBSD does lazy relocation of function calls. Prior to the first function call, rtld is invoked. This happens through an architecture specific label `_rtld_bind_start` that saves, for x86-64, the current registers and then calls `_rtld_bind`. This generic function looks up the symbol, relocates the appropriate GOT entry and returns the function address. `_rtld_bind_start` then restores the registers and stack and calls the function. On the next invocation of the same function, it gets called directly through the PLT entry.

### Symbol resolution

Symbol resolution is implemented by a bunch of functions starting with `symlook_`. Their first argument is a pointer to a struct called `SymLook` containing information about the symbol's name, it's hashes and the defining symbol and object, if those exist.

Most of the queries into the symbol tables start at `symlook_default()`. If the library was symbolically linked[2], it starts searching inside the shared library itself. Otherwise, it directly calls into `symlook_global()`. If the symbol does not exist in the global symbol space or is just defined as weak, all dependency graphs (opened with `dlopen()`) this shared object is contained in are searched. Finally, if there was no successful lookup yet, `symlook_default()` searches the rtld itself. `dlopen()` is defined there, for example.

`symlook_global()` searches in two places. First, the list of shared objects loaded at startup. If nothing or just weak definitions were found, all dependency graphs of shared objects opened with `dlopen()` with global availability are searched. This final result is returned to the caller.

The public API for symbol lookup is `find_symdef()`. It takes the symbol number (`symnum`) and the referencing object (`refobj`) and returns the defining

[2]Meaning, the visibility of a global symbol is just within its shared library. It is enabled with `-Bsymbolic`. [16]

symbol and its object entry in `defobj_out`, if the symbol was found (`NULL` otherwise). This interface is used for symbol resolution in relocations.

Before doing a complete symbol lookup, `find_symdef()` tests if the symbol is defined locally (`STB_LOCAL`), meaning it defines itself. Otherwise, a symbol lookup through `symlook_default()` starts. To make lookup faster, already found symbols are stored in a cache.

Load order lookups after the POSIX standard can be done using `find_symdef()`. Dependency order is achieved through the function `symlook_list()` by providing the list of all dependent objects in dependency order. It goes over a list and looks for the symbol in every member's object.

### Mapping of new file

rtld has a single function to map a new ELF file into memory and create its corresponding `Obj_Entry`.

To get information about the ELF file in general and about the segments one needs the ELF header and the program header. They are at the beginning of the ELF file. FreeBSD optimizes for space by exploiting this fact and that the headers normally are smaller than one page. Thus rtld just maps this part into memory.

After digesting the program header, the mapping function creates a contiguous region in the virtual memory, maps all load segments into memory and sets the rights for every segment. After a successful mapping, an `Obj_Entry` is created, where values such as the load address and relocation base address are stored.

### Runtime API

Dynamic loading is also handled by rtld. The implementation is POSIX-compliant, but the API provides some additional functionality and functions. To open a new shared library, FreeBSD has a `fdlopen()` function in addition to `dlopen()`. Instead of a path name, `fdlopen()` takes a file descriptor (fd) that is used to load the object into the address space. Setting the fd to -1 has the same effect as passing `NULL` to `dlopen()` [2]. The default visibility of the symbols is local if the global flag is not present. Additionally, three more flags can be passed to those functions. `RTLD_TRACE` causes the runtime linker to exit and print all absolute path names of all shared objects after it has loaded all needed objects. `RTLD_NODELETE` prevents the unloading of the object after passing the handler to `dlclose()`. `RTLD_NOLOAD` prevents the runtime linker to load the shared object, if it is not already loaded. When the same shared object is loaded again through `dlopen()`, a counter is used to decide when the last handle is closed with `dlclose()` and the shared object can be unloaded [2].

`dlsym()` on FreeBSD supports some flags passed as a handle. Passing `NULL` is interpreted as a lookup in the calling shared object's symbol table itself. Passing `RTLD_DEFAULT` causes the use of the default load order search algorithm for symbol lookup (see 2.4.1). `RTLD_SELF` and `RTLD_NEXT` cause a search at the object itself or the next loaded object, respectively and searches through all object loaded afterwards [2]. `dlsym()` – like `dlopen()` – has a sibling called `dlfunc()`, returning a function pointer to the search symbol instead of generic pointer (i.e. `dlfunc_t` instead of `void *`) to prevent conflicts with undefined function pointer casts in the ISO C standard.

Two additional functions are provided by FreeBSD. `dladdr()` lets one query about the shared object at a specified address and it nearest run-time symbol [1]. `dlinfo()` gives one information about the passed symbol handle from `dlopen()`, such as its entry in the link map or the search path used [42].

### 2.4.2 Linux

The dynamic linker/loader (ld-linux.so) for ELF files used by most Linux distributions are part of the GNU C Library (glibc) [6]. Their implementation is very similar to the FreeBSD one. Significant differences are explained in this section.

#### Namespaces

ld-linux.so introduces the concept of namespaces. Up to 16 of them can be used to limit visibility and search space for symbols. These namespaces are relevant for data structures, symbol resolution and the Runtime API.

#### Data structures

Every namespace has its own `struct link_namespaces`, which contains information about the loaded objects and a search list for global symbols in a particular namespace. The array of namespaces itself is part of a bigger struct (`struct rtld_global`) containing information about the global state of dynamically loaded objects in the executables' current space [5].

Like FreeBSD's rtld, ld-linux.so uses one main data structure to manage loaded executables and shared libraries. It is called `struct link_map`. Its contents largely match the FreeBSD implementation. One difference is that the first four entries of `struct link_map` are also part of the publicly accessible `struct link_map` that is returned in some calls to the runtime API, but the rest is private and might change without notice.

For storage of depending data, ld-linux.so often uses lists. These are, in contrast to FreeBSD's linked lists, implemented using arrays as underlying structure.

### Relocations

Relocations are more architecture-specific than platform-specific. Meaning, relocations are done the same way as in FreeBSD.

### Symbol resolution

Symbol resolution order is the same as in FreeBSD and defined in POSIX, but Linux has a different implementation with respect to namespaces.

The interface for a symbol lookup is `_dl_lookup_symbol_x()`. Every call goes through this function. Besides the name, reference link map and other information, it takes a list of lists containing `struct link_map`s of loaded shared objects to go through. There are two lists, both stored inside `struct link_map` of the referencing symbol. One is for global symbol lookups inside its namespace (defined in POSIX as load order) used for relocation. The other one is for local lookups inside the shared object and its dependencies (defined in POSIX as dependency order) used for `dlsym()` and `dlvsym()`. On success, it returns `struct link_map` of the defining shared object and the symbol is saved in the `relf` parameter [6].

To speed up symbol lookup, ld-linux.so also tests if the symbol defines itself before starting a full search. Also, the latest lookup is stored in the referencing `struct link_map`.

