Andrew Scull

Hephaestus: a Rust runtime for a distributed operating system

Computer Science Tripos – Part II

St John's College

15th May 2015

Hephaestus (Ηφαιστος) is the Greek god of blacksmiths and craftsmen. In Greek mythology, Hephaestus makes the weapons for the gods: perhaps an apt analogy for a high-level language runtime.

Proforma

Name:	**Andrew Scull**
College:	**St John's College**
Project Title:	**Hephaestus: a high-level language runtime for a distributed data centre operating system**
Examination:	**Computer Science Tripos – Part II, June 2015**
Word Count:	**11863**
Project Originator:	Andrew Scull & Malte Schwarzkopf
Supervisor:	Ionel Gog & Malte Schwarzkopf

Original Aims of the Project

To bring the benefits of the high-level programming language Rust to DIOS, a distributed operating system for data centres. Support for Rust simplifies writing programs for DIOS, which previously could only be written in low-level languages (such as C) directly against the system call API.

Work Completed

All success criteria have been met and many optional extensions have also been implemented. Multi-tasked Rust programs work on DIOS. The Rust source code is more concise, clearer and safer than the equivalent C source code. It is also able to achieve performance equivalent to that of a low-level C implementation written directly against the DIOS system call API.

Special Difficulties

I had to work with two experimental code bases that frequently changed drastically during the project: a pre-alpha programming language and compiler (Rust) and a research operating system (DIOS). Moreover, significant OS kernel modifications created challenging design and failure conditions.

Declaration

I, Andrew Scull of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

This dissertation describes the process of adding support for using the high-level programming language Rust[1] to DIOS, a distributed operating system for data centres. As a result of my work, programs can be written in far fewer lines and with a much reduced risk of errors, without sacrificing performance.

1.1 Motivation

Modern web-based applications such as Facebook, Google and YouTube are backed by large data centres that process enormous amounts of data. The sheer volume of the data means that many machines are required. This has led to the creation of data centres in which thousands of commodity machines work together to complete common tasks.

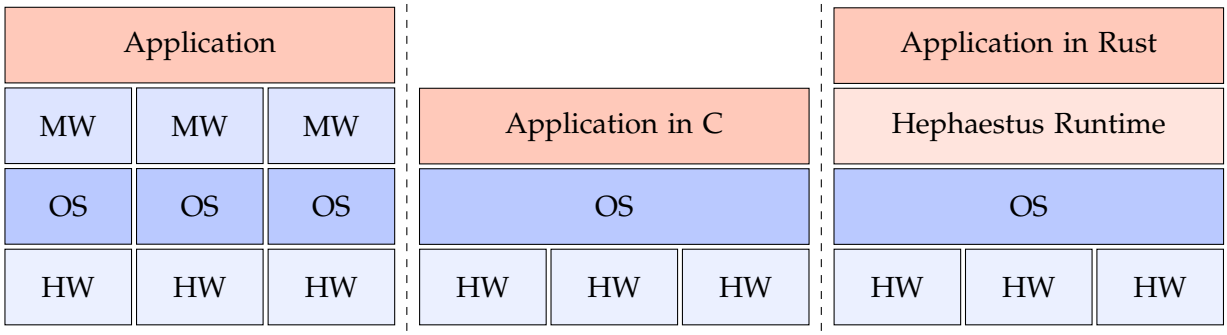
The use of cost-effective commodity hardware and the degree of parallelism means that failures are common in this environment. The role of software now extends to being responsible for managing these failure conditions gracefully, as well as other challenges of distributed programming, such as synchronisation across machines. This adds significant complexity for programmers developing on these platforms, reducing productivity and increasing the potential for errors.

The solution is to abstract and indirect via distributed systems in middle-ware. In current state-of-the-art systems, the first layer of abstraction handles lower level details such as hardware failure, resource synchronization [10,28] and distributed data storage [12,17,21]. The next level of abstraction builds on top of these systems to provide high-level frameworks for executing parallel "tasks" across machines in the data centre.

[1] http://www.rust-lang.org/; accessed 12/05/2015.

This provided developers with a much simpler, more productive and less error prone environment. MapReduce [16], Dryad [30], Spark [54] and PowerGraph [23] are popular frameworks of this kind.

These systems have been built quickly, by taking existing software for single user systems and building components for distribution on top (Figure 1.1a). Sometimes, they are even referred to as "operating systems" for data centres [27, 55]. However, no attempt has been made to integrate awareness of distributed operation into lower levels of the systems software stack.



(a) State-of-the-art. *(b)* DIOS. *(c) Hephaestus.*

Figure 1.1: Abstraction of distribution in state-of-the-art systems compared to DIOS *and Hephaestus. State-of-the-art systems realise distribution via middleware (MW), whereas* DIOS *supports it at the operating system (OS) level. Hephaestus brings Rust to* DIOS *to make writing correct programs easier.*

In my project, I build atop an attempt to do so: DIOS, a distributed OS for data centres that makes the OS kernel aware of its distributed operation. In combination with a high-level language runtime for Rust, this is a powerful model as all layers (application, runtime and OS) are aware of their role in the distributed system allowing information to be shared, performance improved and redundant abstractions simplified.

1.2 Related work

My Hephaestus runtime brings the advantages of Rust to DIOS. This required working with both DIOS and Rust, tying the two projects together.

1.2.1 DIOS

DIOS is a distributed operating system specifically designed for "warehouse-scale" data centres [48]. It abstracts the whole data centre as a single machine, a "warehouse-scale computer" [3]. This simplifies the programmer's mental model by hiding much of the complexity of distribution. However, unlike prior operating systems of the 1980s, information about the distributed state is made available through metadata. This allows programs to optimise their operation when desired, but retains transparency when simple abstractions are preferred. Figures 1.1a and 1.1b show a comparison of the abstractions of distribution between state-of-the-art systems and DIOS.

The DIOS system call API is deliberately minimal, with ten system calls compared to well over 300 in Linux. The system calls are simple, but sufficiently expressive to work as an effective replacement in the restricted domain of data centre operation. Further, since only a few system calls exist, the attack surface is small and more easily verified, improving security.

Only low-level languages which interface directly with the system call API are currently supported on DIOS. This is a major disadvantage, since it prevents the use of high-level languages (e.g. Java [24], Rust and Go [42]), which are easier to use as they offer higher levels of abstraction and memory safety[2]. My project adds support for high-level languages to DIOS. Specifically I implemented a runtime for the memory-safe Rust language.

1.2.2 Rust

Rust is a high-level "systems programming language that runs blazingly fast, prevents almost all crashes, and eliminates data races."[3] It is designed to be suitable for use cases in which C or C++ are the typical language of choice. In order to be a suitable replacement, Rust must offer similar performance, the ability to easily interface with other languages and sometimes allow low-level bit manipulation to directly access hardware. Unlike other low-level systems programming languages, however, Rust offers strict typing and memory safety.

[2] A similar problem was faced in the early days of UNIX [44], only at that time C was considered a "high-level language" when compared to the assembly language it replaced!

[3] http://www.rust-lang.org/; accessed 12/05/2015.

1.3 Challenges

When I started this project, DIOS was at a very early stage of development. Substantial extensions of the DIOS kernel and user-space code were required in order to complete my project. This was challenging since kernel development is notoriously treacherous and requires meticulous attention to detail to avoid errors which can easily bring down a machine. The current DIOS implementation extends the Linux kernel and I consequently had to familiarise myself with the Linux code and development practices in order to successfully work on DIOS.

Moreover, Rust is a very young language and had not reached the first alpha release when I started my project. This meant that each of the Rust compiler, the libraries and the toolchain were highly unstable, with code being moved, APIs changing and even the language's syntax being modified.

As a result, I found compiler bugs, had to change my runtime architecture several times and made significant changes to the DIOS abstractions and system call API. Despite these significant risks my project has been a success: Hephaestus runs multi-tasked, communicating Rust programs on an experimental, non-POSIX operating system. Their performance matches the performance of Rust on Linux and comes close to the performance of hand-tuned C implementations of considerably higher complexity.

Chapter 2

Preparation

2.1 Operating systems

The term operating system (OS) typically refers to a package of software including a kernel, libraries and utility programs. In my project, I am chiefly concerned with low-level user-space and the privileged OS kernel itself.

2.1.1 Kernel

The OS *kernel* is the privileged piece of software responsible for managing hardware and software resources for all programs running on the computer. When the computer boots, the kernel is loaded into memory and executed, only terminating when the computer is powered down.

The kernel runs at the highest privilege level and has complete control over the computer [50, p. 3]. Other programs run in *user-space*, which has a low privilege level and must interact with the kernel in order to perform privileged operations (e.g. I/O or process creation). It is the responsibility of the kernel to ensure the safe sharing of resources between isolated user-space programs.

In addition to managing this safely, the kernel should also aim to maximize the utilization of resources. This is challenging as computer architecture is adopting increasingly parallel designs, causing the degree of concurrency the kernel must manage to increase [8, 9, 15]. This is especially the case in data centre environments where multi-core machines are highly utilized at all times. Indeed, the increasing parallelism and use of on chip networks for communication means that processors themselves are becoming more similar to distributed systems [5].

2.1.2 System calls

System calls (commonly referred to as "syscalls") are the interface between user-space programs and the kernel. The kernel provides a set of functions to perform privileged tasks, which user-space programs can invoke by making a syscall. The application binary interface (ABI) provides a calling convention that defines how arguments and results are passed between the kernel and user-space. The computer's instruction set provides the mechanism for making a system call.

On `x86_64`, the `syscall` instruction is used to transfer control to the kernel, and the System V application binary interface (ABI) [37] calling convention (presented in Table 2.1) is a commonly used standard. The kernel installs a service routine to handle the software interrupt raised by the `syscall` instruction and the user communicates the intended kernel function for the service routine to execute by passing the system call number.

Value	Register
System call number	`rax`
Argument 1	`rdi`
Argument 2	`rsi`
Argument 3	`rdx`
Argument 4	`r10`
Argument 5	`r8`
Argument 6	`r9`
Return value	`rax`

Table 2.1: The System V ABI system call calling convention for `x86_64`.

There exist standard sets of kernel functions which most OSes expose to user-space via system calls. The most common example is POSIX (Portable Operating System Interface) [29] which evolved to standardize the interfaces for both users and developers. However, POSIX is very old, with its key abstractions going back to UNIX [45] of the 1970s, and slow to adapt to modern requirements due to conflicts of interest within standards bodies and the required backwards compatibility.

2.1.3 C standard library

`libc` is the standard library for the C language and is a user-space library which is often released with an operating system. The library includes common algorithms such as string manipulation, sorting and mathematical operations, but also functions

for process management and I/O. The latter two are tightly coupled to the system call interface as they must invoke the kernel to undertake the privileged actions of modifying a process or interacting with hardware.

The majority of user-space programs linking to a `libc`, rather than using operating system's syscall API directly. This layer of abstraction makes programs portable and allows operating systems a degree of implementation and API flexibility with limited effect on user-space programs.

2.2 DIOS

DIOS is a distributed operating system designed for data centre environments. It abstracts all the machines in the data centre as a single "warehouse-scale computer". This approach is known as "single system image" (SSI) [11], although DIOS differs significantly from previous SSI systems. Like distributed OSes of the 1980s (e.g. Amoeba [39], Sprite [40], V [13]), DIOS *transparently* abstracts the distributed system. However, unlike these prior systems, it also exposes sufficient information for programmers to optimize their distributed applications. As a result, the DIOS system call API (Table 2.2) is substantially different from traditional POSIX APIs. The system calls are designed to be more scalable and have semantics better suited to distributed system operation. For example, a DIOS system call can refer to an in-memory object on a remote machine and perform I/O on it.

System Call	Description
`dios_create`	Create a new object.
`dios_lookup`	Obtain a reference by resolving a name.
`dios_delete`	Delete a reference to an object.
`dios_copy`	Delegate a reference to another task.
`dios_begin_read`	Get an input buffer for an object.
`dios_end_read`	Commit an input buffer to an object.
`dios_begin_write`	Get a buffer filled with data read from an object.
`dios_end_write`	Return a read buffer to the kernel.
`dios_run`	Spawn a new task.
`dios_select`	Wait on multiple objects for I/O.

Table 2.2: DIOS *system calls. There are only ten*[1] *compared to over 300 on Linux.*

[1] This number grows to twelve after the changes I make in § 3.5.2.

The DIOS system call API's key differences to POSIX conventions are that it allows for zero-copy I/O, and provides transaction-like "I/O repeat" semantics. Zero-copy I/O means that data are not copied between intermediary buffers, but instead moved directly between the source and destination. Transactional semantics are well-suited to a distributed system, where I/O can be committed in chunks. I/O requests may fail if ordering conditions are not met or concurrent accesses have invalidated an invariant. The API is kept deliberately small as it reduces the attack surface and is easier to verify.

DIOS only supports explicit shared memory across processes ("tasks"), and does not offer threading (which would imply implicit shared memory). This deliberate restriction turns out to be a good fit for Rust, which also does not allow implicit sharing.

2.2.1 Concepts in DIOS

Objects

DIOS uses *objects* to abstract concepts which can be instantiated. Examples include tasks, private memory, shared memory, on-disk blobs and local and network streams. All object types conform to a common I/O interface, which the user accesses via system calls. The POSIX abstraction of a file fulfils a similar role, although object-oriented programming has popularised the concept of objects which are often used in modern operating systems [51]. The four key object categories are tasks, streams, blobs and groups.

Tasks

Similar to a traditional process, a *task* is an object which encapsulates an executing program. Tasks are strictly single-threaded, but concurrency can be exploited on DIOS by spawning multiple tasks. Tasks are managed globally by a cluster scheduler such as Borg [52], Omega [49] or DIOS's own Firmament [47, ch. 5].

Names

Each DIOS object is assigned a unique name to identify it within the DIOS cluster. The name acts as an *identifier capability* as it allows tasks to resolve a name into a reference to an object. Names are 256-bit values within a flat namespace. Names are generated randomly, and since their domain is very large, they are unique in practice.

This comes because the problem of name collision is a generalization of the birthday problem [38]. A formula to approximate the probability of a collision can be derived from the Taylor series expansion of the exponential function. This can be used to approximate the number of names that must be in use for the probability of a collision to be as large as one in a million:

$$\sqrt{-2^{257} \ln |1 - 1 \times 10^{-6}|} \approx 2^{118}$$

The first name collision is expected when there are

$$\sqrt{\frac{\pi}{2} 2^{256}} \approx 2^{128}$$

names in use. These numbers are close to the widely accepted cryptographic security of 2^{128} [2], so in the absence of deliberate leaks, they are safe. The likelihood of a collision is further reduced by the fact that deleted objects release their name for reuse.

Each DIOS kernel instance has a *name table* which contains known mappings between names and objects. To discover unknown mappings, for example for an object on a remote machine, the DIOS kernel communicates with the kernels on other machines using a simple coordination protocol built on reliable UDP using QJump [25].

References

Tasks use *references* to interact with objects via system calls. References are somewhat similar to file descriptors in POSIX, but they are not restricted to local objects. A task can acquire a reference by (i) creating a new object, (ii) looking up an object by its name or (iii) copying a reference between tasks. A reference holds metadata about the object it references, such as the type and proximity of the referred object.

Each task has a *reference table* that maps user-space references to kernel references. A kernel reference contains extra information about the reference that is not exposed to user-space. Since the reference table tracks all references that a task holds, the remaining references can be cleaned up when the task exits. Such housekeeping tasks are an important role of the operating system in order to avoid resource leakage.

A more detailed high-level overview of a DIOS instance is shown in Figure 2.1. The name table is a shared kernel data structure between all tasks on a machine, but the reference table is specific to each task.

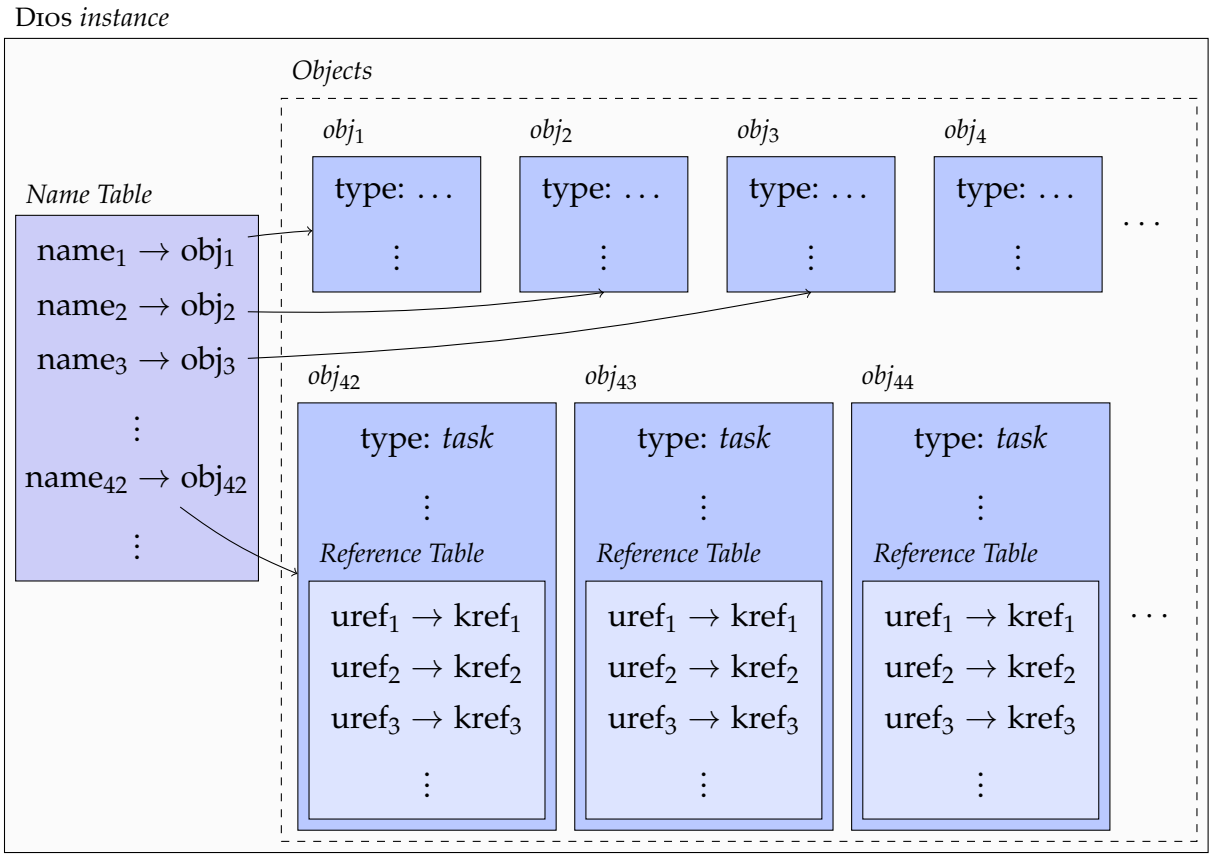


Figure 2.1: Relationships between elements of a DIOS *instance. The name table links names and objects and each task has a reference table to resolve user-space references.*

2.2.2 Implementation

Instead of starting from scratch and implementing the lowest level operations performed by the kernel (e.g. device drivers), DIOS extends and existing OS kernel. This *host kernel* supplies tried and tested low-level functionality and DIOS extends it with its abstractions. DIOS currently supports the Linux kernel as its host kernel, but it is designed to be independent of any particular host kernel, abstracting host kernel details through the DIOS Adaptation Layer (DAL). DIOS also provides a user-space C library, `dlibc`, which implements a subset of the C standard library as well as some DIOS-specific utility functions. Figure 2.2 illustrates these layers of abstraction and their relationships.