### Runtime API

The runtime API for dynamic loading, symbol lookup and information querying is also implemented in ld-linux.so. The basic API is POSIX-compliant. More functions are available – many of them equivalent to FreeBSD's additions.

To make use of the namespaces, Linux provides `dlmopen()`. It works like `dlopen()` and additionally takes an argument defining the namespace it should be loaded into. An existing namespace or two special flags can be passed. `LM_ID_BASE` is the initial namespace of the application – containing at least the executable and its dependencies. `LM_ID_NEWLM` creates a new empty namespace. `dlopen()` loads the shared library inside the namespace of the calling object. Mode flags are the same as in FreeBSD, with the addition of

RTLD_DEEPBIND, which enforces looking for unresolved symbols in the shared object itself before looking at the global namespace scope [25].

dlsym() has a sibling called dlvsym() which takes a version string of the searched symbol as an additional argument. Possible special handles are just RTLD_DEFAULT and RTLD_NEXT with the same meaning as in FreeBSD [27].

Other functions are dladdr(), dladdr1() and dlinfo(), which provide the same information as their FreeBSD equivalents and more: [24, 26]. dladdr1() has an additional parameter (comparied to dladdr()) that provides either the ELF symbol data structure or the public link map for the symbol at the provided address [24].

### 2.4.3 Windows

Shared libraries on Windows are called Dynamic-Link Libraries (DLLs). The operating system was built around that concept ever since its beginning. System APIs are exposed through exported methods (e.g. Kernel32.dll) and core functionality is imported through third-party DLLs (e.g. printer drivers) [36]. Windows has its own binary format (PE) that is used, like ELF, for executables and DLLs.

DLLs are loaded in the virtual address space of their calling process. Variables defined as global inside a DLL are global inside its process, but not shared between different processes, unless explicitly defined as such [12].

#### Exporting methods in a DLL

Methods are private until they are defined as exported. Windows provides two methods to export methods [29]. Is it needed that additionally methods can be exported later on or can the client applications be rebuild easily [28] Those are some of the aspects to be considered when deciding which method to use.

**.def file** All exported methods are defined in an extra file, called module-definition file. It is a text file with the extension .def and at least two sections. LIBRARY is followed by the DLL name. EXPORT lists all method names (mangled, if they are in C++) and optionally the ordinal value following the name and prefixed with @. The ordinal value needs to be between 1 and N, where N is the number of exported methods [31]. Listing 3 is an example of such a file [31].

```
LIBRARY   AVLTREE
EXPORTS
   Create   @1
   Insert   @2
   Delete   @3
   Find     @4
```

Listing 3: Example of a `.def` file for the DLL `AVLTREE`

Using `.def` files, one can add a new exported method and use the highest ordinal value. This does prevent an API change and the rebuild of all client applications. One can also use the the `NONAME` attribute to omit method names in the generated import library resulting in smaller DLL files for a large number of exported methods. On the other hand, one needs to put the decorated names for C++ methods into the `.def` file. The issue is that different compiler (versions) might generate different decorated names [28].

**`__declspec(dllexport)`** Instead of using an extra file, one can annotate methods with `__declspec(dllexport)` to export them. This is especially useful when exporting C++ methods. To prevent cluttering of the definitions, these annotations appear often at the method declaration in the header file. Features such as ordinals or `NONAME` cannot be used with this approach, but it can be mixed it with an additional `.def` file to gain those advantages [30]. The main problem of this annotation is the fact that the complete API might change when using another compiler or adding a new function. This also requires a rebuild of all DLL's client applications [28].

### Link/Import methods

When linking a DLL, the linker (`LINK`) creates two files: the DLL itself (e.g. `DLLNAME.dll`) and an import library (e.g. `DLLNAME.lib`). When linking another program or library depending on a DLL, it needs to pass the `.lib` file to the linker. The linker then links the DLL against this executable or library [11].

**Resolving circular dependencies** If two DLLs depend on each other, Windows offers the following solution: First, one runs `LIB` for the first DLL including the `.def` file and/or the `/EXPORT` attribute. This creates an import library (`.lib`) and a export file (`.exp`). Now one links (with `LINK`) the second library using the the import library that was just created. Finally, one links the first library by passing the `.exp` file instead of a `.def` file or the `/EXPORT` argument [33].

### Load-time dynamic linking

At program startup, the operating system does load-time dynamic linking. It looks up all depending DLLs and maps them into the virtual address space. But it does not load them into physical memory until their first use. If a DLL is not found, the process terminates and displays a dialog box [13].

**Delay loading** Starting with the Visual C++ linker 6.0, Windows also supports delay loading of DLLs. This means, needed libraries can be defined at link time and then marked as "delay load import". At runtime, the DLL gets transparently loaded via `LoadLibrary()` and `GetProcAddress()` when a function is called the first time [32].

### Run-time dynamic linking

The provided runtime API of Windows is similar to the POSIX standard, but has completely different function names. `LoadLibrary()` and `LoadLibraryEx()` are used to look for and load a new DLL. To obtain the exported function or value, `GetProcAddress()` is provided. Finally to close the opened DLL the module is handed to `FreeLibrary()` and unloaded if it was the last reference [14].

### DLL hell

DLL hell is the name for problems that occur when using DLLs. Microsoft started to acknowledge these problems and there exist multiple solutions depending on the problem. The two main problems are incompatible versions and DLL overwriting. Because no versioning system for DLLs existed, every installation overwrote the previous DLL with the same name no matter if it was a newer or older version (DLL overwriting). Some applications depending on another version broke, because the older or newer version had an incompatible change.

Today, many solutions exist to prevent overwriting, enabling coexistence of multiple versions and roll-back if the installation brakes some applications. The simplest solution is to just statically link the libraries. This removes all advantages of shared libraries. System DLLs are now protected and cannot just be overridden by anyone. Side-by-side assemblies enable having multiple versions of the same DLL (even 32 and 64 bit) installed on one system [35].

## 2.5 Barrelfish

Barrelfish is a research operating system developed by ETH Zürich in co-operation with Microsoft Research. It is a multikernel operating system [10], meaning that on every core, a small independent kernel is running, in Barrelfish called CPU driver. The rest of the operating system is structured as a distributed system without shared memory. The communication only happens via local and remote message passing [7].

### 2.5.1 Static linking in Barrelfish

Barrelfish is not self-hosting. This means that another operating system is necessary to build it (cross-compiling). The current toolchain for Barrelfish only supports static linking. Every library needed by a program, including all default system libraries, are linked statically by the program linker at compile time. More than the default set of libraries, can be added by listing them in the Hakefile.

Static linking ensures that all necessary functions and global data are available at link time and it takes most of the relocation burden away from the running system. Also, there is no need for dependency and symbol resolution. A simple ELF library handles all the loading and relocation of a newly spawned program. It provides simple symbol lookup functions using the single global symbol table available.