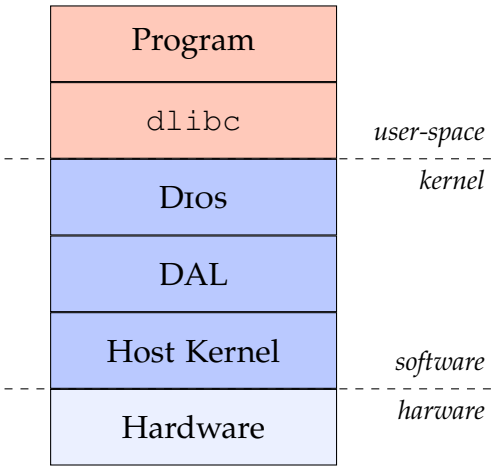


Figure 2.2: Layers of abstraction in Dios.

2.3 Compilers

Hephaestus is a runtime for the Rust programming language. Its implementation required various modifications of the Rust compiler since Rust does not dynamically load a runtime library. Compilers are usually implemented as a series of phases, each of which transforms the program from one representation to the next. The transformations generally remove information or structure from the original source code, until a flat sequence of machine instructions in the form of an executable is produced. A high-level overview of the compilation process is shown in Figure 2.3.

Lexing Converts the source code into tokens. A token represents a keyword, literal, type etc., but ignores insignificant parts of the source code such as white space and comments.

Parsing Converts the token stream into an abstract syntax tree (AST) following a grammar. The AST nodes represent operations, or values (if the node is a leaf), and the branches link to the operands. This stage identifies the structure and semantics of the program and the relationship between the components. High-level, language-specific optimisations can be made on the AST.

Translating Converts the AST into a standard format independent of both the source language and the target platform. This allows generic analyses and optimisations to take place, which can be reused between compilers.

Code Generation Converts the program into machine code. This requires in-depth knowledge of the target instruction set to choose instructions and make best use

of the registers. As the target architecture is known at this stage, further low-level optimisations can be made to produce an optimised binary.

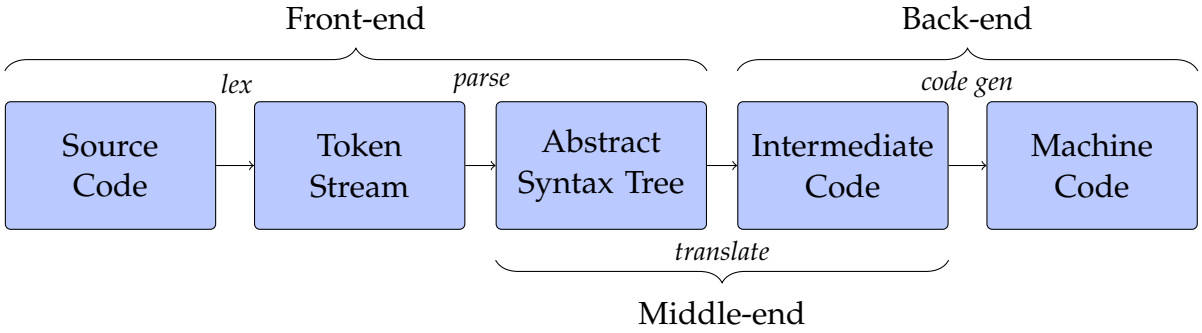


Figure 2.3: The phases of compilation.

LLVM [34] is a modular compiler infrastructure comprising of reusable components for program optimisation and code generation. LLVM intermediate representation (IR) is the intermediate code it uses to represent programs on which it applies its optimisations. There are components to transform the LLVM IR into machine code for many target architectures. Using LLVM greatly simplifies the process of building a compiler, allowing developers to focus on language-specific problems, such as source code to LLVM IR translation and AST-level optimisations. The Rust compiler uses LLVM to do its code generation for this reason.

2.4 Rust

Rust is a modern systems programming language that aims to be fast and safe. It is *safe* in the sense that invalid memory accesses are prevented and data races are eliminated. As it is a systems language that aims to have minimal runtime overhead, its safety guarantees are largely achieved through static analysis at compile time, with help from an expressive type system. A brief comparison of Rust to other commonly used languages is shown in Table 2.3.

Rust is a compiled language and hence follows the compile-link-execute cycle. *Compilation* converts source code into binary representations which have unresolved dependencies on other binaries. *Linking* takes in a set of compiled binaries and resolves the dependencies between them, producing an executable binary. The *executable* binary can then be run on the target platform to complete its task.

Language	Compiled	Memory Management	Predictable	Safe	Static types
C/C++	✓	None†	✓	✗	✓
Java	✗	Garbage collection	✗	✓	✓
OCaml	✓	Garbage collection	✗	✓	✓
Go	✓	Garbage collection	✗	✓	✓
Python	✗	Garbage collection	✗	✓	✗
Rust	✓	Ownership	✓	✓	✓

Table 2.3: Comparison of the features of commonly used languages. †C and C++ do not natively offer memory management, but the standard library offers reference counting implementations.

2.4.1 Memory safety

The aim of memory safety is to prevent access to invalid memory. Invalid accesses include access to unallocated or uninitialized memory, as well as accesses to valid memory outside an allocation (e.g. buffer overrun).

Rust combats logically invalid accesses by ensuring the appropriate checks, for example bounds checking on array indices, are made before accessing memory. Arbitrary pointer initialization or pointer arithmetic, which could circumvent these safety checks, are not permitted. To ensure that all memory is initialized, there are compile-time checks to make sure variables, both on the stack and heap, are correctly initialized before they are used. The remaining problem is to ensure that a valid memory reference does not come to reference invalid memory by way of another reference invaliding the memory.

One way in which memory can be invalidated if by deallocation. The obvious fix to this would be to never free memory after it has been allocated, but this is clearly not practical in the face of finite memory. Instead, memory must be reclaimed, but we must prevent deallocation while the memory is still in use. Reference counting and garbage collection are key techniques traditionally used to achieve this.

Reference Counting: The aim is to ensure memory is not deallocated while it is still referenced. Reference counting ensures this by keeping track of the number of references to each memory allocation. Only when this count reaches zero, is it safe to free the memory. The major flaw with the straightforward application of this technique is introduced by cycles in the reference graph, which cause all elements in the cycle to be permanently referenced.

Garbage Collection: Instead of counting references as they are made and removed, garbage collection computes which references are still valid. By starting with

a base set of references on the stack and following them and any references in the referred allocations, the transitive closure of referenced memory can be computed. Any memory that is not included in this closure is no longer being referenced and so can safely be freed. The main disadvantage of garbage collection is the cost of computing the transitive closure of valid references. This has been greatly improved from the days of "stop-the-world" collection by incremental techniques such as generational collection [1], and concurrent collectors [19].

Both of these techniques suffer runtime overheads which are in many cases unnecessary. The overheads are especially sever in data centre workloads with large heaps and object counts [22, 35]. Consider an allocation which is only used within a function. There is no need for runtime overhead as its safety can be verified at compile time. A novel concept explored by Rust is the notion of memory *ownership*[2], which allows for static analysis of memory safety.

Ownership: Explicit ownership tacking assigns each memory allocation an owning reference, and deallocates the memory when it goes out of scope. There can only be one owning reference at any time but, to allow for more flexibility, ownership can be *moved* and *borrowed*. Borrowing temporarily transfers ownership from one reference to another, while moving ownership is permanent. The *borrow checker* performs static analysis to ensure the consistency of ownership.

There are still times where ownership and static analysis techniques are too rigid, for example multiple independent references to the same memory location must be maintained. In these situations, Rust can fall back on explicit reference counting, which is provided in the standard library.

Unsafe Code

The strict rules about memory ownership and pointers that Rust imposes can be too restrictive, especially when writing low-level systems applications. Interacting with hardware via memory addresses or just interfacing with a C library is difficult without the ability or perform pointer arithmetic. Rust allows blocks of code to be explicitly labelled as "unsafe", permitting them to perform unsafe memory operations such as manipulation of raw pointers. When using an unsafe block, the flexibility comes at the cost of the absence of compiler checks – it is down to the programmer to ensure the code is safe. However, Bugs are easier to locate as the programmer is required to explicitly indicate where they intend to side-step the safety guarantees.

[2] https://doc.rust-lang.org/book/ownership.html; accessed 12/05/2015.

2.4.2 Nomenclature

Rust introduces its own names for some familiar concepts. I give a brief overview of the relevant nomenclature below.

Box[3]**:** Boxes provide ownership for heap allocations that are freed when they go out of scope. They are comparable to the `std::unique_ptr` concept from the C++11 standard library [32].

Item[4]**:** Some concepts in the Rust language are referred to as "items". These include functions, type definitions, structures and modules.

Module[5]**:** Zero or more items can be logically combined into "modules". Modules are useful for organising code. They are similar to modules in Python.

Crate[6]**:** Rust's compilation unit is called a "crate" which contains a tree of nested modules. Unlike other languages, where the compilation unit is restricted to a single file, crates can span multiple files. Each compilation takes a crate in source form and produces the binary form of the crate.

2.5 Development

2.5.1 Programming languages

Since I am working with three existing projects, my implementation language choices were effectively dictated by each project. Use of other languages would have added unnecessary complexity when interfacing with the projects.

C90

Both Dios and the Linux kernel are written in the C90 variant [31] of the C programming language. C today is the de facto standard language for low level systems programming despite many of its short comings. Since C can be built without dependencies on a runtime or a standard library, it is a good choice of language for the standard library for Dios-hosted C programs. I had previous experience of C from the Part 1B *Programming in C and C++* course and from various personal projects.

[3] http://doc.rust-lang.org/std/boxed/; accessed 12/05/2015.
[4] http://doc.rust-lang.org/reference.html#items; accessed 12/05/2015.
[5] http://doc.rust-lang.org/reference.html#modules; accessed 12/05/2015.
[6] http://doc.rust-lang.org/reference.html#crates-and-source-files; accessed 12/05/2015.

Assembly Language

There are some situations, such as calling specific instructions to invoke system calls, in which even C is too high-level. When this need arose I had to use assembly language. Because assembly language is essentially "readable" machine code without types of scope, it is difficult to write and maintain, so strove to use it sparingly. DIOS currently only targets the `x86_64` architecture, so I used `x86_64` assembly language when required.

Rust

The Rust project, including the compiler, is written in the Rust language itself. This was especially challenging as Rust itself kept changing on conceptual and syntactical levels. I had no experience of using Rust before this project and had to learn it "on-the-go". Rust is a multi-paradigm language, predominantly imperative and object-oriented with functional aspects also incorporated. I am familiar with each of these paradigms, which were explored in the Part 1B course *Concepts in Programming Languages*. Understanding of these concepts meant that learning the basics of the language was reasonably straight forward. To become proficient I has to acquire experience and in-depth knowledge of the standard library.

Shell

For the construction of simple tools and utilities, I used shell scripts and standard UNIX utilities. The Part 1B course *Unix Tools* presented an introduction to the capabilities of the shell, as well as techniques for using it effectively, and I built upon this knowledge.

2.5.2 Development environment

Version control

Both the DIOS and Rust projects use Git[7] for their version control. Git was developed for use with the Linux kernel code base and is designed for distributed collaboration of many developers. Creating patches that encapsulate modifications is part of the standard work flow, and turned out to be very useful for upstreaming my changes to new versions of the projects.

[7] `http://git-scm.com/`; accessed 12/05/2015.

Backup

Backups are vital in order to recover from unexpected events which might have rendered my work inaccessible or corrupted. This could have been caused by software or hardware failure, or indeed by an excitable programmer making changes they later regret. This last case is covered by using a source control manager, in this case Git, which allows checkpoints to be made. In the case of software or hardware failure, copies of the work must be made in different places.

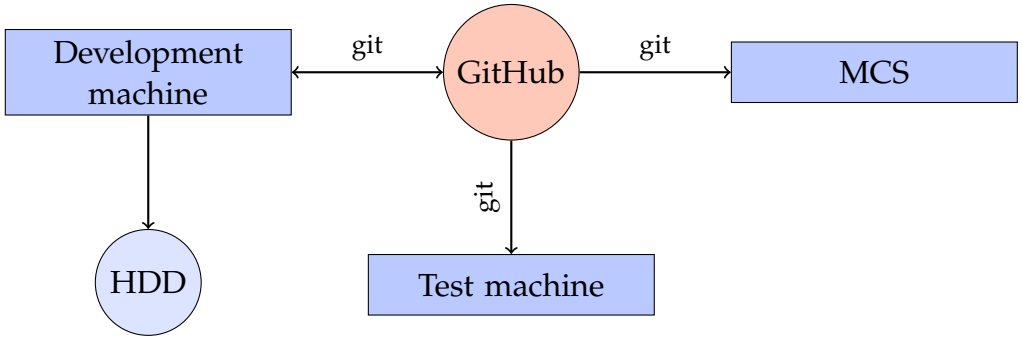


Figure 2.4: The backup strategy I used for my project. Clones were made using git, but the backup on the external hard drive (HDD) just contained a snapshot of the files.

I hosted my Git repositories remotely on GitHub[8] and cloned them into local working repositories. Figure 2.4 illustrates my backup strategy. At all times, I had one clone on my main development machine, from which I pushed my commits to GitHub at the end of each day. At frequent intervals, I also pulled changes from the GitHub repository into a clone stored in the MCS. When running tests, I cloned the repository onto the test machine giving another copy. Finally, I made regular backups to an external hard drive from my development machine.

2.5.3 Development model

I adopted an agile approach to the implementation, specifically using a feature-driven development style. This is an iterative process in where a feature is first identified, then it is planned, designed and built before moving on to the next feature. This more adaptable style of development was the most suitable for Hephaestus, as the code bases I was working with changed significantly without much notice. If I had extensively planned out multiple dependant features before implementing any, I might have been caught out by unexpected upstream changes.

[8] https://github.com/; accessed 12/05/2015.

To implement features, I worked vertically through the layers of abstraction in Dios, starting at the lowest level and working up to the highest level. Working bottom-to-top made all the dependencies available at each stage, so I was be able to successfully build the code and run functional tests. The four main levels I had to work at were:

1. **Linux** – I first ensured that the feature could be implemented with the Linux host kernel features. If this was not possible, I had to make the required modifications to make it possible.
2. **DIOS** – Given the host kernel offered the required functionality, I had to expose it to user-space via Dios. This involved modifications to systems calls and to the implementation of object drivers.
3. **dlibc** – In order to more elegantly expose the required functionality, I often had to add a wrapper around the kernel interaction to `dlibc`. This provides a simplified interface consisting of functions, as opposed to system calls.
4. **Rust** – Finally the feature had to be implemented in Rust by using the functionality that is now available in the lower levels. The functions from `dlibc` had to be imported and given a Rust interface so that all code, not just unsafe code, can use it.

2.6 Requirements analysis

The core aim of this project is to allow programs written in Rust to run on Dios. As extensions, I investigated the integration of Dios's task spawning mechanism with Rust and made improvements to the Dios implementation. The key features I will be implementing in this project are summarized in Table 2.4.

Feature	Priority	Risk	Difficulty
Printing panic errors to console.	High	Medium	Medium
Printing to console.	High	Medium	Medium
Dynamic memory allocation in user-space.	High	High	High
Spawning tasks from Rust.	Medium	Medium	High
Channels to allow tasks to communicate.	Medium	Low	Medium
Improved resource management in Dios.	Medium	Medium	High

Table 2.4: Features to be implemented for the Hephaestus project.

Chapter 3

Implementation

This chapter describes the implementation of the Hephaestus runtime. Firstly, I configured the Rust build for DIOS (see § 3.1), which required the implementation of a dynamic memory allocator (see § 3.2). I then extended Hephaestus to support DIOS tasks (see § 3.3) which communicate via channels (see § 3.4). Finally, I describe the further improvements I made to DIOS (see § 3.5).

3.1 Rust configuration

In order to make Rust compatible with DIOS, I had to make some changes to it, which I describe in this section.

3.1.1 Conditional compilation

Rust allows conditional compilation of parts of the source code through the use of attributes. Attributes can be applied to any item (function, struct, type, etc.), with the `cfg`[1] attribute providing support for conditional compilation. `cfg` evaluates boolean expressions and if the expression evaluates to true, the item is included in the compilation, otherwise the item is excluded (much like `#if` in the C pre-processor). The configurations that must be defined are shown in Table 3.1.

[1] https://doc.rust-lang.org/reference.html#conditional-compilation; accessed 12/05/2015.

Configuration	Description
`target_arch`	Target CPU architecture.
`target_endian`	Endianness of the target CPU.
`target_family`	Operating system family of the target.
`unix` | `windows`	Shorthand for the above.
`target_os`	Operating system of the target.
`target_word_size_os`	Target word size in bits.

Table 3.1: Rust's required configuration values

The Rust compiler can target a number of architectures and operating systems, combinations of which are identified using the "target triple" notation[2]. The compiler populates the configurations based on target triple provided.