### 2.5.2 Hake

Hake is the build system for Barrelfish. It defines a Haskell embedded domain-specific language (DSL) to be used in Hakefiles. Every program or library directory needs a Hakefile to define the building rules. Hake collects all Hakefiles and generates one large Makefile out of it to compile every component of the system [9].

Listing 4 shows a Hakefile to build two static libraries and one depends on the other.

```
[
    build library {
        target = "foo",
        cFiles = [ "foo.c" ],
        addLibraries = libDeps [ "bar" ]
    },
    build library {
        target = "bar",
        cFiles = [ "bar.c" ]
    }
]
```

Listing 4: Example of a Hakefile for two static libraries

### 2.5.3 Error handling

Barrelfish uses `errno.h` for error handling. The the error file is generated by a small DSL called Fugu. It generates the error codes and functions for error checks and error message retrieval. It also implements a simple error stack [8]. Barrelfish-specific functions always return error values (`errval_t`). For other return values, pointers are used.

# Chapter 3

# Design

The following chapter describes the design for dynamic linking and loading in Barrelfish. Parts are still a vision. What is already implemented and what not is explained in detail in chapters 4 and 6. The design of Barrelfish's runtime linker (rtld) is inspired by the ELF runtime linkers of FreeBSD and glibc (used in Linux). The main component is a rtld library (librtld) depiced in the center of Figure 3.1.

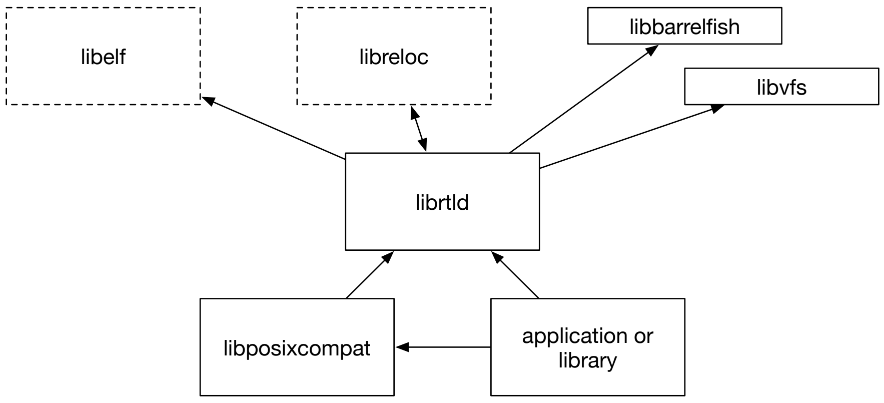

Figure 3.1: Library and application dependencies for librtld

librtld depends on two new libraries: an ELF library (libelf) and a relocation library (libreloc) as well as two system libraries: the Barrelfish library OS (libbarrelfish) and a virtual file system library (libvfs). librtld is used by libreloc, the POSIX compatibility library (libposixcompat) and directly or indirectly by applications or other libraries.

The split of the functions for dynamic linking and loading into the three libraries librtld, libelf and libreloc is chosen to reuse code for ELF parsing and relocations in different parts of the operating system and to use common libraries from other developers. Directly integrating all functionality into librtld would make it hard to reuse parts of the functionality. All these libraries are designed to be architecture independent.

This chapter shows what the purpose and interfaces of libelf, libreloc, librtld, libposixcompat and other applications and libraries is, and how they interoperate with each other.

## 3.1 ELF library

Barrelfish has a libelf currently used for parsing, loading and relocating all executables. However, the library does not implement a standard interface to just parse ELF files. This is useful because different parts of the system need ELF parsing capabilities and open source common libelf implementations exist for that purpose. Using a common library takes the implementation burden away from the Barrelfish team and provides an interface first seen in Solaris and implemented in many common libelf implementations [15].

This design is based on the decision to continue to use the ELF format for executable in Barrelfish and to start using it for shared libraries too. ELF is the format supported by the existing tool chain for building and executing shared objects and used in many open source and closed source operating systems today [39].

## 3.2 Relocation library

Relocations are also needed at different places in the operating system. A common library providing this functionality does, however, not exist. But the Barrelfish team is motivated to build such a library that implements the architecture specific relocations and provides a simple interface to this.

libreloc defines four functions for its clients. `reloc_rel()` does all relocations needed before anything in the shared object can be executed. These relocations are for position independent code primary global data relocations in the GOT. `reloc_plt_fixup()` is used when lazy (function) relocations are used. This sets the correct first two GOT entries and updates the GOT entries for the corresponding PLT entries to point to the right address. `reloc_plt_entry()` is used to do the relocation of a specific PLT entry at runtime. Finally `reloc_plt()` is used when the PLT is not relocated lazily.

Many relocations need to look up symbols. libreloc expects this functionality to be provided by its clients who need to provide an appropriate function.

## 3.3 rtld library

librtld is a shared library provided by Barrelfish that is loaded together with the executable and then called before the program's entry point. Started by the operating system librtld sets up itself and then runs the steps provided on page 8 with the executable as the *defined shared object*.

At runtime, librtld defines an API for dynamic loading of shared libraries. This API is Barrelfish specific using its own error handling system (fugu) but borrows most functions defined in POSIX. This provides an interface familiar to may programmers, but uses specific error extensions provided by Barrelfish. The following functions are available: `rtld_dlopen()`, `rtld_dlsym()` and `rtld_dlclose()`. `rtld_dlerror()` does not exist, because an error message can be obtained from the returned error values. The steps for opening are the same as starting a new executable (see page 8). Symbol lookup via `rtld_dlsym()` wraps `find_symbol_dependency_order()` looking for symbols in the loaded shared object itself and all its dependencies. For the executable additionally all shared objects loaded with global availability are searched.

For ELF parsing, librtld calls into libelf and for all the relocation libreloc is used. Starting with `reloc_rel()` and then depending on the modes provided via environment variables or mode flags in `rtld_dlopen()` the lazy relocation function or the other is called.

For symbol lookup librtld provides the `lookup_symbol()` function – doing a load order symbol lookup – for libreloc. For runtime lookups by other clients `rtld_dlsym()` is provided (see above for more details).

## 3.4 POSIX compatibility library

In order to provide a POSIX-compliant interface, libposixcompat provides a wrapper around the runtime functions provided by librtld. This makes porting applications from other platforms convenient, because no changes are needed as long as they only use the standardized functions and constants.

## 3.5 Application/Library

Applications and libraries can choose between the public interface of librtld or libposixcompat to load dynamically shared libraries, look up symbols and close them afterwards. Both APIs are similar. The librtld interface integrates more into Barrelfish and provides a familiar API with native error handling. The libposixcompat keeps an application more portable by using the standardized interface.