For Dios, I created the new target triple "`x86_64-linux-dios`" representing the `x86_64` architecture, Linux as the host kernel and Dios as the operating system. The choice was inspired by Android's use of `arm-linux-androideabi`. The configuration corresponding to the Dios target triple is based on the `x86_64-unknown-linux-gnu` configuration with the `target_os` set to `"dios"`. Rust categorises operating systems into one of two families: POSIX or Windows. Dios fits into neither of these, so I added a new family called `"dios"` with the shorthand configuration `dios`. Listing 3.1 contains an example of its use. This change gives the ability to conditionally select Dios-specific items to be included in compilation, or modify behaviour based on the target OS, while keeping the code backwards-compatible with the upstream release.

3.1.2 Standard library

Rust comes with a feature-rich standard library comprised of many crates (libraries). The `libstd` crate is of particular significance as it is automatically imported and many of its symbols are automatically brought into the global name space. While it is possible to prevent the automatic import of `libstd` by using the crate-level attribute `no_std`, this leaves the user with a very impoverished programming environment. Supporting `libstd` in Hephaestus was therefore a high priority.

[2] http://clang.llvm.org/docs/CrossCompilation.html#target-triple; accessed 12/05/2015.

```
1  #[cfg(dios)]
2  fn is_this_dios() {
3    println!("This is DIOS");
4  }
5
6  #[cfg(not(dios))]
7  fn is_this_dios() {
8    println!("This is *not* DIOS");
9  }
10
11 fn main() {
12   is_this_dios();
13 }
```

Listing 3.1: An example of Rust's conditional compilation. The `is_this_dios` function is the item being conditionally compiled. The configuration being tested is `dios`, which is shorthand for `target_family = "dios"`.

liblibc

The `liblibc` crate handles the interface with native C libraries. It exposes `libc` to Rust, as well as `libposix` and Windows APIs for the appropriate targets. It uses Rust's foreign function interface[3] (FFI) to declare symbols which can be resolved at link time. Types, constants and function signatures must be transcribed. This introduces some redundancy, but is required in order for Rust's type checker and code generator to have all the information they need.

To adapt `liblibc` for Dios, I use the `cfg` attribute to exclude unavailable symbols – such as the Windows API – from the build. I also add the necessary boilerplate code to expose the functions from `dlibc` (the Dios-specific standard library that I developed) to Rust. This means Dios system calls can now be invoked from other crates in the standard library which is required to implement some features.

libstd

Part of the role of `libstd` is to abstract away the differences between different target platforms. For example, thread-local storage is done very differently on Windows and Linux, but `libstd` provides a common API. This abstraction was designed for portability and proved extremely effective when it came to porting onto Dios.

[3] https://doc.rust-lang.org/book/ffi.html; accessed 12/05/2015.

Threads: Threads, thread local storage, mutexes and condition variables are of little value on Dios, where all tasks are single threaded. Other parts of the standard library make use of the these features, so they needed to be implemented, even as stubs. I implemented thread-related functionality to raise an error (as it is unsupported) and thread-local storage just uses memory on the heap (as there is no chance of interference from other threads).

`stdio`: The standard input, output and error streams generally output to the console, although they can be redirected to files on most platforms. Dios has a console object which supports output, but does not implement input as it is not designed for interactive use. An instance of the console object is used to implement the standard output and error streams, but an error is returned for all operations on the standard input stream.

File system: Dios does not have a normal hierarchical file system. Instead, this would be a service provided by a user-space program. A file system for Dios has not yet been built so for now these functions are stubbed out, returning an error if called.

The `rt` (runtime) module in `libstd` contains the runtime components. This handles important actions such as storing the command line arguments, as well as handling stack unwinds and backtraces. If a program panics, it is the `rt` module which manages the situation cleanly. Hence, `rt` cannot depend on anything outside of its module, as the external dependency might be panicking. Useful error messages are printed to the screen to aid debugging, and `rt` needs to implement the printing functionality itself, independently of `stdio`. Since the logic is almost exactly the same for the two cases, I duplicated the `stdio` code with a few minor modifications.

3.1.3 Alpha release

Midway through making my modifications to Rust, it was announced that the first alpha release of Rust would arrive in the near future. This prompted a wave of significant modifications by the Rust development community, from relocating or deleting code to refining APIs. One particularly affected was operating system integration, which happened to be where the majority of my work was taking place.

During this period, there was far too much instability to be able to make any progress and I was forced to hold back until everything stabilized again. In order to make use of this time, I instead worked on the modifications to Dios that were required. After the Rust interfaces had stabilized for the alpha release, I picked up the new code and forward-ported my changes.

3.2 Dynamic memory allocation

Dynamic memory allows programs to adapt to workloads of varying sizes. Without it, programs would be very limited since large arrays and linked lists would be difficult to implement. This makes dynamic memory allocation and important feature to implement in Hephaestus.

3.2.1 `liballoc`

Rust's dynamic memory allocation is handled by the `liballoc` crate of the standard library. The crate offers some high-level abstractions, such as boxes and reference counted allocations, as well as low level utilities for directly managing allocations on the heap. The low-level functions listed in Table 3.2 are used to implement the high-level abstractions.

Function	Description
`allocate`	Allocate aligned bytes on the heap.
`reallocate`	Resize an allocation.
`reallocate_inplace`	Resize an allocation without moving it.
`deallocate`	Deallocate some memory .

Table 3.2: The heap allocation functions of `liballoc`.

The implementation of the low-level functions is configurable when building Rust. By default, Rust uses `jemalloc` [20], which is included in the source tree. As discussed later, porting `jemalloc` to Dios would prove not only difficult but also ineffective, so instead I disabled the use of `jemalloc` by passing the `--disable-jemalloc` flag to the Rust configure script. With `jemalloc` disabled, `liballoc` uses a platform-dependent implementation, with Unicies calling `libc`'s memory allocation functions and Windows calling equivalent functions from the Windows API. For Dios, I also call `libc`'s memory allocation functions.

After disabling the use of `jemalloc`, I encountered a severe problem when building the Rust compiler. The host compiler, which is used to build the new version of the Rust compiler, would crash and report an illegal instruction error. I reported this bug[4] to the Rust developers and performed a binary search on the commits to discover which commit introduced the problem. The identified commit seemed fairly innocuous, and neither myself nor the Rust developers understood how it could introduce the problem. However, after reverting this commit, the built error disappeared and I used this as a temporary fix. After I reported this bug, others

reported they were also experiencing the problem on FreeBSD and OpenBSD. Later in the project, new code depended upon the commit that I had reverted so the commit needed to be reapplied. Even though the bug remains undiagnosed, there was no compilation error after reapplying the commit. The suspicion in the Rust community is that the bug was present in the first-stage compiler of the bootstrap process, which interacted poorly with some later changes. After the first stage compiler snapshot was updated and the problem appears to have disappeared.

3.2.2 `malloc`

`malloc` is the dynamic memory allocation function for the C standard library. In fact, there is a family of functions related to dynamic memory allocation. These functions are flexible, allowing arbitrarily sized areas of memory to be allocated, resized and deallocated (Table 3.3).

Function	Description
`malloc`	Allocate uninitialized memory.
`calloc`	Allocate zeroed memory.
`realloc`	Expand or shrink an area of memory.
`aligned_alloc`	Allocate aligned memory.
`free`	Deallocate an area of memory.

Table 3.3: Dynamic memory management functions from the C11 standard library [33].

The memory is allocated in a region of the process address space called the "heap". When a process is first started, its initial mapped range of virtual memory is likely too small to match the memory requirements of the application. The dynamic memory management functions interact with the operating system in order to grow and shrink the process's address space as required. The POSIX standard prescribes the `mmap` (memory map) and `munmap` (memory unmap) system calls to create and free mappings in the process's virtual address space.

Since DIOS does not conform to the POSIX standard, neither `mmap` nor `munmap` are supported system calls. Instead, memory is mapped to a process by creating a new private memory object. Calling `dios_begin_write` on this object returns an input buffer that corresponds to the newly mapped memory.

[4] https://github.com/rust-lang/rust/issues/21526; accessed 12/05/2015.

A simple allocator

My first attempt at building an allocator took a very naïve approach. However, it was quick to implement and worked correctly.

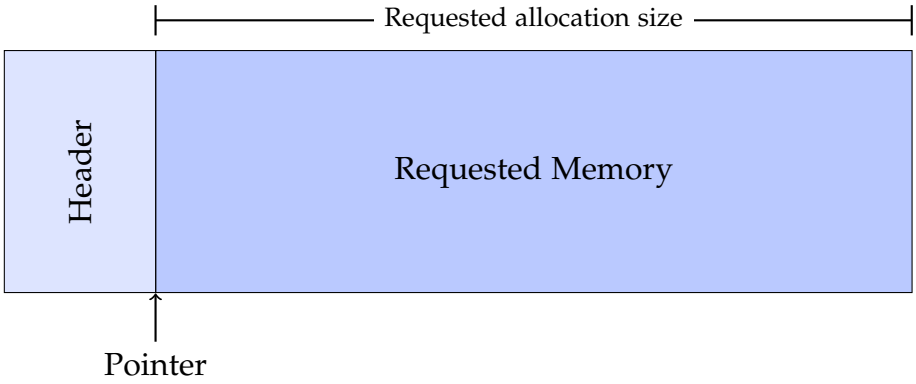


Figure 3.1: Memory layout of a block allocated by the simple allocator.

Allocation. Every single memory allocation creates a new private memory object. The private memory object is requested to be slightly larger than the allocation that the user asked for, so that a header can be added, which holds information about the private memory object. The pointer returned by the allocator points just after this header. Figure 3.1 illustrates the memory layout.

Freeing. When freeing memory, the header is found in the memory just before the pointer provided. The information held in the header about the private memory object is then used to delete it. In order for this to be safe, it is required that the pointer provided is one that was previously returned by a call to the allocation function. This constraint is no different from the standard interface for `free`.

Performance. This simple implementation has suboptimal performance with respect to time and memory efficiency:

- Each allocation and free makes a system call, meaning that there is a large overhead associated with these operations.
- Each allocation creates its own private memory object, resulting in potentially high fragmentation. DIOS private memory objects have a granularity of one page (typically 4096 bytes on `x86_64`), so an allocation of anything other than a multiple of the page size will be wasting the difference. The exact value must also take into account the size of the header.

dlmalloc

Building an effective general-purpose dynamic memory allocator is a complex task. I therefore decided to adapt an existing implementation for DIOS, rather than attempting to design one myself. There exist many algorithms, and Table 3.4 lists those in common use.

Allocator	Description
`dlmalloc`[5]	General purpose `malloc` developed by Doug Lee starting in 1987.
`ptmalloc2`	Derived from `dlmalloc` with improved threading support.
`tcmalloc`[6]	Designed for threading performance and low memory overhead.
`jemalloc`	Focuses on low fragmentation and threading performance.

Table 3.4: Popular `malloc` implementations.

From these, I chose `dlmalloc`. It was designed to minimize the space and time required for allocations, although it focusses on single-threaded programs. This is no problem for DIOS, since DIOS tasks are themselves single-threaded. `dlmalloc` is also designed to be portable and has minimal dependences. Other implementations tend to focus on improving multi-threaded performance, and so introduce more dependencies and complexity with little or no benefit for DIOS.

Emulating mmap

The major dependencies that `dlmalloc` does have are on `mmap` and `munmap`, which I had to emulate in user-space on DIOS. The full specification for `mmap` supports the mapping of files and devices into memory with many flags for different configurations. `dlmalloc` only uses a very small subset of these and even abstracts the use into a macro of just one parameter, the requested size of the memory region (Listing 3.2). The `munmap` interface is left unchanged as it is already simple. It takes an address, `a`, and a size, `s`, and unmaps all mappings which occur in the range `[a, a+s]`.

The difficulty for DIOS is introduced by unmapping from a given address. The address lies within the input buffer of a private memory object, and to unmap the memory, the private memory object must be deleted. This means that a mechanism is required to translate the memory address to the private memory object it is associated with. In fact, it requires a mechanism for finding *all* private memory objects in a range of addresses. To do the translation, the relation from address to object needs to be stored

[5] http://g.oswego.edu/dl/html/malloc.html; accessed 12/05/2015.
[6] http://goog-perftools.sourceforge.net/doc/tcmalloc.html; accessed 12/05/2015.

```
1 #define MMAP_PROT               (PROT_READ|PROT_WRITE)
2 #define MMAP_FLAGS              (MAP_PRIVATE|MAP_ANONYMOUS)
3
4 #define MMAP_DEFAULT(s)         mmap(0, (s), MMAP_PROT, MMAP_FLAGS, -1, 0)
5 #define MUNMAP_DEFAULT(a, s)    munmap((a), (s))
```

Listing 3.2: The `mmap` abstraction used by `dlmalloc`. The only parameter is the size of the memory region to map. The call to `mmap` requests private, anonymous memory region at any location with read and write permissions. `munmap` is left unchanged.

in a data structure. Unfortunately, `malloc` is not yet available so this data structure must manage its own memory directly.

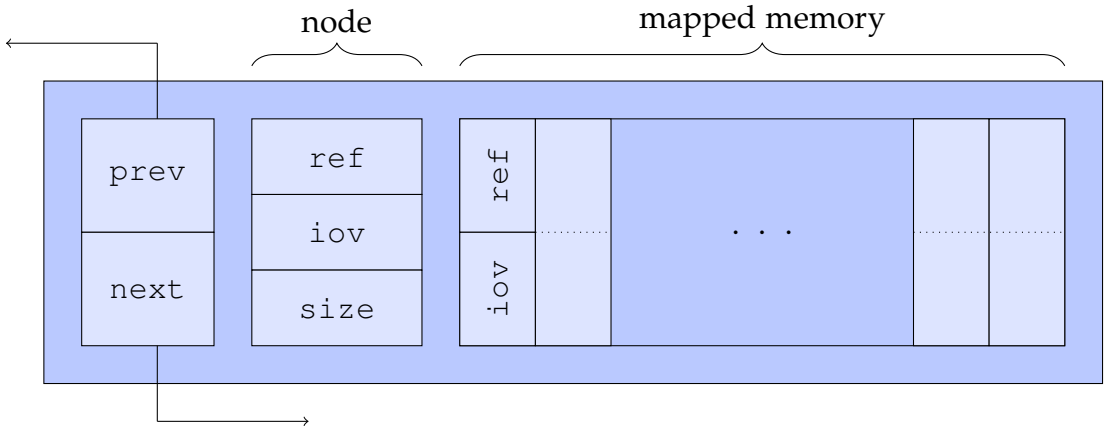


Figure 3.2: Structure of a node from the `mmap` data structure.

The data structure I chose is a dynamically sized list of private memory object references and iovecs (I/O vectors, the buffer returned from `dios_begin_write`) ordered by the virtual address of the memory that they map into the process address space. To allow for dynamic sizing, this is implemented as a linked list of partitions of the list. Figure 3.2 shows the structure of the nodes in the linked list. Since memory for each of these nodes comes from DIOS memory objects, they must track their own private memory object, as well as those being added to the list.

Inserting into the list using Algorithm 1 preserves the order of the list and has $\mathcal{O}(n)$ time complexity. Deleting from the list using Algorithm 2 also preserves the order and again has $\mathcal{O}(n)$ time complexity. The implementation of these algorithms must manage iteration across the nodes in the list as well as adding and removing the nodes.

Improvements. This implementation has $\mathcal{O}(n)$ time complexity. `dlmalloc` aims to reduce the number of expensive calls it makes to `mmap` and `munmap` by mapping

Algorithm 1: Insert a block into the `mmap` list.

input : A new block which does not overlap with any other block. This will be the case as the memory-mapped areas cannot be overlapping.

```
1  ensure space for block at end of list
2  if list is empty then
3  |  insert new_block at start
4  else
5  |  block ← last valid block in list
   |  invariant: successor of block is an empty cell
6  |  while block is valid and new_block should be before block do
7  |  |  prev_block ← predecessor of block
8  |  |  swap block with successor
9  |  |  block ← prev_block
10 |  end
11 |  move new_block into empty cell successor of block
12 end
```

Algorithm 2: Delete a range from the `mmap` list.

input: A range of addresses which may span multiple blocks, all of which must be deleted.

```
1  if list is not empty then
2  |  block ← first valid block in list
3  |  while block not in range do
4  |  |  block ←successor of block
5  |  end
6  |  empty_cell ← cell of block
7  |  repeat
8  |  |  block ←successor of block
9  |  until block not in range;
10 |  repeat
11 |  |  move block into empty_cell
12 |  |  empty_cell ←successor of empty_cell
13 |  |  block ←successor of block
14 |  until end of list;
15 end
```

large chunks of memory to create memory pools, which are used to service allocation requests. The cost of mapping a large amount of memory is amortized across the allocations it serves. This means that $\mathcal{O}(n)$ complexity should be acceptable as the list will not grow very large and will only be called infrequently. In the event that time complexity does become an issue there are some options:

Skip-list: Instead of iterating through each private memory object, first identify the partition it is contained in. This can be done as the list is ordered, so each partition has an address range to compare against. This will still be $\mathcal{O}(n)$, but the constant factor will be much lower.

Red-black tree: Instead of storing the items in a flat array, store them in a balanced binary tree. Insertion and deletion from a red-black tree has complexity $\mathcal{O}(\lg n)$. `munmap` would be $\mathcal{O}(m \lg n)$, where m is the number of private memory objects in the range being unmapped.

3.3 Tasks

Tasks are the unit of execution on Dios, similar to threads and processes in traditional operating systems. In order to execute concurrent workloads tasks must be *spawned*. This an important and common operation. There is a fairly substantial process involved in spawning a new task, most of which is repetitious boiler plate. It would be nice if our Rust programs could simply request to run a function in a new task.

3.3.1 Threads in Rust

Concurrency model

The Rust concurrency model has changed significantly over the duration of my project. When I started, Rust had the concept of generalized "tasks" as units of execution and even provided multiple implementations: one for kernel threads and one for green (user-space) threads. It was possible to provide a custom implementation for tasks by creating a new runtime to implement the necessary interfaces. However, as Rust moved into its alpha releases, this generalized concurrency model was dropped and only the kernel threading model was retained.

The original design would have integrated very nicely with Dios tasks as there was no reliance on shared resources. Instead, all transfer and communication had to be explicit. In the new model, it is assumed that shared memory is available and pointers

can be passed between threads. This is a problem as Dios tasks do not meet this requirement, and therefore are no longer compatible.

However, in the following, I show how I implemented a task API for Dios in Rust's standard library, offering compatible functionality to what the original Rust facilities would have offered.

Thread API

In Rust, threads are created with the `thread::spawn`[7] function, which takes a function-like parameter and spawns a new thread to execute it. The most prevalent function-like types are functions and lambdas. The ability to pass lambdas makes the API very neat, as the captured environment of the lambda is made available to the new thread with no extra code.

```
1 pub fn spawn<F>(f: F) -> JoinHandle where F: FnOnce(), F: Send + 'static
```

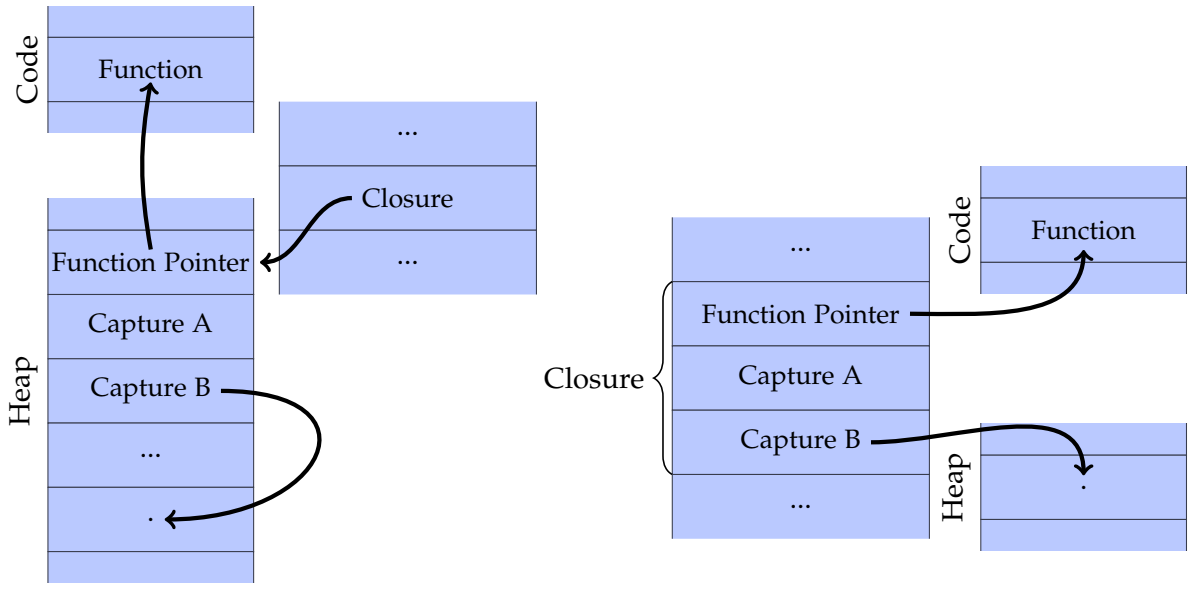
Listing 3.3: Signature of the `thread::spawn` function. It takes an argument of a type which implements the trait `FnOnce` which signifies that it can be invoked like a function at most one time. The argument type must also implement the `Send` trait, meaning that it can be transferred across thread boundaries. Finally, the argument must have a `static` lifetime, meaning it will be valid in all scopes. It returns a `JoinHandle`, which can be used to wait for the thread to exit.

3.3.2 Unboxed closures

When I started this project, Rust only had support for *boxed* closures, that is closures allocated on the heap (Figure 3.3a). The structure of the closure was defined as a function pointer followed by all captured variables. This caused some debate in the Rust community as it *required* the use of the heap. Use of the heap comes with the overhead of allocation which has an impact on performance, and in many cases can be avoided by allocating on the stack instead.

The concept of *unboxed* closures (Figure 3.3b) was introduced to address these points and allow for allocation on the stack. At the same time, the definition of a closure was relaxed to any type which implements the `Fn`, `FnOnce` or `FnMut` trait. The structure of the closure is no longer strictly defined, but depends on the specific type. To invoke

[7] http://doc.rust-lang.org/std/thread/index.html; accessed 12/05/2015.



(a) Boxed closure. *(b) Unboxed closure.*

Figure 3.3: A closure is either boxed (heap allocated) or unboxed (stack allocated). Captured variables can be values (Capture A) or pointers (Capture B). The structure of a closure is not defined and could be any type that implements the `Fn`, `FnOnce` *or* `FnMut` *trait.*

the closure, the `call` function of the trait is executed. The implementation of `call` is aware of the specific type and so knows the structure of the closure.

If the Rust threading API were to be implemented for DIOS, the captured environment would need to be copied to the new task because there is no shared memory. All captured pointers would need to have their target data copied too. This process would need to be recursively repeated until all the required memory had been copied. Another complication is introduced by the new trait based closures. Since the structure of the closure is not known, the function pointer is not known. This means the entry point for the new task is not known, and means a DIOS implementation of the API is infeasible.

3.3.3 DIOS tasks for Rust

Having seen that the semantics of DIOS tasks and Rust threads differ sufficiently to make the implementation of the latter with the former unsuitable, I decided to create a

separate API for Dios tasks. This API is syntactically similar, but due to the problems associated with closures, I was forced to only accept functions as the target to run in the new task. Passing a function works by passing a pointer to the function, which means that the entry point of the new task is known. However, there is no captured environment, and hence all transfer of data needs to be explicit. This makes the source more verbose, but could be addressed by better library wrappers in the future.

```
1 pub fn spawn_task(f: fn()) -> io::Result<()>;
```

Listing 3.4: Signature of the `dios::spawn_thread` function. It takes a function which takes no arguments, produces no result and returns an indication of success. The function argument itself must have a strict type as type information cannot be communicated to the new task.

Entry point

When a Rust program starts, it sets up some runtime state before invoking the main function. When spawning a task, this process is identical, except that we must invoke the task entry function instead of the main function. In order for this to be possible, the entry function must be communicated to the new task. As Rust does not have reflection, the only way to identify a function at runtime is by a pointer to its code.

Position independent code (PIC) complicates this matter, as the program text is loaded at a random address each time it is run. Hence, the address of the function will differ between runs. However, when using PIC, the *offset* between functions is consistent across runs so that relative jumps are still valid without needing to rewrite the program text. This means that I can identify the entry function via an offset from a base function.

This technique, however, is vulnerable to a potential security exploit because the spawner can choose an arbitrary value for the entry instruction pointer. If the attacker can find a sequence of bytes in the program binary which, when interpreted as instructions, allow the attacker to achieve something malicious, then they have an attack on the system. This problem can be fixed by predefining a "white-list" of entry points which the programmer intended to be used. These entry points can be enumerated and referenced by their enumeration value, with any invalid value inducing a panic. This prevents the attack from being possible by validating the untrustworthy input.

DIOS program brands

When spawning a task with `dios_run`, extra state is added to the host kernel's process data structures. This allows the new task to access DIOS-specific information and grants it the ability to make DIOS system calls. The information provided to the process includes the DIOS name of the executable that is being run. If a task wants to spawn another version of itself, it needs to know this information. However, this information is only available if the task has been spawned by `dios_run`. DIOS uses the BusyBox[8] shell, which is responsible for using `dios_run` when appropriate. This means that there must be a way to identify which programs should be started with `dios_run` and which should use legacy mechanisms such as `fork+exec`.

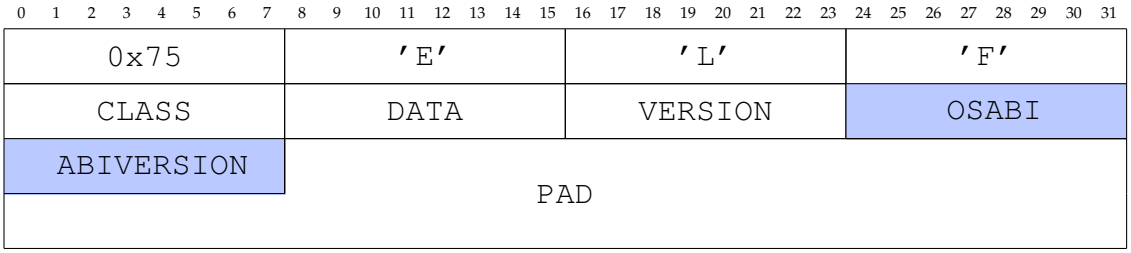


Figure 3.4: Structure of the `e_ident` field of the ELF header. The highlighted fields, `OSABI` and `ABIVERSION`, are used to implement branding.

To acheive this, I use a technique used by FreeBSD: *branding* the program executable to signify which ABI it uses[9]. Both FreeBSD and DIOS use the ELF file format for executables, so this technique can be applied to solve this issue. There are several techniques for branding an ELF file, but the neatest and most modern approach is to make use of the `OSABI` and `ABIVERSION` fields of the ELF header. These fields are actually sub-fields of the `e_ident` field, which is illustrated in Figure 3.4. There exist a range of standard values for these fields, so I chose unreserved values for DIOS. Table 3.5 lists each of the brands, their corresponding values and how they are treated by the kernel.

ELF loader. The loading of a program from a file into a process is a job for the kernel. DIOS delegates this to the host kernel which, in the current implementation, is Linux. When Linux loads an executable, it examines the file to decide its format, and then invokes the format-specific loader to interpret the file. In the ELF loader, I examine the brand and ensure the `task_struct`, which contains process information, is correctly initialized. I also added validation checks to ensure that a program is started using the correct method.

[8] `http://www.busybox.net/`; accessed 12/05/2015.
[9] `http://fxr.watson.org/fxr/source/kern/imgact_elf.c#L266`; accessed 12/05/2015.

OSABI	ABIVERSION	Name	Launch as	System calls	
				Legacy	Dios
0xD1	0x05	Pure Dios	Dios task	✗	✓
0xD1	0x11	Dios Limbo	Dios task	✓	✓
0xD1	0x13	Legacy Limbo	Legacy process	✓	✓
<any>	<any>	Pure Legacy	Legacy process	✓	✗

Table 3.5: ELF brands used in Dios. *Different brands are limited in the system calls they can make and are launched by the kernel with different methods. Limbo brands are transitional brands to make developing and porting applications easier. Ideally the system would only have Pure* Dios *tasks as this provides the best security.*

DiziBox. The shell must ensure that the program is executed using the correct method: `dios_run` for Dios executables and `fork+exec` for legacy executables. This required modifications to the BusyBox shell used by Dios. By default, the `fork+exec` method is used: `fork` creates a clone of the process and `exec` loads a new executable for the process's text section. I created functions which take the same arguments as the `exec` family of functions and which check how a program should be started before starting it in that way. To inject this code, I used a build rule to replace calls to `exec` with calls to my functions. The resulting shell is not perfect as the prompt does not wait for the program to finish and unnecessarily forks when launching a Dios program, but it is sufficiently functional for its purpose which is to aid debugging given that Dios does not target interactive use.

Command-line arguments. Passing arguments from the command-line (also known as `argv` or the "argument vector") to Dios tasks needs to be implemented. The Linux implementation of Dios uses the usermode-helper API[10] to create user-space processes for new tasks and the API allows the argument vector to be defined. In order to fill the argument vector with the command-line arguments, I had to add an extra parameter to `dios_run`, so that the spawner can choose the argument vector. The implementation of task spawning in the Dios kernel module then needed to be modified to copy the argument vector parameter to the usermode-helper. Care must be taken at this stage, as the argument vector passed in is stored in user-space pages which may not be resident and the kernel does not automatically swap the pages in when they are needed. Instead, the user-space page must be explicitly checked and paged in if required before it can be safely accessed. The final step to link everything together is to modify DiziBox to pass the command-line arguments to `dios_run`.

[10] https://www.kernel.org/doc/htmldocs/kernel-api/modload.html; accessed 12/05/2015.

3.4 Channels

Concurrent tasks which cannot communicate are of limited value. Rust provides channels[11] to allow threads to communicate. Data is placed into a channel at the sending side and removed, in FIFO order, from the channel at the receiving end. I have adapted and implemented the concept of channels for DIOS tasks to allow them to communicate.

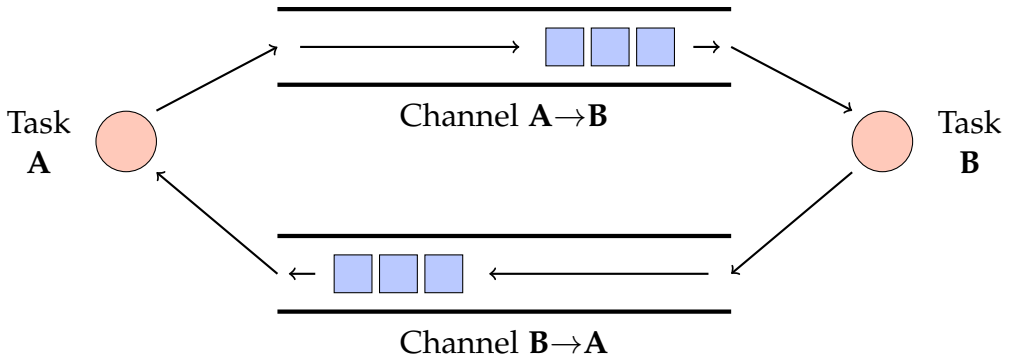


Figure 3.5: Communication between two (circles) tasks using channels. There are separate send and receive routes each providing buffered, in-order, delivery of data (boxes).

To transfer the data from one task to another requires interprocess communication (IPC). On DIOS, this is achieved by creating a shared instance of an object whose type is suitable for IPC. A shared object is one which can be referenced by multiple tasks, i.e. it is named and can be looked up by other tasks, or a reference is delegated to other tasks with `dios_copy`. The stream-style objects are well-suited to the requirements of channels: key examples are shared memory FIFOs and TCP streams. Clearly, a network stream is required if the tasks are on different machines, but a shared memory FIFO is more efficient if both tasks are on the same machine. This suggests that there is scope for selecting the most appropriate implementation for a channel, given information about the tasks.

To expose inter-task channels to Rust, I created a new type which acts as a simple wrapper around DIOS IPC objects. Rust channels only support unidirectional communication, but I made DIOS channels bidirectional for convenience. DIOS shared memory FIFOs are also unidirectional, but a pair of them can create a bidirectional channel using the construction shown in Figure 3.5.

Creating a pair of communicating tasks follows the process in Figure 3.6. This is exposed to Rust by modification of the `spawn_task` function. The signature of `spawn_task` is changed from that in Listing 3.4 to the signature in Listing 3.5. A

[11] http://doc.rust-lang.org/std/sync/mpsc/fn.channel.html; accessed 12/05/2015.

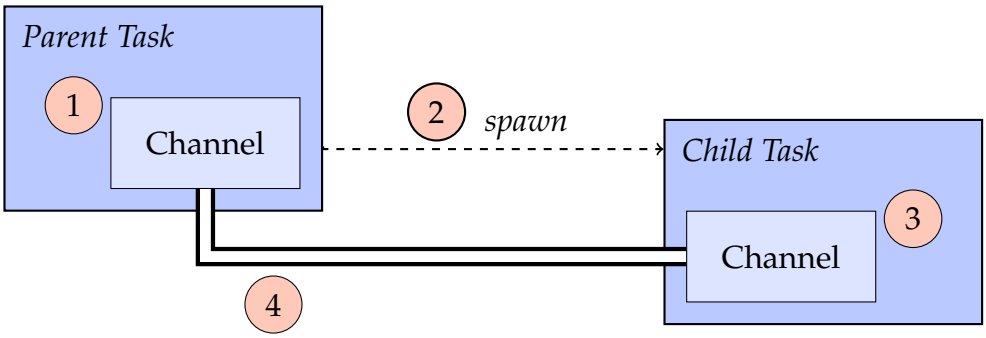


Figure 3.6: The process of setting up communicating tasks. (1) The parent task creates the channel out of Dios *objects. (2) The child task is spawned and passed the names of the channel objects. (3) The child task resolves a reference to the channel objects by looking up the names. (4) The communication channel is ready to send data.*

snippet of the implementation of `spawn_task` is shown in Listing 3.6. The parent task receives its end of the channel from the return value, and the child task is passed its end of the channel as an argument of the entry function. When the child task is starting, the Rust runtime resolves the channel objects which are passed to the entry function.

```
1 pub fn spawn_task(f: fn(Channel)) -> io::Result<Channel>;
```

Listing 3.5: The new signature of the `dios::spawn_thread` function. The entry function now receives a `dios::Channel` which is used to communicate with the spawning task. The return value indicates success, and if successful, one side of the channel is returned.

3.5 DIOS

As part of my project, I made several key improvements to the upstream Dios code base, which I describe in the following.

3.5.1 Reengineering

The initial Dios implementation was not in a clean or maintainable state. It contained substantial code duplication that needed to be kept synchronised in order to maintain correct behaviour. For example, the system call interfaces were defined in a kernel

```rust
pub struct Channel {
  send: *mut libc::dios_ref_t,
  recv: *mut libc::dios_ref_t,
}

pub fn spawn_task(f: fn(Channel)) -> io::Result<Channel> {
  unsafe {
    // Get a reference for the executable
    let exe_ref = libc::dios_pickref(libc::dios_self_exe());

    // Pass the entry address offset in argv
    let entry: *mut i8 = mem::transmute(CString::new(format!("{}",
      mem::transmute(f)-mem::transmute(sys_common::thread::start_thread)
    )).unwrap().as_ptr());

    // Create the channels
    let (chan, mut sname, mut rname) = make_channels().unwrap();
    let mut sname_str: [libc::c_char; 44] = mem::uninitialized();
    let mut rname_str: [libc::c_char; 44] = mem::uninitialized();

    // Launch the new task
    let mut argv = [entry,
                    libc::dios_nametostr(&mut sname, &mut sname_str[0]),
                    libc::dios_nametostr(&mut rname, &mut rname_str[0])];
    let mut task_spec = libc::dios_task_spec_t {
      argc: argv.len() as i32,
      argv: argv.as_mut_ptr()
    };
    let mut new_ref: *mut libc::dios_ref_t = mem::uninitialized();
    if libc::dios_run(0, exe_ref, &mut task_spec, &mut new_ref) != 0 {
      Err(io::Error::new(ErrorKind::Other, "dios_run failed", None))
    } else {
      Ok(chan)
    }
  }
}

fn make_channels()
-> io::Result<(Channel, libc::dios_name_t, libc::dios_name_t)> {
  match (make_shmem_fifo(), make_shmem_fifo()) {
    (Ok((send, sname)), Ok((recv, rname))) =>
      Ok((Channel::new(send, recv), sname, rname)),
    (Err(e),_) | (_,Err(e)) => Err(e)
  }
}
```

Listing 3.6: An extract from `libstd/sys/dios/ext.rs` which contains DIOS*-specific extensions to `libstd`. The `Channel` type and `spawn_task` function implementation included are the public interface of the module. The code interacts closely with the* DIOS *API, which is exposed to Rust through the `libc` module. An `unsafe` block is needed due to the use of the foreign function interface (FFI) to interact with C code, and the manipulation of pointers.*

header file as well as a user-space header file. When making modifications, errors were frequently introduced as a result of the header files differing.