# Chapter 4

# Implementation

The implementation is heavily influenced by FreeBSD's rtld and Barrelfish's current libelf and libspawndomain. Barrelfish's libraries help to understand how operating system specific APIs work and FreeBSD's rtld shows how a lightweight implementation of a runtime linker might be designed and implemented. The current implementation works with shared libraries that are dynamically loaded at runtime using the runtime API. Shared libraries might have static or dynamic dependencies that are automatically loaded and relocated if needed. However, this implementation only works on the x86-64 architecture and does not cover the complete design. To see what is missing consult chapter 6.

This chapter starts with the different approaches leading to the final choice and then covers different aspects of the implementation. The description of the final implementation starts with the data structures and memory allocation used. It continues describing how symbols and dependencies are resolved and how the runtime APIs look and work. And finally describes the changes made to Hake and requirements for the shared library ELF file.

This chapter covers just the implementation of librtld, because libreloc and libelf are not part of this work. Nevertheless, librtld is implemented with those in mind.

## 4.1 History

The implementation started with code in a single application that dynamically loaded a shared library. This code was then moved into a pseudo implementation of the dynamic loading POSIX interface, already part of Barrelfish. Both implementations depended on an alternative load function implementation in the current libelf. The default implementation had two problems. It allocated memory for each load segment independently and did relocation after each mapping of a load segment. This is not suitable for shared libraries. First because position independent code depends on fixed offsets the allocated space for the library needs to be contiguous to accommodate all load segments. Second relocation should happen the earliest after all segments are loaded, because there is one relocation table for all segments and needed data structures during relocation might be part of another segment.

In order to check what the current implementation supports, different shared libraries were used. The simplest was a shared library that did not depend on any other library. This test case worked from the beginning. The more advanced version was a shared library statically linked to its dependencies. In this case relocations were needed for global functions and data, but just one symbol table existed. So no symbol lookup in multiple tables and dependency resolution is needed. This case worked also from the start with additional changes to the load function regarding load order and more supported relocations. Finally the last test case was a shared library depending on another shared library. To implement startup of a dynamically linked executable the same functionality is needed with custom initialization. To get this working, all steps described on page 8 need to be done in this specific order. This use case did not work with the first two implementations, because no dependency resolution and advanced symbol lookup was possible.

Significant changes would have been needed for libelf implementing these steps in this particular order, breaking most of the operating system's functionality. Barrelfish depends heavily on the current libelf implementation. Instead, a new library – librtld – was implemented.

## 4.2 Data structures

In order to support lists containing an undefined number of objects, different structs exist depending on the information stored at each entry. `struct shared_object_name` is used to store a name at every entry. To point to a shared object and its name offset into the string table `struct shared_object_needed` is used. And finally `struct open_object` is a linked list pointing to a shared object at every entry. Normally the pointer to the first entry is stored in a variable, but sometimes also a pointer to the tail to make appending at the end of the list faster.

The main data structure used is `struct shared_object_info` containing all needed information for a shared object loaded by librtld. Figure 4.1 shows the different information segments inside `struct shared_object_info`.

The first section of information contains a linked list oft `struct shared_`

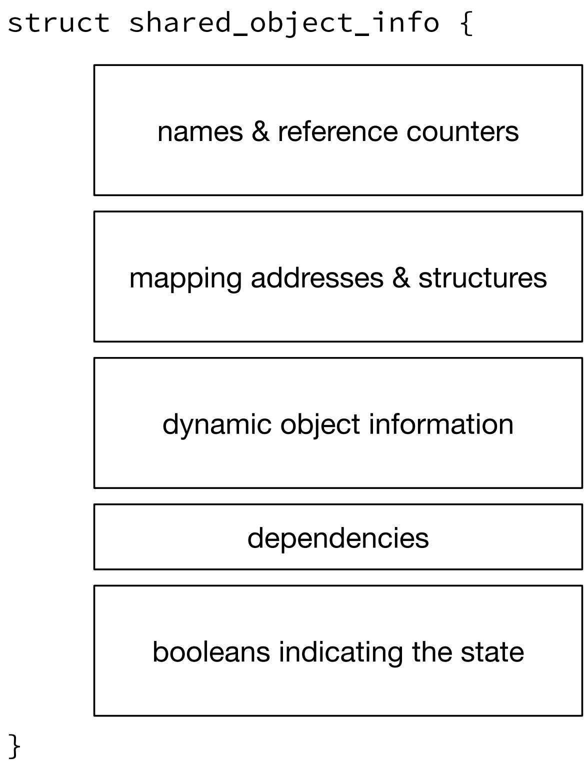
struct shared_object_info {
names & reference counters
mapping addresses & structures
dynamic object information
dependencies
booleans indicating the state
}

Figure 4.1: Information contained in `struct shared_object_info`

`object_name` where all names and paths are stored that were used to reference this shared object. If a client requests the loading of a new shared object first every name in all objects is checked in order to prevent the loading of the same shared library again. The section also contains reference counters containing the number of references to this object (`ref_count`) and the number of openings through `rtld_dlopen()` (`dlopen_count`). These are used to decide if a shared object can be unloaded or is still in use. At the end there is a pointer to the next loaded shared object. This pointer is used for load order symbol lookups and to load all needed objects.

The second section is populated by `load_shared_object()` and `map_shared_object()`. The `path` is used for debug outputs. The base address in memory (`map_base`), the requested size (`map_size`), the base address of the virtual addresses in the ELF file (`vaddr_base`) and the relocations base address (`reloc_base`) are used for relocation and debugging. The pointer to the dynamic section is important because it contains information used to populate the third section of this struct. `memobj`, `vregion`, `frames` and `no_of_frames` store the Barrelfish specific information regarding allocated memory and are used to destroy the allocated space. `entry_pt` stores the entry point of the shared object. This is normally only useful for program executables and points to the location where execution should start. It might be used to initialize a shared library, but this should be done with init functions (currently not implemented).

The next section contains information extracted from the dynamic ELF segment and filled in by `digest_dynamic()`. It stores a linked list of all needed objects (`needed`), the address of the PLT or GOT table (`pltgot`), symbol and string table (`symtab`, `symtab_count`, `strtab` & `strtab_size`), links to the relocations required at load time (`rela` & `rela_size`) and relocations that can be delayed (`plt_rela` & `plt_rela_size`) and information about the hash table (`buckets`, `nbuckets`, `chains` & `nchains`). All this information is needed for dependency resolution, relocation and symbol lookups.

A linked list of all dependencies in breadth-first order (`dependencies`) is needed for dependency order symbol resolution. And `dlopenroots` contains the roots of the dependency graphs this shared object is part of. These roots are used for load order symbol lookups (e.g. at relocation).

Finally booleans are used to indicate if it is an executable (`main_exec` – for future use), if the shared object is already relocated (`rela_relocated` & `plt_relocated`), if a valid hash table exists (`valid_hash_tab`) and if a dependency graph exists (`dep_created`).