I decided to undertake a reengineering of the Dios code base to resolve these issues and improve the general structure of the project to aid the development of Hephaestus. I merged the duplicated header files into one and updated the `#include` directives accordingly. The Linux kernel needed the include path of the Dios headers added to the build system. Linux uses kbuild[12] to configure builds and I added the include path to the `ccflags-y` (C compiler flags when module is selected) variable of the Dios module. I then split the header files into kernel and user-space sections following the UAPI (user-space API) refactoring effort on the Linux kernel[13]. The aim of this change is to simplify the process preparing the header files that get exported for use in user-space code.

As a result of these changes, updating the system calls and constant values which are shared between the kernel and user-space now consists of a change in just a single file followed by a rebuild. This reduced the number of mistakes I made due to inconsistencies and allowed me to be more productive.

3.5.2 I/O API

As a result of my work, some limitations of the Dios API became apparent. I significantly contributed to the new I/O syscall API for Dios that was designed in response to this [47, ch. 3]. The I/O API is the subset of the systems calls related to I/O. It originally consisted of `begin` and `end` operations for both reading and writing. A buffer was created for the transaction with `begin`, and the transaction was committed and the buffer destroyed with `end`. This meant that the buffer could not be reused and each transaction required two system calls, compared to one system call needed for `read` and `write` on POSIX. Dios offers the advantage of mapping buffers into the user's address space, meaning that there is no need to copy the memory, but it would be desirable to reduce the overhead due to the extra system calls.

The solution we arrived at involves making the buffers more flexible. Buffers take on a Rust-inspired ownership-based management style, illustrated in Figure 3.7, where references own a set of buffers and can borrow or move buffers to and from other references. The `begin` and `end` calls are replaced with a tri-stage sequence of `acquire`, `commit` and `release`.

[12] http://trac.netlabs.org/kbuild; accessed 12/05/2015.
[13] https://lwn.net/Articles/507794/; accessed 12/05/2015.

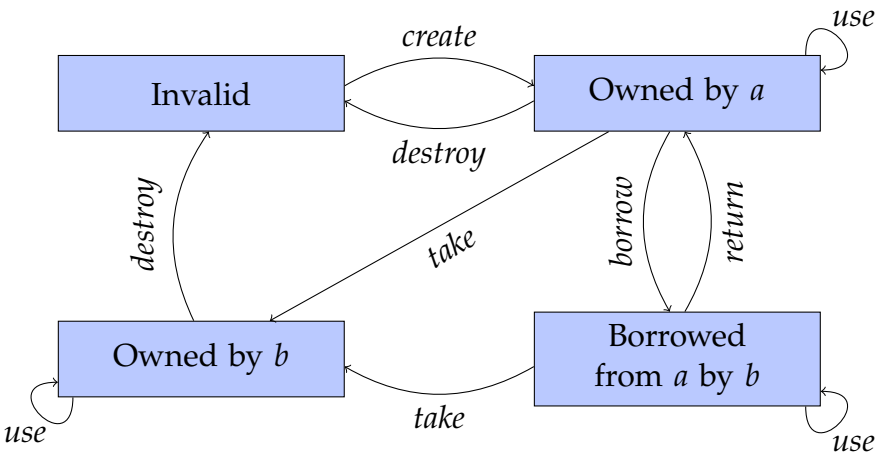


Figure 3.7: State transition diagram for buffers from the point of view of the creating reference, a, and any other reference, b.

- **`acquire`:** Create a buffer associated with the reference, move a buffer from another reference into this one, or temporarily borrow a buffer from another reference. In the case of a read, the buffer can optionally be filled with data to read.
- **`commit`:** The contents of the buffer have been read and the read should be validated, or the contents should be written to the referenced object.
- **`release`:** Destroy the buffer with an optional commit in the case of a write, and return associated resourced to the kernel.

As a result of these changes, the number of system calls required for I/O is reduced. Even though the cost of a single transaction remains unchanged, in the more common case of multiple transactions, the buffer creation cost is amortized over all transactions.

3.5.3 Object interface

All objects in DIOS implement the same set of operations. This was originally implemented using indirection functions which take the object type as an argument and invoke the appropriate object type-specific function. This approach has the advantage that it is easy to trace the code, but has two disadvantages: it does not scale well to more object types and "proxy objects" are not very transparent. Proxy objects are important in DIOS, as they are used to invoke operations on an object from another machine, in effect making a remote procedure call.

My alternative to the indirection functions is to use a structure of function pointers to the object type's operations, a technique similar to the virtual table used in object-oriented languages. An instance of this structure is associated with each object, and the type-specific function for an operation can be found by looking up the pointer in the structure. Proxy objects can now be implemented by replacing the standard operations structure with a structure containing the proxy operations, and scalability improves because new object types just need to publish their implementation of the operations structure.

3.5.4 Object deletion

Resource management is a key role of any operating system is. This involves making efficient use of the limited resources available and sharing them safely between the user-space applications. When I started this project, Dios did not delete objects that were created and instead they were leaked.

Objects can be referenced by many tasks and these references are stored in the tasks' reference table. Since objects are shared, care must be taken to ensure that an object is not referenced by any task before it is deleted. To solve this problem, we could use garbage collection or reference counting. Garbage collection could be used to clean up objects by scanning the reference tables of all tasks to see which objects are still needed. However, there may be many tasks and thus garbage collection has unpredictable performance. This is less desirable than the deterministic performance of reference counting in an operating system context. Rust-style ownership is not applicable to managing references because objects are shared, instead of having a single, well-defined owner.

Accesses to the reference count, for both reading and writing, must be synchronized across concurrent processes. The obvious solution would have been to protect the reference count with a lock. However, locking in the kernel is costly. Lock-free techniques are preferable, as they remove the need for locks and rely on properties of the hardware for consistency. I therefore decided to use atomic counters which are well suited to reference counting. They use memory barriers to ensure that all reads and writes to the variable are fully committed to avoid outdated cache replicas causing inconsistency.

Atomics support a variety of operations, including increment and decrement-and-test, an operation which atomically decrements the value and returns whether the new result is zero. I use these operations to implement safe reference counting by atomically incrementing when a new reference is created, and using the decrement-and-test operation to remove a reference. When the count reaches zero, the object

is deleted. A problem with this algorithm occurs if a new reference is taken after the reference count has already reached zero. In this case the object is already being deleted, but "resurrected" by the new reference. To avoid this problem, I use the increment-not-zero operation in place of the standard increment, as it first checks the count and fails if it was zero. To add the first reference, the count is initialized to one when the object is created.

Atomics still require global memory barriers on a machine. To improve scalability to many tasks on a single machine sharing and object, reference counting techniques such as SNZI [18] counters or Refcache [14] could be used.

Zombification

In some situations, a task may wish to forcibly delete an object via a reference and invalidate other existing references to it. This is somewhat analogous to closing a pipe in POSIX. DIOS must ensure that this is done in a safe manner, and allow other tasks currently performing operations on the object to complete correctly.

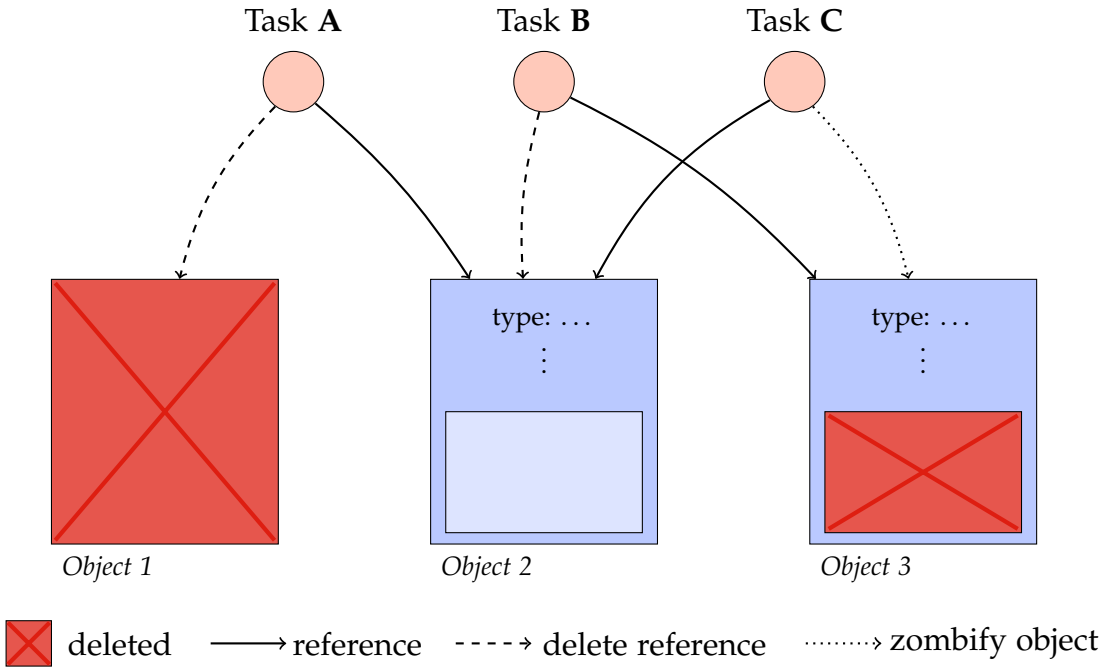


Figure 3.8: Object 1 is no longer referenced by any task, so has been deleted. Object 2 has had a reference deleted, however there are still two other references to it, so it remains live. Object 3 has been zombified leaving the metadata in tact but the internals of the object deleted.

For this purpose, I introduced the concept of a *zombie object*, which is an object that appears to be live but is dead on the inside. This means that the metadata for

the object is still valid (and can be used), but there is no implementation available for any of the object's operations. References to the object are still valid but, since there is no implementation for the operations, any attempt to invoke an operation fails. Figure 3.8 shows the difference between standard reference deletion and object zombification.

In order to convert an object into a zombie object, the operations are removed atomically to prevent any further use. The number of tasks currently performing operations on the object is counted, so that the object is only deleted once there are no tasks left performing any operations. Since no new operations can be started, when the count reaches zero, it is safe to delete the object.

A reference to a valid object can be converted to a reference to a zombie object at any time. In the event that this happens, the metadata for the object does not change but any future attempt to perform an operation on the object fails. On the surface, it may seem wrong or inconsistent that the metadata does not immediately change to inform user-space that the object has been zombified as soon as it happens. However, it is possible to construct a valid ordering so that it appears to the task as though the object was deleted just before it tried to perform the operation.

If the metadata for the reference were to be updated at the point of zombification, it would introduce a race condition between the task which owns the reference and the kernel thread that is making the modifications. Avoiding this race would require synchronization, which is expensive and slow. However, we get synchronization for free when the task makes a system call, as the user-space code pauses and passes control over to the kernel. This can be exploited by waiting for a task to make a system call before making modifications to the object metadata.

3.5.5 Anonymous objects

Objects in Dios are given names to allow them to be looked up by other tasks. But many objects are never intended to be shared (e.g. private memory objects). Name generation itself takes some cycles and once the name has been generated, a mapping must be added to the name table, which is a shared data structure. Access to the name must be protected with synchronisation. This increased use of the name table increases the contention between processes, which can cause them to block, or otherwise impact on performance. Given that many objects do not need to have a name in the first place, it seems wasteful to give them one.

I therefore implemented an optimization that allows objects to be created without a name, but still allows a name to be created if required. This avoids the need to generate the name or access the name table, making object creation faster and reducing the load

on the name table. Another benefit is the reduced chance of name collisions, as a result of the reduced number of names being used in the system.

The assumption that every object had a name was strongly believed throughout the DIOS implementation. Implementing this optimization thus required modifications to eliminate this assumption without impacting correctness. In the new code, it is the responsibility of the user-space programmer to decide when an object requires a name; hence a modification to the `dios_create` system call was required to allow this information to be passed to the kernel.

3.6 Summary

In this chapter, I presented the implementation work I have completed for the Hephaestus runtime. The modifications I made to the Linux kernel, the DIOS operating system, the DIOS C library, the Rust compiler and the Rust standard library have been detailed. First, support for the the core Rust language was provided for DIOS via the Hephaestus runtime. This required the implementation of a user-space memory allocator. I next extended the Hephaestus runtime to allow for tighter integration with DIOS by adding support for multi-tasked programs. The further addition of channels enables tasks to communicate, allowing them to coordinate with each other. Finally, I made improvements to the DIOS kernel in both design and implementation.

Chapter 4

Evaluation

The objective of this chapter is to assess the success of my project and the usefulness of the Hephaestus runtime. I begin by testing the correctness of the Hephaestus runtime, focusing on the modifications that I made for DIOS. I found that Rust programs behave identically under Hephaestus and on Linux, validating the success of my project. Then, I compare the performance of the dynamic memory allocators that I implemented to justify my final choice. Finally, I make a comparison of the performance of programs running on Hephaestus against programs running on existing systems. I found that my Hephaestus runtime offers competitive performance compared to low-level C implementations written directly against the DIOS system call API.

All the experiments were carried out on an AMD Opteron™ 6168 processor, consisting of twelve cores running at 1.9 GHz, with 16 GiB of system memory. The host kernel is a modified Linux 3.14.0 kernel.

4.1 Correctness

These tests are not intended to be extensive tests of Rust's logic, but instead serve to verify that the modifications I made for DIOS work correctly. I wrote some simple programs and compared the output on Hephaestus to Rust on an officially supported platform (Linux). The expected result is that both Hephaestus and Rust on Linux produce the same output and behaviour.

4.1.1 Hello, world!

This iconic program simply prints the string `"Hello, world!"` to the console. This will test that the Dios implementation of `stdout` is working correctly which will be required for the later tests.

```
1 fn main() {
2     println!("Hello, world!");
3 }
```

Listing 4.1: Hello, world! written in Rust.

Listing 4.1 shows the Rust code for this simple program. It runs as expected on both Linux and Dios, printing `Hello, world!` to the console.

4.1.2 Error printing

Rust tries to help the programmer by gracefully handling panic conditions that occur when an unrecoverable error is encountered. An error message, and sometimes a stack trace to explain what happened, are printed to the error stream. The standard library is not available when panicking, so the runtime has its own code path for printing to the console.

Testing this required forcing a program to panic. One way a panic can be induced is by attempting to access an out of bounds array index, as demonstrated in Listing 4.2. The Rust runtime correctly prints the message "`thread '<main>' panicked at 'index out of bounds: the len is 3 but the index is 8', panic.rs:3`" on both the Linux and Dios implementations.

```
1 fn main() {
2     let x = [0,2,3];
3     let y = x[8];
4     println!("{}", y);
5 }
```

Listing 4.2: Forcing a Rust program to panic by accessing an out of bounds index on an array. A panic message is printed to explain that the index 8 is too large for an array of length 3.

4.1.3 Fibonacci

This program calculates the 10^{th} Fibonacci number by following the recurrence relation that defines the Fibonacci sequence:

$$F_0 = 0, \quad F_0 = 1, \quad F_n = F_{n-2} + F_{n-1}$$

This give the expected result of F_{10} as 55. This test checks that basic control flow and function invocation is working correctly.

```
1  fn fib(n: u32) -> u32 {
2      match n {
3          0  => 0,
4          1  => 1,
5          _  => fib(n-2) + fib(n-1),
6      }
7  }
8
9  fn main() {
10     println!("fib(10) = {}", fib(10));
11 }
```

Listing 4.3: A recursive Fibonacci algorithm to calculate the 10^{th} Fibonacci number in Rust. The use of pattern matching, familiar to ML-style languages, but new to C-style languages, makes for a very readable listing.

Listing 4.3 shows the implementation of the recursive Fibonacci program in Rust, which prints the result `fib(10) = 55` when run on both Linux and Dios. This is the expected result and indicates that are no problems.

4.1.4 String manipulation

The Rust standard library offers a variety of string manipulation functions[1]. Rust supports strings allocated on the heap as well as on the stack. This test uses heap-allocated strings which make use of dynamic memory allocation. The test creates and drops strings, as well as removing, inserting and appending other strings and characters. It is expected that there is no memory corruption and that there are no segmentation faults (which would be caused by a malfunctioning memory allocator).

This test is somewhat more complex than the previous simple tests yet no problems were encountered. Rust running on Dios did not experience any memory corruption or

[1] http://doc.rust-lang.org/std/string/struct.String.html; accessed 12/05/2015.

segmentation faults and also produced the same output as the same program running on Linux.[2]

This section tested the capabilities of Hephaestus and validated their correctness. Table 4.1 summarizes the features that have been tested.

Feature	Description	Correct
Control flow	Follow the correct path though the program.	✓
Function invocation	Call and return from functions.	✓
Console output	Print to the standard output console.	✓
Error output	Print to the console when panicking.	✓
Dynamic memory	Allocate memory on the heap.	✓
String processing	Creating, modifying and dropping strings.	✓

Table 4.1: Summary of Hephaestus correctness tests. All features work correctly.

4.2 Performance evaluation

I now evaluate the performance of specific features of my Hephaestus runtime and of the runtime as a whole. Comparing the performance of programs using Hephaestus to the same programs using Rust for Linux and equivalent programs written in C highlights any inefficiencies in my modifications. I did not expect there to be a significant difference in performance between the programs written in Rust for Hephaestus and Linux. However, I expected the C programs to be marginally faster than Rust due to the reduced overhead of an unsafe language, such as lack of array bound checking.