## 4.3 Memory allocation

Memory allocation for a new shared library is based on the x86 implementation of `elf_allocate()` and part of the `map_shared_object()` function. It first optimized the size of the allocated space, because a single frame is always a multiple of two. After a memory object is created it is mapped into a virtual address space and filled with the allocated frames. Finally the load segments are copied into the virtual address space and its range is set to the rights defined for this segment.

## 4.4 Dependency resolution

The needed dependencies are declared in the dynamic section and stored in a linked list starting at the `needed` entry. The list consists of `struct shared_object_info`. `load_needed_objects()` iterates over this list and loads the dependencies and then the dependencies of the dependencies and so on until the complete dependency graph is loaded in breadth-first order. By only checking the dependencies of new loaded objects, the function prevents infinite loops for circular dependencies.

## 4.5 Symbol resolution

The two symbol resolution modes described in the POSIX standard are supported [21]. `find_symbol_load_order()` does a load order lookup for relocation or global symbol searches. Starting with all shared objects that are loaded with `rtld_dlopen()` in global mode it then checks all the dependency graphs the referencing shared object is part of. If no matching symbol was found or the found definition is weak. `find_symbol_dependency_order()` searches itself and its dependencies in breadth-first order and is used by `rtld_dlsym()`.

In order to prevent multiple lookups in the same symbol table both functions keep a `struct checked_objects` containing an array of all visited objects.

## 4.6 Runtime API

librtld provides a native runtime API based on the POSIX interface with implementation and Barrelfish specific adaptions. libposixcompat implements a wrapper around this API to provide a POSIX-compliant implementation.

### 4.6.1 Native API

The native interface has the public functions shown in Listing 5.

```
errval_t rtld_dlopen(const char *filename, int32_t flags,
                     struct shared_object_info **obj);
errval_t rtld_dlsym(struct shared_object_info *obj,
                    const char *restrict symbol,
                    lvaddr_t *addr);
errval_t rtld_dlclose(struct shared_object_info *obj);
```

Listing 5: Public API of librtld

The public functions return an error value starting with `RTLD_ERR_` or `RELOC_ERR_`. It is a native Barrelfish error and indicates the success or failure of the called function.

`rtld_dlopen()` is used to load a new shared object located at `filename` or obtain the reference to an already loaded object. On success `obj` is filled with the pointer to the corresponding `struct shared_object_info`. The only flag currently implemented is `RTLD_GLOBAL` to make the shared object globally available in future relocations.

`rtld_dlsym()` can be used to look up the address of a global function or object by name (`symbol`). The provided `struct shared_object_info` is valid if librtld knows this object and the shared library is opened via `rtld_dlopen()`. The return value indicates if a symbol was successfully found and the address is returned through `addr`.

`rtld_dlclose()` decrements the reference pointer and unloads the passed `obj` if it is no more referenced.

### 4.6.2 POSIX API

The POSIX interface wraps the native API by redirecting the error to `dlerror()` and casting the object and address to and from `void *` pointers. It has the same limitation for flags like `rtld_dlopen()`.

## 4.7 Hake

Hake was extended to build shared libraries and add them as dependencies to other build targets.

To build a shared library one defines a new build target using `build sharedLibrary`. This creates a target for `lib + targetname.so` in the `lib` folder. The new target is achieved by introducing a new linker in Hake called `dynLinker`. It works like the existing linker, but takes another set of flags – `optLdDynFlags`. The default flags defined by `ldDynFlags` exclude standard libraries from gcc and add `-shared` needed to compile shared libraries and `-Wl,--hash-style=both` to create symbol hash tables in ELF and GNU style[1]. This linker currently only creates targets for the x86-64 architecture and there it adds `-fPIC` to all build targets, because shared libraries need to be position independent code and no problem was observed to use this with executables, too.

The standard to build a static library was to define it with `build library`. This target is now extended to create targets for static and shared libraries. To only create a static library the newly defined `build staticLibrary` target is used.

[1]As default gcc only creates the GNU style hash tables.

If a build target depends on another shared library it can now define that by using the addSharedLibraries keyword. It works like addLibraries for static libraries.

If we take the example from Listing 4 and make the libraries dynamically loaded we get Listing 6.

```
[
    build library {
        target = "foo",
        cFiles = [ "foo.c" ],
        addSharedLibraries = libDeps [ "bar" ]
    },
    build library {
        target = "bar",
        cFiles = [ "bar.c" ]
    }
]
```

Listing 6: Example of a Hakefile for two shared libraries

## 4.8 ELF requirements

The current implementation of Barrelfish's librtld requires some segments to be in any shared library: First every shared library with any global symbol needs to contain a .hash section (DT_HASH entry in the dynamic section) that contains a hash table defined by the *ELF-64 Object File Format* definition [17]. The GNU hash table format (.hash.gnu section referenced by the DT_GNU_HASH entry in the dynamic section) is currently not supported. Additionally to make symbol lookup possible there needs to be a symbol table and a string table. Finally it is assumed that any other needed shared library has a DT_NEEDED entry in the dynamic section.

# Chapter 5

# Evaluation

For the evaluation section an application was created to demonstrate and test the functionally currently implemented (Demo1). Figure 5.1 shows the shared objects involved and the dependency graph. The main application calls `dlopen()` for every of its dependencies. The other dependencies are loaded automatically. Via `dlsym()` various data and functions are retrieved and accessed or called. Before returning the application will close all its dependencies.

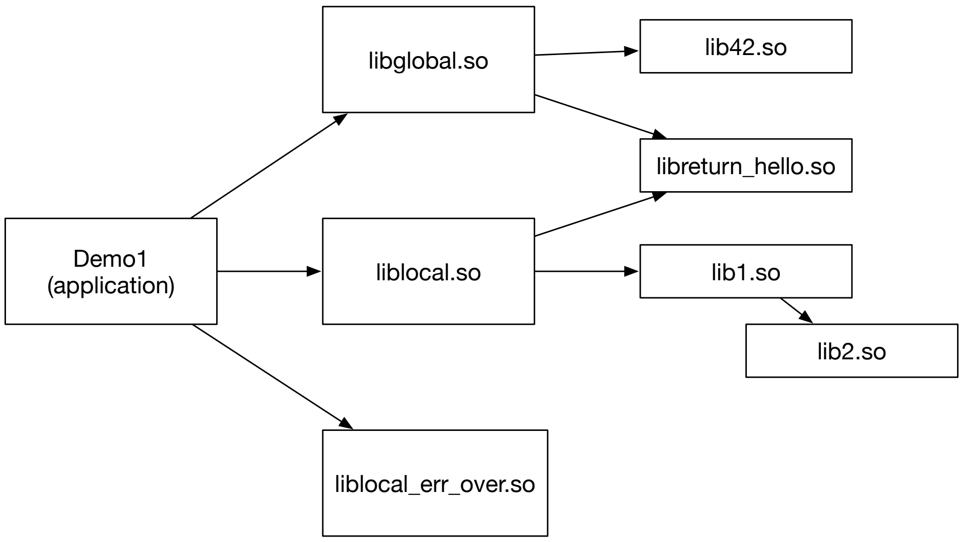

Figure 5.1: Demo application (Demo1) and its dependency graph

In the first section of this chapter binary sizes of statically and dynamically linked binaries are compared and the second section compares the speed of `dlopen()` in Barrelfish and Linux.

## 5.1 Binary sizes

Comparing binary sizes of statically linked applications to the size of the same applications dynamically linked and each dynamic library leads to problems with the current Hake implementation. There does not exist a way to completely dynamically link an executable to the default system libraries as well as libbarrelfish and libnewlib (two of the default system libraries) currently do not compile as shared libraries. These are the reasons no simple applications exclusively depending on system libraries are compared, but the demo application (Demo1) and a second application (Demo2) depending on a set of other shared libraries. Demo2 only depends on the general library. The resulting file sizes are shown in Table 5.1.

| Name | static linked | dynamic linked |
|---|---|---|
| Demo1 | 8974406 | 8972887 |
| Demo2 | 8972433 | 8971536 |
| libglobal.so | | 7452 |
| liblocal.so | | 7512 |
| liblocal_err_over.so | | 7094 |
| lib42.so | | 6396 |
| libreturn_hello.so | | 6764 |
| lib1.so | | 7281 |
| lib2.so | | 7077 |
| Sum | 17948812 | 17993999 |

Table 5.1: File sizes (in bytes) for static and dynamic linked Demo1 & Demo2

The dynamically linked libraries are smaller than the equivalent statically linked libraries, but the difference are only a few kilobytes. Summing all up for just a Demo1 and Demo2 application dynamically linked they need more space. If the difference between the the sums (45187 bytes) is taken and divided by the saved space of every additional Demo1 and Demo2 application (2419 bytes) it results in 19 Demo1 and 19 Demo2 applications needed to save space compared to the statically linked applications. Because only non-system libraries are compared this result is much more significant if all system libraries can be dynamically linked. Then every library needs to be stored only once and the binaries will get even smaller compared to their statically linked counterparts. Having more than 37 applications (all depending on the same system libraries) is to be expected in a normal setting.

## 5.2 Speed of `dlopen()`

The speed of `dlopen()` was compared on Linux (Ubuntu 14.04 LTS) and Barrelfish with the new librtld and libposixcompat implementation. The benchmark was run on the test machine “babybel4”. The hardware specification can be found in Table 5.2.

| | |
|---|---|
| CPU | Intel® Xeon® CPU E5-2670 v2 |
| Cores | 2 x 10 Cores |
| Frequency | 2.5 GHz |
| Main memory | 16 x 16 GB |

Table 5.2: Hardware specification of “babybel4”

### 5.2.1 Methodology

The libraries libglobal and liblocal from the demo application were loaded via `dlopen()`. The cycles needed to open both libraries after one another was measured. Then the handles were closed (not measured). This was repeated 30 times and the average of cycles was calculated.

### 5.2.2 Results

The benchmark was run on each operating system seven times. The average of each benchmark and the average over all runs are listed in Table 5.3. On average `dlopen()` is about 44 times faster in Linux compared to Barrelfish.

| Run | Linux | Barrelfish |
|---|---|---|
| 1 | 363165 | 15917151 |
| 2 | 351449 | 16305795 |
| 3 | 352272 | 14828584 |
| 4 | 343797 | 14573419 |
| 5 | 347121 | 14621588 |
| 6 | 347707 | 15534010 |
| 7 | 345303 | 16603307 |
| Average | 350116 | 15483408 |

Table 5.3: Average of cycles at every benchmark and average over all runs

### 5.2.3 Discussion

The results show that `dlopen()` is about 44 times faster in Linux. Two reasons that support this result are: First dynamic loading (and linking) is a central part of Linux to run any program. Many applications today would not run without this feature. Second the librtld implementation in its first implementation was not optimized for speed but rather to function in the Barrelfish environment and to be improved and extended in the future.

# Chapter 6

# Future work

This chapter contains two parts. Missing implementation of parts that are already described in the current design throughout this work and design and implementation ideas to extend the current functionality.

## 6.1 Missing implementation of design features

### 6.1.1 librtld

librtld has some major and minor features missing in the current implementation.

The first big missing part is the loading and invocation of librtld at program startup. This enables dynamic linking for executables and global symbol lookups starting at the executable. To implement this feature an entry function needs to be implemented that sets up the rtld itself, the executable and its dependencies. The operating system needs to be aware of this and needs to call into the rtld before the entry point of the executable is called. However, most of the infrastructure already exists. Functions for loading of shared objects, dependency resolution, symbol lookup and relocation already exist.

Additionally there are currently problems to load libbarrelfish dynamically. This needs some further investigation, before a program and its dependencies can be dynamic linked and loaded.

The second big feature missing are lazy relocations. Implementation of this feature could improve startup time of shared objects, because it does not relocate functions that are never called. To support this feature a runtime entry function needs to be implemented to call into librtld and relocate the appropriate function. During the relocation at loading also the second and third GOT entries need to be set to the right address as well as all GOT entries corresponding to a PLT entry need to be updated with the relocated PLT entry addresses by adding the relocation offset to the each entry.

Because of these two missing features, `dlopen()` lacks flags that are needed to achieve POSIX-compliance. With dynamic linking, passing of `NULL` could return a global symbol table and lazy relocations could be enabled with `RTLD_LAZY` or the current implementation enforced with `RTLD_NOW`. Even though they are not part of the standard, support for the special handles `RTLD_DEFAULT` and `RTLD_NEXT` in `dlsym()` would be useful. The first handle starts the lookup in the global scope and the second starts the lookup in loading order after the shared object calling `dlsym()`.

Multithreading is not directly mentioned in the design, but is needed to have a full featured implementation of a runtime linker. The current implementation has no locks to prevent race conditions in librtld if two threads from of the same application access it. An implementation of reader and writer locks can solve this situation. Access to the same resource for reading is no problem, but writes should not interleave with reading access and writes should be atomic.

Minor features missing are the support for init and fini functions that are called before executing any code in a shared object and before unloading the object respectively. The implementation of this feature needs lists to keep tracks of init and fini functions in the order they need to be called and two functions that call these functions.

### 6.1.2 libelf & libreloc

libelf and libreloc are not part of this work and as such currently do not exist in Barrelfish. However, it is planned to add them to Barrelfish in the future and librtld is implemented with these libraries in mind. The relocation code is in a extra `reloc.c` file using and providing the API described in the design chapter. On the other hand, libelf will be integrated much more tightly and thus the current implementation lacks complete separation of ELF parsing from the rest. Refactoring is needed to integrate with the future libelf.