4.2.1 Dynamic memory allocation

In the previous chapter, two possible implementations of `malloc` were presented. The simple allocator simply forwards the request to the kernel by creating a new private memory object for each allocation. The more complex solution, by contrast, uses `dlmalloc` to amortize the cost of creating private memory objects across multiple allocations.

I first compare these implementations on small allocations up to 1024 bytes. This size of allocation is typical of many uses of data being stored on the heap for: example, linked list nodes or small buffers. Figure 4.1 shows the results of this experiment.

[2] While this result may superficially appear trivial, it requires all the layers described in § 2.2.2 to work correctly – a significant achievement on an experimental OS kernel!

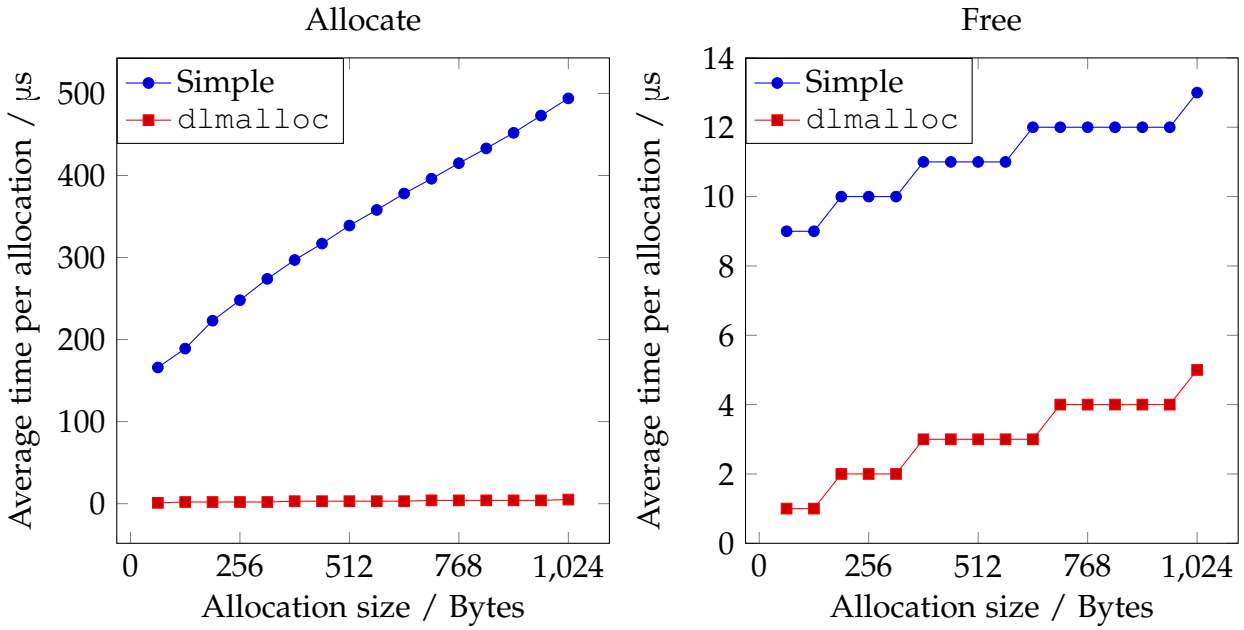


Figure 4.1: Comparing the performance of allocating and freeing small amounts of memory less than 1KiB. Each data point is the average time over 100 allocations.

For these small amounts of memory, the performance of `dlmalloc` is better than the simple allocator by a couple of orders of magnitude for allocation but similar when it comes to freeing. Figure 4.2 and Figure 4.3 present data that suggest possible reasons for the difference in performance. Firstly, it can be seen that the amount of time spent in user-space is similar for both allocators but there is significantly more time spent in the kernel for the simple allocator. This is because the simple allocator makes a system call to map and unmap pages for each and every operation. `dlmalloc` reduces the number of system calls by amortizing the cost of memory mapping across multiple allocations. As a result, `dlmalloc` spends far less time in the kernel; most allocations spends no time at all in the kernel.

I expected the allocation performance for the simple allocator to be constant as all allocations are rounded up to one page. However, this was not the case. After further investigation, I discovered the source of the variation is virtual memory allocation in the Linux kernel that happens when Dios creates iovecs in response to system calls. This further demonstrates the advantage `dlmalloc` gains by amortising system calls

Next, I looked at allocations of larger amounts of memory (multiple megabytes). These are representative of buffers for storing large amounts of data, for example when reading a file into memory. As seen in Figure 4.4, the performance of the simple allocator and `dlmalloc` are very similar for large allocations.

The difference in behaviour for large and small allocations suggests that there is a change in the performance characteristics which was missed by these two tests. In the next experiment, I tried to discover this change by decreasing the interval between the allocation sizes. The results of this test are displayed in Figure 4.5.

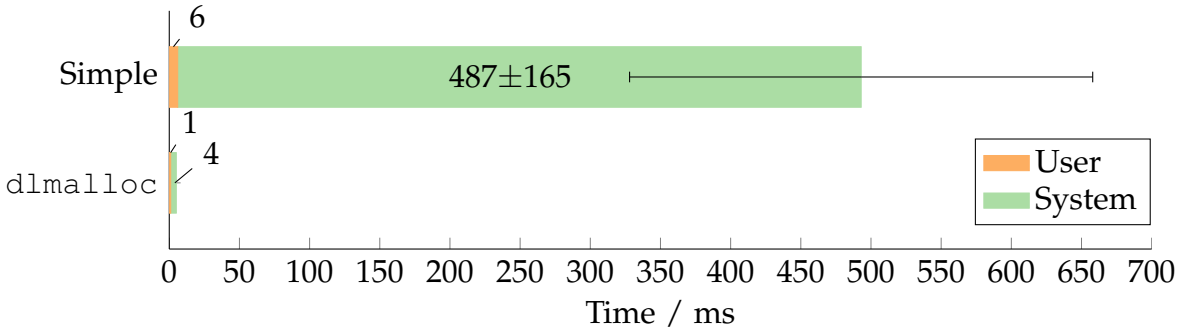


Figure 4.2: Timing of test programs making intensive use of small allocations using the simple and `dlmalloc` implementations of `malloc`. For a total of 1,600 allocations, the simple implementation makes 6,400 system calls compared to just six for `dlmalloc`. There is a large variation in the amount of time the simple allocator spends in the kernel (System).

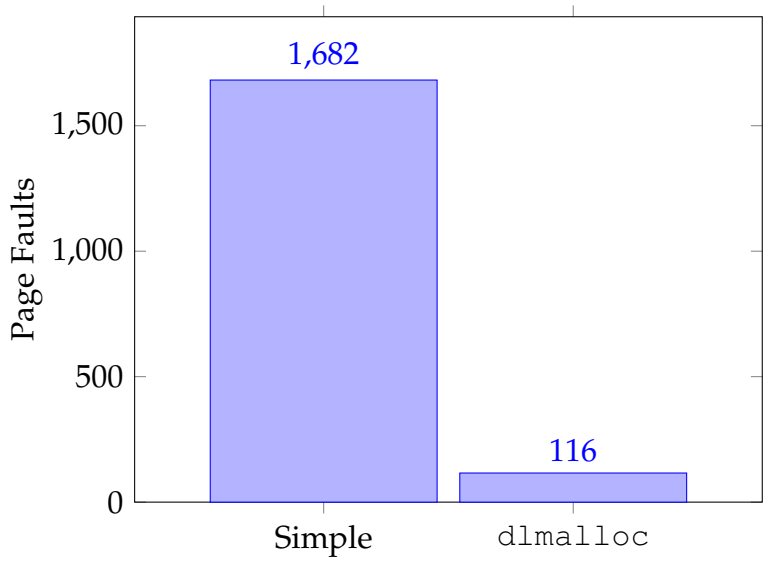


Figure 4.3: Comparison of the number of page faults produced by test programs making intensive use of small allocations using the simple and `dlmalloc` implementations of `malloc`. A total of 1,600 allocation were made. These are minor page faults which do not require I/O, but they do require modification of the page tables. The results indicate significantly greater use of pages by the simple allocator, which lead to overhead observed in Figure 4.1.

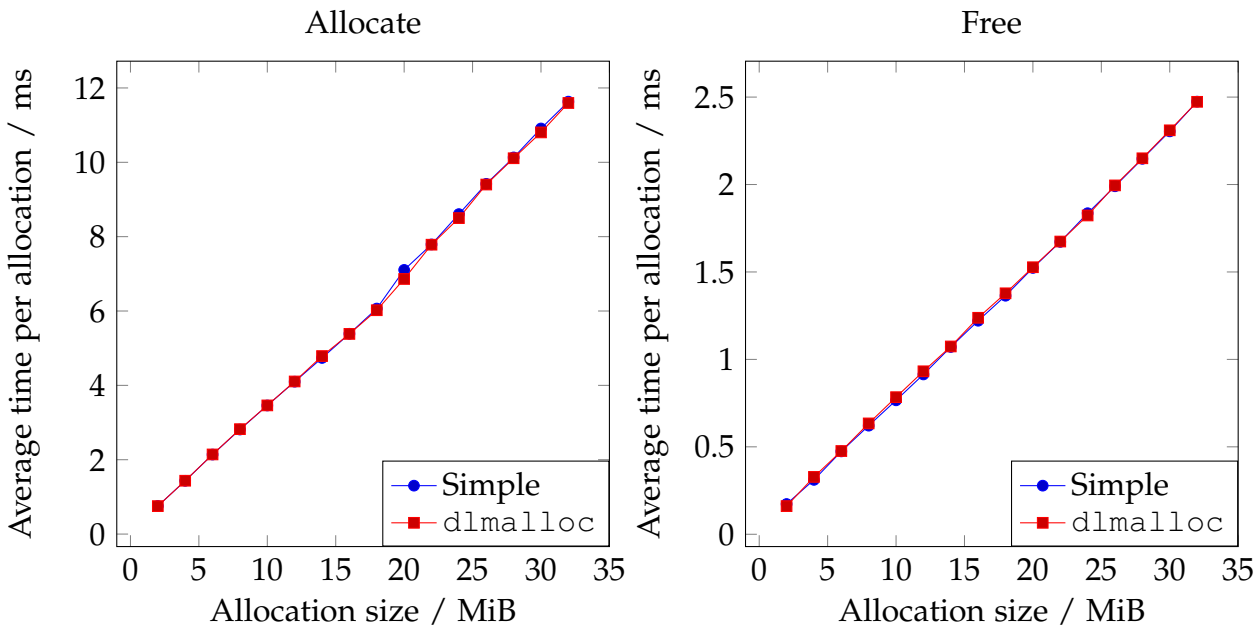


Figure 4.4: Comparison of allocation performance for larger amounts of memory up to 32MiB. The lines follow so closely that the simple allocator's line is obscured behind that of `dlmalloc`. Each data point is the average time over 20 allocations.

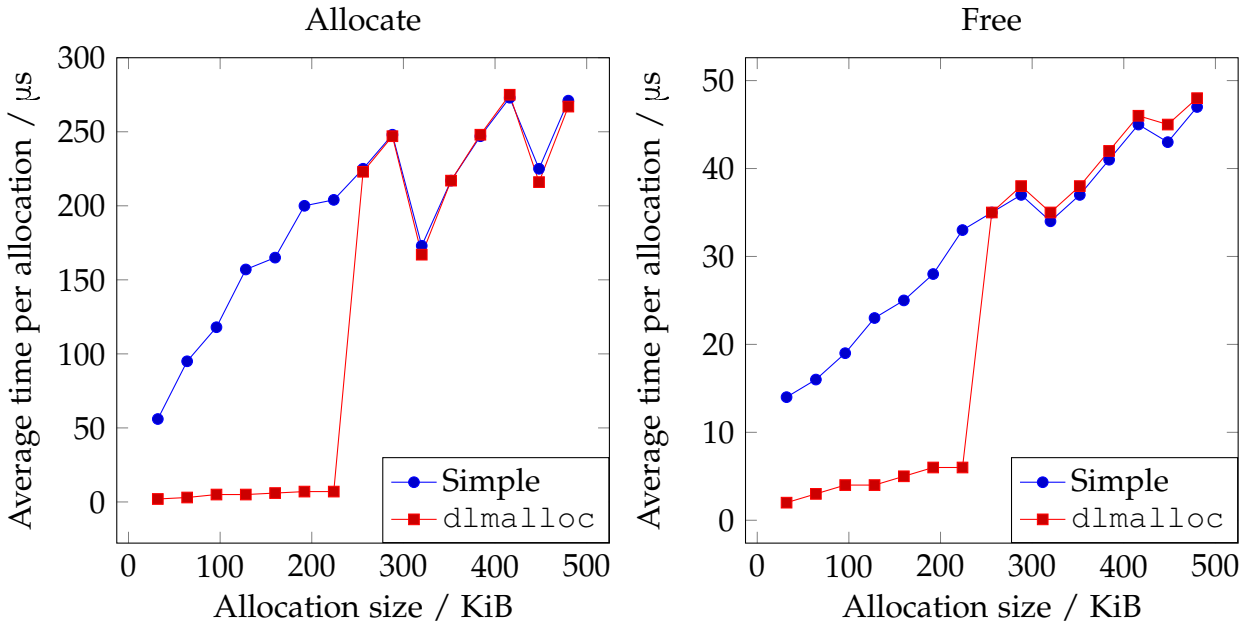


Figure 4.5: Comparison of allocation performance for a range of memory sizes between the large and small analysed previously. There is a drastic change in performance for `dlmalloc` around the 256KiB mark. Each data point is the average time over 20 allocations.

While the allocation overhead of the simple allocator grows at a steady rate, there is a clear jump in the performance of `dlmalloc` at the 256 KiB mark. Before this point, performance is similar to the small allocation experiment, but after this point, the performance closely matches the simple allocator. After inspecting the `dlmalloc` implementation, I discovered there to be a threshold size after which the allocations are made purely by a call to `mmap` instead of using memory pools. This process is very similar to that of the simple allocator, as it allocates a new private memory object for each allocation. The default value for this threshold is 256 KiB which the developers describe in the comments as "an empirically derived value that works well in most systems".

4.2.2 Anonymous objects

In § 3.5.5, I implemented an optimisation that allows objects to be created without a name. Figure 4.6 shows that anonymous objects can be created in half the time it takes to create a named object. While the absolute difference is small (≈5 µs), when summed over many objects this soon becomes significant. On top of this very clear speed up, anonymous objects also do not access the name table, and avoid system wide lock contention. This improves the performance system as a whole.

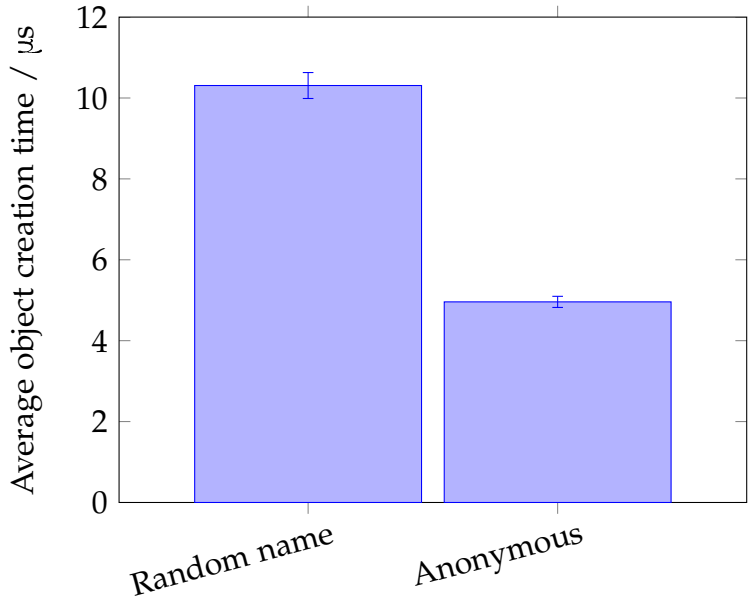


Figure 4.6: Comparison of the time taken to create named and anonymous objects. Each of the ten trials created 1,000 private memory objects, and the average object creation time was taken as the result of the trial. The height of the bar is the average creation time of the ten trials with the maximum and minimum trials represented by the error bars.

4.2.3 Fibonacci

For the first evaluation of my Hephaestus runtime as a whole, I investigate the performance of a CPU-bound program consisting of a single task. The recursive algorithm for calculating the Fibonacci numbers has exponential time complexity, so can generate a large amount CPU-bound work for the test programs.

To increase the workload, the program calculates 30th Fibonacci number instead of the 10th used in the previous test. Increasing this even higher causes the C implementations to experience a stack overflow due to the large number of nested function calls, each of which pushes a call frame onto the stack. This is not a problem for Rust, because the stack is automatically extended by the runtime when the guard pages at the end of the stack are reached. This experience exemplifies the precautions Rust takes to ensure the memory safety of its programs. To make this a meaningful experiment, I made sure to avoid triggering a stack overflow.

The Rust code for this test is the same as in Listing 4.3, only changing the `10` to `30`. No other modifications were required to run it on Linux or Dios.