## 6.2 Extensions

Three extensions with small impact on the design are:

- Tracing or debugging is not possible in the current implementation and environments variables to enable these and more features are missing, too.

- An elaborate search order for shared libraries is missing in the current design and implementation. The implementation works like this: `/x86_64/lib` is added in front of the file string if its a name or relative path and the provided path is used as it is if it is absolute. A more sophisticated search order that takes environment variables and rpath or runpath into account would be useful to have for more locations to store libraries.
- To speed up symbol lookup GNU hash tables where introduced by the GNU team. In comparison to the standard ELF hash algorithm it has a bloom filter before accessing the hash table to minimize the length of the hash chain and enable a better distribution of the entries.

More elaborate extensions are described in the following three sections.

### 6.2.1 Sharing of read only segments

One benefit of shared libraries on Linux, FreeBSD and Windows is the sharing of read only segments. This is a useful feature in Barrelfish, too. But some research needs to be done to learn when sharing of these sections is useful. Probably only sharing of code running on the same core is useful, because every process has the same local memory. The benefits might be lower between different cores, because they have different local memory and access of remote memory introduces takes more time than for local memory.

### 6.2.2 Symbol versions

Linux and FreeBSD have the concept for symbol versions to mark incompatible changes in the API by bumping the version of the affected symbols. Dependencies in shared libraries then depend on a specific version of such a symbol and the runtime linker check at symbol lookup if the right version is available. This makes API changes more easily to handle and should be taken into consideration when extending the current librtld design and implementation.

### 6.2.3 Namespaces

Namespaces is a concept introduced by Linux for the dynamic loading mechanism. It allows encapsulation of dynamically loaded libraries with no dependencies on already loaded shared objects in other namespaces. This could be useful for loading untrusted code (e.g. plugins or drivers).

# Chapter 7

# Conclusion

The first step was the exploration of the implementation of dynamic linking and loading capabilities in different operating systems, which showed that there is a big variety of implementation details. Nevertheless, operating systems that use the same binary format have similar implementations sometimes also defined by the binary file.

The design showed that a new set of libraries is needed in Barrelfish to implement a full featured runtime linker and how the implementation used many concepts from other operating systems and adapted them to run well integrated with Barrelfish. The results on binary sizes are promising that in the future space can be reduced on disk and in memory. However, the measurements and missing implemented features show, that there is still engineering work needed in order to provide a full featured rtld in Barrelfish.

# Glossary

**ABI** application binary interface. 6

**API** application programming interface. 6, 18–21, 26, 27, 31, 32, 39, 40, 50

**COFF** Unix Common Object File Format. 11

**DLL** Dynamic-Link Library. 19–21, 50

**DSL** domain-specific language. 22, 23

**dynamic linking** Loading and linking during load time of a new program. Ensures that all shared libraries exist and are loaded und linked until the execution is passed to the program. Otherwise the execution normally stops and the running of the program fails. It increases startup time, because all is loaded before any execution of the program itself. (e.g. in C dynamic liking happens before `main` is called). 5, 42

**dynamic loading** Loading and linking during execution (i.e. at runtime) of a program. Done via system calls. One can speed up startup time by loading libraries only later when they are used or they are not essential for the program. One can also handle the case that the shared library does not exist. This cannot be done for dynamic linking. 5, 7, 11, 16, 18, 26

**ELF** Executable and Linkable Format. 8, 9, 11, 16, 17, 19, 22, 24–27, 29, 30, 32, 39, 40, 43, 48

**glibc** GNU C Library. 17, 24

**GOT** global offset table. 9–11, 14, 15, 25, 30, 39

**IAT** import address table. 11

**ld-linux.so** dynamic linker/loader. 17, 18

**libbarrelfish** Barrelfish library OS. 24, 35, 38

**libelf** ELF library. 24–28, 39

**libposixcompat** POSIX compatibility library. 24–26, 31, 36

**libreloc** relocation library. 24–27, 39

**librtld** rtld library. 24–28, 31–33, 36–40, 48, 50

**libvfs** virtual file system library. 24

**PE** Portable Executable. 11, 19

**PLT** procedure linkage table. 9–11, 14, 15, 25, 30, 39, 48

**POSIX** Portable Operating System Interface. 7, 11–13, 16, 18, 21, 24, 26, 27, 31, 32, 39, 43, 50

**program linker** Offline linker that links the object files into an executable. `gcc` for example automatically invokes the program linker after compilation. 8, 22

**rtld** run-time link-editor. 14–17, 27

**rtld** runtime linker. 24, 38, 41

**runtime linker** Also dynamic linker. Linker invoked at runtime to handle dynamically loaded executables and shared libraries. 8, 11

**shared object** An executable or shared library. 8–10, 14–17, 25, 26, 28–31, 34, 38–40

# Bibliography

[1] FreeBSD 10.1. *dladdr – find theshared object containing a given address*. 1998-02. URL: `https://www.freebsd.org/cgi/man.cgi?query=dladdr&sektion=3&manpath=FreeBSD+10.1-RELEASE` (visited on 2015-08-08).

[2] FreeBSD 10.1. *dlopen, fdlopen, dlsym, dlfunc, dlerror, dlclose – programmatic interface to the dynamic linker*. 2011-12. URL: `https://www.freebsd.org/cgi/man.cgi?query=dlopen&sektion=3&manpath=FreeBSD+10.1-RELEASE` (visited on 2015-08-08).

[3] FreeBSD 10.1. *ld-elf.so.1, ld.so, rtld – run-time link-editor*. 2012-06. URL: `https://www.freebsd.org/cgi/man.cgi?query=rtld&sektion=1&manpath=FreeBSD+10.1-RELEASE` (visited on 2015-07-28).

[4] FreeBSD 10.1. *rtld-elf source code*. 2014-12. URL: `https://svnweb.freebsd.org/base/release/10.1.0/libexec/rtld-elf/` (visited on 2015-07-28).

[5] glibc 2.22. *glibc source code*. 2015-08. URL: `https://sourceware.org/git/?p=glibc.git;a=tree` (visited on 2015-08-12).

[6] glibc 2.22. *ld-linux.so source code*. 2015-08. URL: `https://sourceware.org/git/?p=glibc.git;a=tree;f=elf` (visited on 2015-08-12).

[7] Team Barrelfish. *Barrelfish Architecture Overview*. Tech. rep. 000. Systems Group, ETH Zurich, 2013-12. URL: `http://www.barrelfish.org/TN-000-Overview.pdf`.

[8] Team Barrelfish. *Barrelfish Glossary*. Tech. rep. 001. Systems Group, ETH Zurich, 2013-12. URL: `http://www.barrelfish.org/TN-001-Glossary.pdf`.

[9] Team Barrelfish. *Hake*. Tech. rep. 003. Systems Group, ETH Zurich, 2010-06. URL: `http://www.barrelfish.org/TN-003-Hake.pdf`.