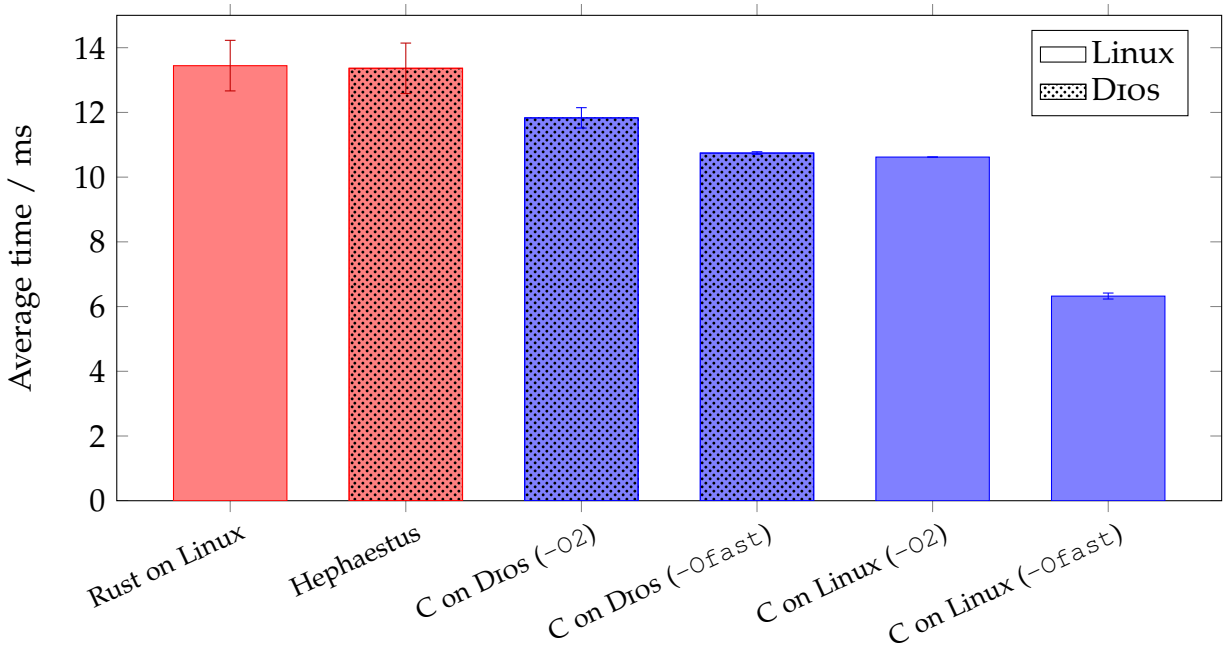


Figure 4.7: Comparison of performance for the computation of the 30th Fibonacci number using the recursive algorithm. The height of the bar is the average of eight trials with the maximum and minimum represented by the error bars. The Rust examples are built with Rust's highest level of optimisation. The `-O2` optimisation flag for the C example corresponds to the most commonly used optimisation level and `-Ofast` is the highest level of optimisation offered by `gcc`.

Figure 4.7 shows are roughly as expected. The Rust implementation running on DIOS and Linux shows identical performance, which proved that my modifications do not affect the performance of CPU-bound programs. The C implementation was expectedly faster than Rust (≈11.5%), but the difference is not insignificant. `gcc` offers five levels of optimisation for speed: `-O0`, `-O1`, `-O2`, `-O3` and `-Ofast`. The levels range from no optimisations to performing many optimisations, including those that may break a standards-compliant program. The Rust programs have been built with the highest level of optimisation, but comparing them to the highest level of optimisation for C (≈52% faster) demonstrates a disadvantage of Rust. The Rust compiler has had far less time to mature than the existing C compilers, which have been gradually improved and accumulated optimisation techniques over many decades, leading to the ability to produce highly efficient programs. As Rust matures, it will hopefully come to benefit from similar experience and improvements made by the developers.

This test produced an excellent result, as it shows Rust programs using my Hephaestus runtime on DIOS are just as fast as those using the standard runtime on Linux.

4.2.4 Fibonacci with Tasks

This test again uses the recursive method for calculating the Fibonacci numbers. However, instead of recursively calling functions, it recursively spawns new tasks. While tasks are a DIOS-specific concept, they can be considered similar to traditional processes since they run in their own address space. In order to compare to Linux, I wrote a Linux implementation that creates new processes in place of spawning new tasks, and which communicates via pipes instead of the DIOS I/O API.

Listing 4.4 shows the Rust code for this experiment when using the Hephaestus runtime. It is very simple and clear, with only small modifications from the single-task version in Listing 4.3. The same program written in C consists of over 100 lines, four times the number of Rust lines. While line counts are not a rigorous means of evaluating a programming language, it does act as an indication of the reduced complexity.

The code in Listing 4.4 also demonstrates an advantage of Rust's type system. The use of the `unwrap()` function is required as all I/O operations return a type which is either the result of the I/O or an error. This forces the programmer to handle errors, resulting in more robust code. `unwrap()` is a naïve form of error handling which asserts that an error will not occur. If the I/O did not produce an error, the result value is returned, but if an error was encountered a panic is induced.

The results of this experiment are shown in Figure 4.8. As expected, the Rust implementation on Linux is somewhat slower than its C counterpart. On DIOS the

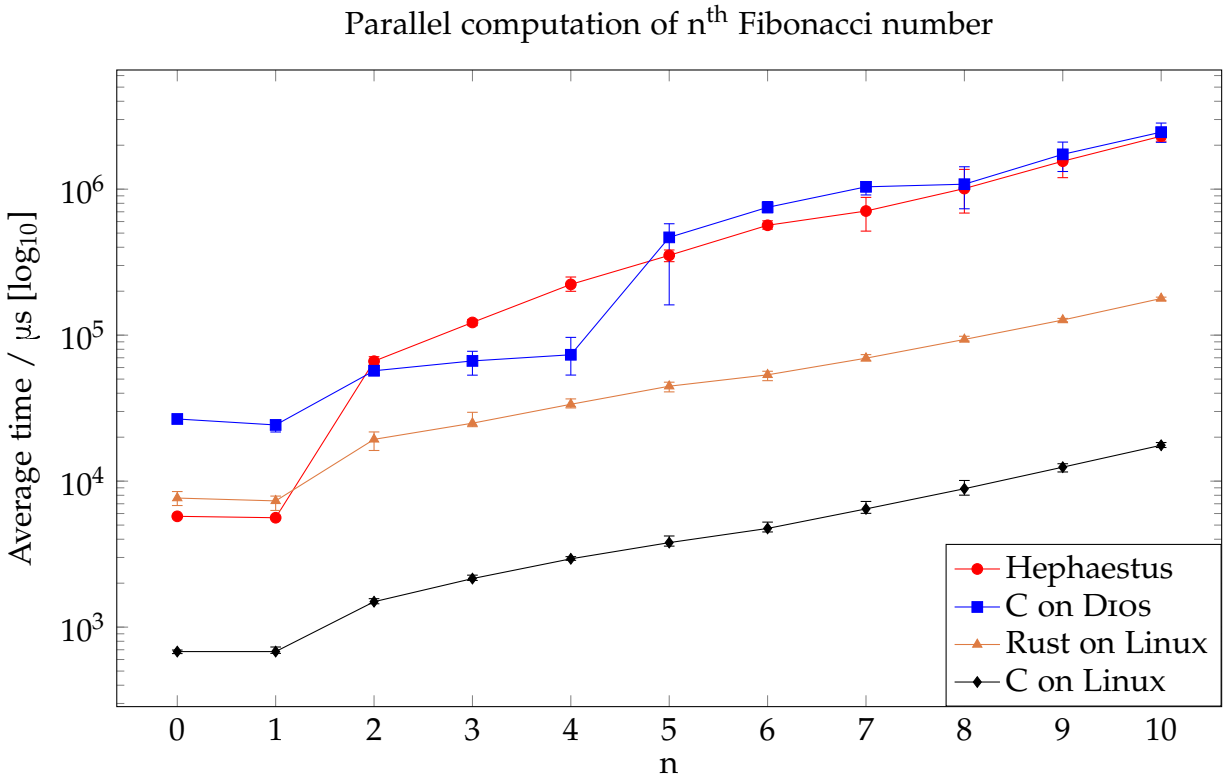


Figure 4.8: Comparison of the parallel computation of the n^{th} Fibonacci number. Each point is the average of eight trials with the maximum and minimum represented by the error bars. Dios *implementations achieve parallelism with tasks, whereas the Linux implementations use processes. On* Dios*, the Rust and C have similar performance due to the the high system overhead of task spawning and I/O.*

Rust and C implementations have similar performance. This is due to there being a large system overhead from task spawning and I/O operations in Dios. As a result, the differences between C and Rust fade into insignificance.

The Dios results are also less consistent, with greater variation in the run time. Again, this is a result of there being a large amount of system work. With the current implementation of Dios being an early stage research prototype, there has been little effort towards optimisation. One possible reason for the difference is the Dios I/O implementation which maps and unmaps a buffer page from the task's address space for each read and write. This extra overhead also helps to explain the significant difference in performance between the Linux and Dios tests. The current Dios I/O path is not efficient for small data transfers such as those used in this example; this will be addressed in future improvement to Dios. However, it is worth noting that Dios's more heavyweight abstractions also bring the benefit of being able to operate across machines.

```
1  use std::os::dios;
2
3  fn fib(n: u32) -> u32 {
4      fn fib_task(chan: dios::Channel) {
5          let n = chan.recv().unwrap();
6          chan.send(fib(n)).unwrap();
7      }
8
9      match n {
10         0  => 0,
11         1  => 1,
12         _  => {
13             let fib1 = dios::spawn_task(fib_task).unwrap();
14             fib1.send(n-2).unwrap();
15             let fib2 = dios::spawn_task(fib_task).unwrap();
16             fib2.send(n-1).unwrap();
17             fib1.recv().unwrap() + fib2.recv().unwrap()
18         }
19     }
20 }
21
22 fn main() {
23     println!("fib(10) = {}", fib(10));
24 }
```

Listing 4.4: A tasked computation of the 10^{th} Fibonacci number using the recursive Fibonacci algorithm. The Hephaestus runtime is used to easily spawn communicating tasks. See Appendix A.1 for the Linux version of this program using processes and pipes.

The result of this test is very positive as it shows my Hephaestus runtime to be performing well, and being very closely comparable to C.

4.2.5 Sum of primes

My final experiment calculates the sum of the primes below 1,000 in a parallel manner. A MapReduce [16]-style approach is taken, with the mappers testing if each number is prime and reducers summing the primes. Twelve mappers and eight reducers are used in each test and Figure 4.9 illustrates the flow of data between the tasks. Again, the Dios implementations use tasks and the Dios I/O API, while the Linux implementations create processes and communicate with pipes. The results are shown in Figure 4.10.

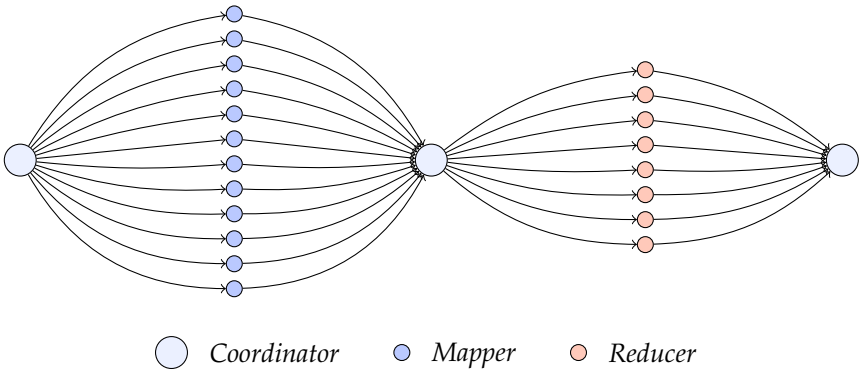


Figure 4.9: Data flow graph of the MapReduce-style parallel computation of the sum of primes. This is not a true MapReduce, as the data is not sent directly between the mappers and the reducers, but is instead sent via the coordinator.

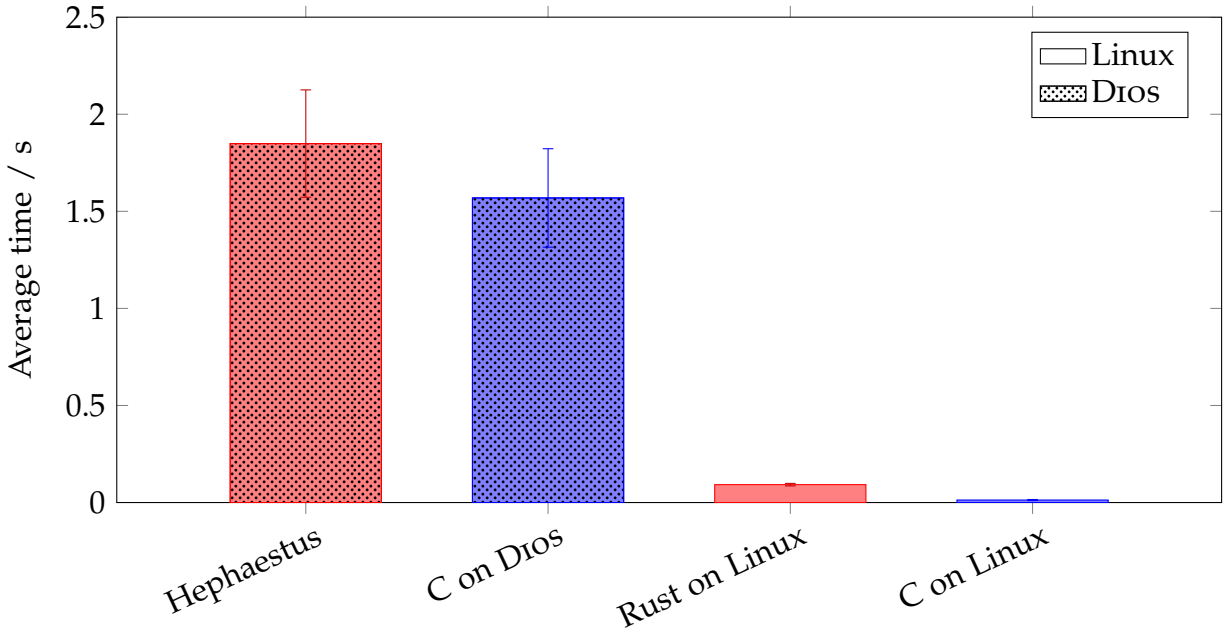


Figure 4.10: Comparison of performance for the parallel computation of the sum of the prime numbers less than 1,000. The height of the bar is the average of eight trials with the maximum and minimum represented by the error bars. The Rust version are slightly slower than their C counterparts, as expected.

As expected, comparing implementations that run on the same operating system shows Rust to be slightly slower than C. The large difference in performance between Dios and Linux tests is again a result of the Dios I/O API. In this case, the modifications to the I/O API discussed in § 3.5.2 will help to reduce this overheard as larger volumes of

data are sent between tasks instead of just single values (as in the previous experiment). The new I/O API allows for the reuse of buffers, meaning mapped buffer pages can be used multiple times before the are unmapped. To take advantage of this, the tasks should create, for example, a single buffer to receive and send all their data. This means that they only incur the cost of a single page mapping, as opposed to hundreds as seen in this experiment.

The source code for these tests is longer than the previous tests, but Hephaestus allows the Rust version (see Appendix A.2) to be written in less than half the number of lines used by the C version. As well as being shorter, the Rust code is much clearer and Hephaestus grants Dios users the benefit of other advantages of Rust.

4.3 Summary

These experiments have shown that my project has been successful. My Hephaestus runtime works correctly in all cases. CPU-bound programs running on Hephaestus have performance that is equivalent to Rust running on Linux, and I/O-bound programs have comparable performance to their equivalent programs written in C directly against the system call API.

The programs written in Rust benefit from strict compile time analysis, which eradicates errors commonly found in C code. They are also shorter and clearer, making them easier to write and maintain. These properties are extremely valuable to programmers, assisting them in producing high quality software – especially in data centre environments, where programmer productivity is usually more important than fine-grained optimisation of low-level code.

Chapter 5

Conclusions

This dissertation described the design and implementation of Hephaestus, a runtime for the high-level programming language Rust on the DIOS distributed data centre operating system. With Hephaestus, programs that take advantage of DIOS can be written more easily, in fewer lines of code and benefiting from Rust's memory safety guarantees.

5.1 Achievements

I have achieved all of my success criteria and also implemented several optional extensions which allow Rust programs to easily create communicating tasks in a DIOS cluster. Programs previously written in hundreds of lines of C code across multiple files can now be condensed into tens of lines of Rust code. However, reaching this appealing simplicity was far from trivial: it required extensive development on DIOS, ranging from resource management in the kernel through utility functions in the user-space C library, and extensions to the Rust compiler and standard libraries.

5.2 Lessons learnt

As a result of working with three substantial existing code bases, I have gained significant experience reading and understanding large volumes of code written by other people. This is a valuable skill, especially in situations when documentation is scarce (as is the case with Rust, DIOS and the Linux kernel).

I have also learnt how to use many tools and techniques indispensable for kernel development, particularly on debugging and tracing. I did the majority of my work using a virtual machine, but used real hardware for the evaluation. Preparing my code to run in a "bare metal" environment proved to be a challenging, but informative experience.

5.3 Further work

I have greatly enjoyed working closely with the Dios developers and I am excited by the opportunities it offers. I hope to continue helping to develop Dios to realise its full potential. Some topics of particular interest to me include:

- **Compound system calls:** to reduce the overhead when making multiple Dios system calls in series, they could be combined into a single system call which carries out all the operations. Similar techniques have been investigated for the Linux kernel, ranging from combining pairs of system calls which commonly occur together [53] to using a simple language to describe the composition of system calls [43].
- **Zero-copy networking:** ways of achieving zero-copy networking on Linux [46] have been shown to provide significant improvements to performance. The Dios I/O API is designed to be suitable for zero-copy drivers, and this provides an opportunity to improve performance. Given the large amounts of data being handled in modern data centres, zero-copy networking has the potential to make a significant impact, as evidenced by recent work on operating systems which offer direct hardware access to applications (e.g. Arrakis [41] and IX [6]).
- **Barrelfish host kernel:** the Barrelfish kernel [4] is designed specifically for many-core processors without cache coherency. It embraces the networked nature of multi-core machines, treating them as a small message-passing distributed system. Dios could potentially benefit from using Barrelfish as a host kernel, especially since Barrelfish includes an Ethernet message-passing driver [26]. A Java Virtual Machine for Barrelfish has been built, and showed that high-level languages work well in the Barrelfish model [36].

Rust's initial model of tasks looked to be a promising match for Dios tasks. Unfortunately, this model was replaced with the more common shared-memory threading model halfway through my project. I nevertheless managed to add support for Rust tasks without requiring shared memory, but I did so by extending the standard library, rather than relying on a language-level feature. Other languages that offer concurrency semantics that might be a good match for Dios include Go and Julia:

- **Go** [42] follows the philosophy "Don't communicate by sharing memory; share memory by communicating." It ensures that all data transfer is explicit and there is less implicit dependence on shared memory. This sounds like a promising match to the semantics required by DIOS tasks, and could allow for more natural integration than was possible with Rust. However, Go has a more elaborate runtime than Rust and requires a garbage collector to operate.
- **Julia** [7] is designed for parallel and cloud computing, It allows for more declarative descriptions of programs granting implementations greater flexibility in identifying parallelism which can be exploited transparently.

It would be an interesting future effort to add support for these languages to DIOS, and my work on Hephaestus has paved the way for such endeavours.

Bibliography

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.