[10] Andrew Baumann et al. “The Multikernel: A new OS architecture for scalable multicore systems”. In: *Proceedings of the 22nd ACM Symposium on OS Principles.* ACM, 2009. URL: `http://www.barrelfish.org/barrelfish_sosp09.pdf`.

[11] Windows Dev Center. *Dynamic-Link Library Creation.* URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682592(v=vs.85).aspx` (visited on 2015-08-14).

[12] Windows Dev Center. *Dynamic-Link Library Data.* URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682594(v=vs.85).aspx` (visited on 2015-08-14).

[13] Windows Dev Center. *Load-Time Dynamic Linking.* URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms684184(v=vs.85).aspx` (visited on 2015-08-14).

[14] Windows Dev Center. *Run-Time Dynamic Linking.* URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms685090(v=vs.85).aspx` (visited on 2015-08-15).

[15] Oracle Corporation. *elf – object file access library.* URL: `http://docs.oracle.com/cd/E26502_01/html/E29036/elf-3elf.html#REFMAN3Delf-3elf` (visited on 2015-08-15).

[16] Oracle Corporation. *The Use of `-Bsymbolic`.* 2010. URL: `http://docs.oracle.com/cd/E19957-01/806-0641/chapter4-16/index.html` (visited on 2015-08-11).

[17] *ELF-64 Object File Format.* Version 1.5 Draft 2. 1998. URL: `http://www.uclibc.org/docs/elf-64-gen.pdf`.

[18] The IEEE and The Open Group. *dlclose - close a symbol table handle (POSIX.1-2008).* 2013. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlclose.html` (visited on 2015-08-07).

[19] The IEEE and The Open Group. *dlerror - get diagnostic information (POSIX.1-2008).* 2013. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlerror.html` (visited on 2015-08-07).

[20] The IEEE and The Open Group. *dlfcn.h - dynamic linking (POSIX.1-2008).* 2013. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/dlfcn.h.html` (visited on 2015-07-28).

[21] The IEEE and The Open Group. *dlopen - open a symbol table handle (POSIX.1-2008).* 2013. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlopen.html` (visited on 2015-08-07).

[22] The IEEE and The Open Group. *dlsym - get the address of a symbol from a symbol table handle (POSIX.1-2008).* 2013. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlsym.html` (visited on 2015-08-07).

[23] The IEEE and The Open Group. *The Open Group Base Specifications Issue 7 / IEEE Std 1003.1, 2013 Edition.* 2013. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/` (visited on 2015-08-10).

[24] Linux. *dladdr, dladdr1 - translate address to symbolic information.* 2015-08. URL: `http://man7.org/linux/man-pages/man3/dladdr.3.html` (visited on 2015-08-11).

[25] Linux. *dlclose, dlopen, dlmopen - open and close a shared object.* 2015-08. URL: `http://man7.org/linux/man-pages/man3/dlopen.3.html` (visited on 2015-08-11).

[26] Linux. *dlinfo - obtain information about a dynamically loaded object.* 2015-08. URL: `http://man7.org/linux/man-pages/man3/dlinfo.3.html` (visited on 2015-08-11).

[27] Linux. *dlsym, dlvsym - obtain address of a symbol in a shared object or executable.* 2015-08. URL: `http://man7.org/linux/man-pages/man3/dlsym.3.html` (visited on 2015-08-11).

[28] Microsoft Developer Network. *Determining Which Exporting Method to Use.* URL: `https://msdn.microsoft.com/en-us/library/900axts6.aspx` (visited on 2015-08-14).

[29] Microsoft Developer Network. *Exporting from a DLL.* URL: `https://msdn.microsoft.com/en-us/library/z4zxe9k8.aspx` (visited on 2015-08-14).

[30] Microsoft Developer Network. *Exporting from a DLL Using __declspec(dllexport).* URL: `https://msdn.microsoft.com/en-us/library/a90k134d.aspx` (visited on 2015-08-14).

[31] Microsoft Developer Network. *Exporting from a DLL Using DEF Files.* URL: `https://msdn.microsoft.com/en-us/library/d91k01sh.aspx` (visited on 2015-08-14).

[32] Microsoft Developer Network. *Linker Support for Delay-Loaded DLLs.* URL: `https://msdn.microsoft.com/en-us/library/151kt790(v=vs.140).aspx` (visited on 2015-08-15).

[33] Microsoft Developer Network. *Using an Import Library and Export File.* URL: `https://msdn.microsoft.com/en-us/library/kkt2hd12.aspx` (visited on 2015-08-14).

[34] Ian Lance Taylor. *Linkers part 4 – Shared Libraries.* 2007. URL: `http://www.airs.com/blog/archives/41` (visited on 2015-07-23).

[35] Wikipedia. *Dependency hell — Wikipedia.* 2015. URL: `http://en.wikipedia.org/w/index.php?title=Dependency_hell&oldid=650338824` (visited on 2015-03-29).

[36] Wikipedia. *Dynamic-link library — Wikipedia.* 2015-07. URL: `https://en.wikipedia.org/w/index.php?title=Dynamic-link_library&oldid=672654838` (visited on 2015-08-14).

[37] Wikipedia. *Dynamic linker — Wikipedia.* 2015-08. URL: `https://en.wikipedia.org/w/index.php?title=Dynamic_linker&oldid=677486739` (visited on 2015-08-23).

[38] Wikipedia. *Dynamic loading — Wikipedia.* 2015-08. URL: `https://en.wikipedia.org/w/index.php?title=Dynamic_loading&oldid=676666342` (visited on 2015-08-23).

[39] Wikipedia. *Executable and Linkable Format – Applications — Wikipedia.* 2015-08. URL: `https://en.wikipedia.org/w/index.php?title=Executable_and_Linkable_Format&oldid=674682573#Applications` (visited on 2015-08-23).

[40] Wikipedia. *Library (computing) – Shared libraries — Wikipedia.* 2015-07. URL: `https://en.wikipedia.org/w/index.php?title=Library_(computing)&oldid=672741711#Shared_libraries` (visited on 2015-08-14).

[41] Wikipedia. *Portable Executable — Wikipedia.* 2015-05. URL: `https://en.wikipedia.org/w/index.php?title=Portable_Executable&oldid=664270315` (visited on 2015-08-23).

[42] FreeBSD 10.1: Alexey Zelkin. *dlinfo – information about dynamically loaded object.* 2003-02. URL: `https://www.freebsd.org/cgi/man.cgi?query=dladdr&sektion=3&manpath=FreeBSD+10.1-RELEASE` (visited on 2015-08-08).

# List of Figures

# List of Tables

# List of Listings

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):

Dynamic Linking and Loading in Barrelfish

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

| **Name(s):** | **First name(s):** |
|---|---|
| Keller | David |

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)** |
|---|---|
| Wohlen, 23. August 2015 | [illegible] |

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