[2] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management – Part 1: General (Revision 3). *NIST Special Publication*, 800(57):67, 2007.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

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

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Broomfield, CO, October 2014.

[7] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. Technical report, September 2012. http://arxiv.org/pdf/1209.5145v1.pdf.

[8] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey:

An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 43–57, 2008.

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–8, 2010.

[10] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.

[11] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, May 2001.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[13] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[14] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 211–224, New York, NY, USA, 2013. ACM.

[15] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2013.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220, 2007.

[18] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: Scalable nonzero indicators. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 13–22, New York, NY, USA, 2007. ACM.

[19] John R. Ellis, Kai Li, and Andrew Appel. Real time, concurrent garbage collection system and method, February 11 1992. US Patent 5,088,036.

[20] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*, 2006.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43, 2003.

[22] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX/SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, May 2015.

[23] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 2, 2012.

[24] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[25] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter if you can JUMP them! In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015.

[26] Jonas Hauenstein, David Gerhard, and Gerd Zellweger. Ethernet message passing for Barrelfish. Technical report, Distributed Systems Lab, ETH Zurich, 2011. http://www.barrelfish.org/hauenstein-gerhard-zellweger-dslab-ethernetmessagepassing.pdf.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, page 22, 2011.

[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[29] IEEE. *IEEE Std 1003.1-2013 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 7*. IEEE, 2013.

[30] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72, 2007.

[31] ISO/IEC. *ISO/IEC 9899:1990 Information technology – Programming languages – C*. ISO/IEC, 1990.

[32] ISO/IEC. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. ISO/IEC, 2011.

[33] ISO/IEC. *ISO/IEC 9899:2011 Information technology – Programming languages – C*. ISO/IEC, 2011.

[34] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.

[35] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[36] Martin Maas and Ross McIlroy. A jvm for the barrelfish operating system. In *Proceedings of the 2nd Workshop on Systems for Future Multicore Architectures (SFMA)*, April 2012.

[37] Michael Matz, Jan Hubicka, Andreas Jäger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement, 2007. http://www.x86-64.org/documentation/abi.pdf.

[38] Earl H McKinney. Generalized birthday problem. *American Mathematical Monthly*, pages 385–387, 1966.

[39] Sape J. Mullender, Guido Van Rossum, A. S. Tananbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[40] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, 1988.

[41] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[42] Rob Pike. The Go programming language. *Talk given at Google's Tech Talks*, 2009.

[43] Amit Purohit, Joseph Spadavecchia, Charles Wright, and Erez Zadok. Improving application performance through system call composition. Technical report, 2003.

[44] Dennis M. Ritchie. The evolution of the UNIX time-sharing system. In Jeffrey M. Tobias, editor, *Language Design and Programming Methodology*, volume 79 of *Lecture Notes in Computer Science*, pages 25–35. 1980.

[45] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[46] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2012.

[47] Malte Schwarzkopf. *Operating System support for warehouse-scale computing*. PhD thesis, University of Cambridge, 2015.

[48] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSYS)*, page 9, 2013.

[49] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[50] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Pearson Prentice Hall, third edition, 2006.

[51] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud. In *Proceedings of the 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[52] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th SIGOPS European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[53] Elder Vicente, Rivalino Matias, Lúcio Borges, and Autran Macêdo. Evaluation of compound system calls in the linux kernel. *ACM SIGOPS Operating Systems Review*, 46(1):53–63, 2012.

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster

computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, San Jose, CA, 2012.

[55] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, and Ion Stoica. The Datacenter Needs an Operating System. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, pages 17–21, 2011.

Appendix A

Appendices

A.1 Rust code for parallel computation of Fibonacci numbers on Linux

```
1  extern crate byteorder;
2
3  use std::convert::AsRef;
4  use std::env;
5  use std::io;
6  use std::process::{Command, Stdio};
7  use byteorder::{LittleEndian, ReadBytesExt, WriteBytesExt};
8
9  fn fib() {
10     let n = io::stdin().read_u32::<LittleEndian>().unwrap();
11     io::stdout().write_u32::<LittleEndian>(match n {
12         0  => 0,
13         1  => 1,
14         _  => {
15           let proc1 = Command::new("./fib").arg("c")
16               .stdin(Stdio::piped()).stdout(Stdio::piped())
17               .spawn().unwrap();
18           let proc2 = Command::new("./fib").arg("c")
19               .stdin(Stdio::piped()).stdout(Stdio::piped())
20               .spawn().unwrap();
21
22           proc1.stdin.unwrap().write_u32::<LittleEndian>(n-1).unwrap();
23           proc2.stdin.unwrap().write_u32::<LittleEndian>(n-2).unwrap();
24           let res1 = proc1.stdout.unwrap()
25               .read_u32::<LittleEndian>().unwrap();
26           let res2 = proc2.stdout.unwrap()
27               .read_u32::<LittleEndian>().unwrap();
```

```
28              res1+res2
29          }
30      }).unwrap();
31  }
32
33  fn main() {
34      let args = env::args();
35      if args.len() > 1 {
36          match env::args().nth(1).unwrap().as_ref() {
37          "c" => fib(),
38          _ => println!("error"),
39          }
40      } else {
41        let mut proc1 = Command::new("./fib").arg("c")
42            .stdin(Stdio::piped()).stdout(Stdio::piped())
43            .spawn().unwrap();
44
45        proc1.stdin.as_mut().unwrap().write_u32::<LittleEndian>(n).unwrap();
46        let rest = proc1.stdout.unwrap().read_u32::<LittleEndian>();
47        let res = rest.unwrap();
48      }
49      println!("fib({}) = {}", n, res);
50  }
```

A.2 Rust code for parallel summation of primes on Linux

```
1  use std::os::dios;
2
3  const EOD: u32 = 0xFFFFFFFF;
4
5  fn filter_prime(chan: dios::Channel) {
6      'all: loop {
7          let input = chan.recv().unwrap();
8          if input == EOD { break; }
9          match input {
10             0|1 => continue 'all,
11             2 => (),
12             _ => for i in 2..input {
13                 if input%i == 0 { continue 'all }
14             }
15         }
16         chan.send(input).unwrap();
17     }
18     chan.send(EOD).unwrap();
19 }
20
21 fn sum(chan: dios::Channel) {
22     let mut sum = 0;
23     loop {
24         let input = chan.recv().unwrap();
25         if input == EOD { break; }
26         sum += input;
27     }
28     chan.send(sum).unwrap();
29 }
30
31 fn main() {
32     const NR_MAPS: u32 = 12;
33     const NR_REDS: u32 = 8;
34     const NR_DATA: usize = 1000;
35
36     let mut mappers = Vec::new();
37     let mut reducers = Vec::new();
38
39     // Spawn tasks
40     for _ in 0..NR_MAPS {
41         mappers.push(dios::spawn_task(filter_prime).unwrap());
42     }
43     for _ in 0..NR_REDS {
44         reducers.push(dios::spawn_task(sum).unwrap());
```

```
45      }
46
47      // Send data using RR
48      for i in 0..NR_DATA {
49          mappers[i%NR_MAPS as usize].send(i as u32).unwrap();
50      }
51      for i in 0..NR_MAPS {
52          mappers[i as usize].send(EOD).unwrap();
53      }
54
55      // Shuffle mappings
56      // ... and dance! --> http://youtu.be/dQw4w9WgXcQ
57      let mut shuffle = 0;
58      while mappers.len() > 0 {
59          for i in 0..mappers.len() {
60              let input = mappers[i].recv().unwrap();
61              if input == EOD {
62                  mappers.remove(i);
63                  break;
64              }
65              reducers[shuffle%NR_REDS as usize].send(input).unwrap();
66              shuffle += 1;
67          }
68      }
69      for reducer in &reducers { reducer.send(EOD).unwrap(); }
70
71      // Collect reductions
72      let mut sum = 0;
73      for reducer in &reducers { sum += reducer.recv().unwrap(); }
74      println!("{}", sum);
75  }
```

A.3 Project proposal

Part II Project Proposal

A High Level Language Runtime for a Distributed Data Centre Operating System

Andrew Scull, St John's College

22nd October 2014

Project Originators: Andrew Scull & Malte Schwarzkopf

Project Supervisor: Ionel Gog

Director of Studies: Dr Robert Mullins

Project Overseers: Prof Larry Paulson & Dr Richard Gibbens

Introduction and Problem Statement

In recent years, there has been a move away from powerful, reliable mainframes towards data centres containing vast numbers of commodity machines. In these data centres, the hardware does not offer fault tolerance or synchronization of resources; instead software is used to manage these issues. Programming on such complex systems is extremely difficult, and as a result, high-level abstractions have developed to hide the complexity.

GFS (Google File System) and HDFS (Hadoop Distributed File System) are examples of abstractions for a distributed file system, and ZooKeeper is an example of a synchronization service. Frameworks are then build on top of these abstractions. Popular Frameworks include MapReduce [16] (for operations which can decompose to map and reduce stages), PowerGraph [23] (for processing graphs) and Apache Spark [54] (for more general purpose use). One common theme between these abstractions is that the programmer does not need to know that their code will run on multiple machines. Instead it appears as though the code runs on a single machine, allowing the programmer to focus on what the code does; not how to coordinate all the machines to do it.

The abstractions above are all user-space application level abstractions. They run on top of an operating system (OS) which is not currently aware that it is part of a larger system. However if the OS were aware of its role in the larger system, then it could use the knowledge to make more informed decisions, resulting in increased efficiency and security. This is the rationale behind a *distributed* OS.

DIOS [48] is such a distributed OS designed for data centre environments. It aims to abstract the whole data centre as a single machine (a "warehouse-scale computer") by automatically distributing memory and processing jobs. Bundling all this complexity into the OS means application writers can share the implementation and instead focusing on what they want to achieve.

As an OS, DIOS has no high-level programming abstraction and all programs are written against a system call API (at present consisting of just 10 functions, compared to the 316 offered by Linux). This means that, although the distribution complexity is hidden, the programmer still has to cope with the complexity of writing their program in low-level C. C has a reputation for being very powerful, but also very difficult to get right due to its flexibility and lack of safety mechanisms. Higher level languages address these issues and focus on helping the programmer write safe code more easily.

Another common aim of high-level languages is to enable code portability. The *runtime* of a language is the layer of abstraction which allows the code of a program to run

without modification on diverse platforms. Runtimes can come in the form of a virtual machine or a runtime library. There are many high-level programming languages which offer portability via a runtime, e.g. Java, Rust[1], Go [42] and the .NET family. Some execute on a virtual machine (e.g. Java), whereas others use a runtime library (e.g. Rust).

Java is an object oriented language originally developed for set-top boxes, but has developed into a multi-paradigm language with a wide range of uses. Java is being used in the distributed data centre environment already: for example, it is commonly used for MapReduce tasks in Apache Hadoop. Java programs are compiled into Java bytecode, which is then executed on the Java virtual machine (JVM). The HotSpot JVM has been developing since it was released and is now very mature with good stability and optimizations. Given the JVM's strongly positive reputation, it has been the target runtime for several languages, most notably Scala (the language used by Spark).

Rust is a modern systems programming language currently being developed by Mozilla. The goal of Rust is to be a good language for writing systems that would normally be written in C or C++[2]. It offers many modern programming language features such as type inference and traits, but also follows a zero-cost abstraction model and has efficient bindings with C. As a result, the language is much safer than C or C++ and has performance comparable to equally safe code written in C or C++.

This project will port the runtime of a high-level language to DIOS, allowing programs written in the language to run on the distributed OS. As a result it will be much easer to write programs for DIOS.

Starting Point

DIOS

DIOS is a distributed OS currently in development by the Systems Research Group (SRG). A working prototype capable of executing simple programs exists. DIOS has a bespoke system call API that does not follow the usual POSIX conventions as they are not suitable for a distributed OS. A major difference with POSIX is that everything in DIOS is an object as opposed to a file. Differences between DIOS and Linux include the size of the system call API (10 functions on DIOS compared to 316 on Linux) and the the fact that any process in DIOS can be executing on a remote machine.

[1] http://www.rust-lang.org/
[2] http://www.infoq.com/news/2012/08/Interview-Rust/

DIOS only has a very small subset of `libc` (called `libd`) implemented. Extending DIOS both in the kernel and user-level libraries will be necessary for completion of the project. Most notable is the current lack of support for `malloc` in `libd`, which will need to be implemented. This must be done conforming to the DIOS distributed memory model.

I have read about DIOS at an abstract level, but so far have had no experience with the details of its implementation. I will need to learn the specifics of its paradigms and system call API before I start the implementation.

Languages and Runtimes

I have had practical experience with Java and C# (one of the .NET languages), but only have a basic understanding of the Rust and Go programming languages. Learning more about them will be part of this project.

Both Rust[3] and Go[4] are open source with their code available on GitHub. Java has now been open sourced, as has the HotSpot implementation of the JVM[5]. There also exist other open source implementations of the JVM. Although .NET is still proprietary, the Mono project[6] maintains an open source implementation of the .NET Framework based on the ECMA standards for C# and the Common Language Runtime.

From initial investigation, Rust appears to be a good candidate for this project. Its runtime library is written mostly in Rust itself; however there is a native crate (Rust module) which allow for OS integration. `liblibc`[7] is the crate which interfaces with the native `libc` and OS system call API and hence will be the focus of the project.

Development Experience

The majority of the implementation work will be in C. I have used C for a number of personal projects, including a basic operating system and a Linux USB driver. This will be useful as DIOS is implemented as a Linux kernel module.

[3] https://github.com/rust-lang/rust
[4] https://github.com/google/go-github
[5] http://openjdk.java.net/
[6] http://www.mono-project.com/
[7] https://github.com/rust-lang/rust/blob/master/src/liblibc/lib.rs

Resources Required

For this project, I will mainly use my quad-core desktop running Ubuntu Linux. I may also use my quad-core MacBook Pro laptop running Mac OS X. I will be running a DIOS instance inside QEMU on top of a patched Linux kernel.

For evaluation, I will require access to a set of machines running DIOS. I will be able to use the SRG's test cluster.

A repository on GitHub will be used for backup and I will pull copies onto my desktop, laptop and MCS account at regular intervals and occasionally onto a USB drive.

Work to be done

The project breaks down into the following sub-projects:

1. Set up a DIOS instance to gain familiarity with it and its system call API.
2. Research the details of the chosen language and its runtime.
3. Configure the language compiler to cross-compile for DIOS (this is not necessary when the runtime is a virtual machine).
4. Identify which parts of the language runtime are dependent on `libc` and the system call interface provided by the OS.
5. Iterative coevolution of `libd` and the DIOS kernel to add required features and optimisations.
6. Build the language runtime against the extended `libd` ready for executing programs on DIOS.

At this point, I will have a working prototype that can execute basic high-level language programs. I will then work on extensions to allow programs to execute distributively using DIOS.

Success Criteria

The project will be a success if I can build and run a program written in the chosen language on a single-machine DIOS platform. When the program is running on DIOS it must:

1. Start at the entry point as defined by the language.
2. Run to completion and exit cleanly unless a language defined failure is reached, in which case fail as the language defines.
3. Produce the same result as the same program run on a existing system that is officially supported by the language.

I will use a suite of test programs to test the features of the runtime. Programs in the test suit must meet the criteria above to pass. The features to be tested will depend on the chosen runtime but may include:

- Program flow (i.e. starting, finishing and failing correctly)
  - Example test: Recursive calculation of n^{th} Fibonacci number
- Printing output
  - Example test: Hello, World!
- Dynamic memory allocation and deallocation
  - Example test: Linked list with nodes allocated on the heap
- Bindings to native functions (e.g. importing libc functions)
  - Example test: Calling `libd` functions or DIOS system calls

For evaluation I will compare my implementation of the runtime with various baselines. The findings of the evaluation may help to identify opportunities for optimisations in DIOS, `libd` and my implementation. The metrics I will evaluate will include:

- The number of system calls used to implement the runtime features in my implementation compared to an existing implementation.
- The runtime of a program running on my implementation compared to the runtime of the same program written directly against the DIOS system call API in C.
- The performance of my implementation before and after applying an optimisation.

Possible Extensions

If I achieve the above early, I will try to extend the runtime to work with across multiple machines to take advantage of DIOS's concurrency and distribution mechanisms.

Possible extensions include:

1. The ability to run a program which can spawn a task on a remote machine.
2. The ability to share objects between separate tasks running on DIOS, possibly on different machines.
3. Optimisation to improve performance of the runtime where applicable.

Timetable: Workplan and Milestones to be achieved.

9th October – 22nd October

- Work on project proposal.
- Investigate high-level languages which may be suitable targets for the porting.

Deadline (13th October): Phase 1 project proposal submission.

Deadline (17th October): Draft project proposal submission.

23rd October – 5th November

- Finish on project proposal.
- Set up development environment including installing DIOS on a QEMU instance.
- Prepare the chosen language's compiler to build for DIOS.
- Begin implementing a minimum set of features required for a functioning system.

Deadline (24th October): Final project proposal submission.

6th November – 19th November

- Continue implementing a minimum set of features required for a functioning system.
- Implement any functionality required in the DIOS kernel module.
- Write a suite of programs to test the runtime, covering the runtime's features.

20th November – 3rd December

- Finish implementing minimum set of features modifying the DIOS kernel module as necessary.
- Identify a set of extra features which would enhance the runtime and implement those.

4th December – 14th January (Christmas Vacation)

- Ensure the basic success criteria of the have been met.
- Fix bugs and other identified issues.

15th January – 28th January

- Complete progress report for upcoming deadline.
- Prepare for for progress report presentation.
- Evaluate the the working runtime with the test suite.
- Write the introduction section of dissertation.

Deadline (30th January): Progress report submission.

29th January – 11th February

- Apply optimizations as identified as relevant, prioritised by importance and usefulness.
- Begin research into requirements for project extensions.
- Write the preparation section of dissertation.

12th February – 25th February

- Continue applying optimizations as identified.
- Implement features that will enable a program to spawn remote tasks.
- Write the implementation section of dissertation.

26th February – 11th March

- Continue applying optimizations as identified.
- Implement features that will allow tasks to share objects if task spawning feature are complete.
- Complete initial draft of dissertation by writing evaluation and conclusion.

12th March – 22nd April (Easter Vacation)

- Evaluate the extension features and optimisations.
- Ensure all implementation and evaluation work is included in the dissertation.
- Add detail make sure that the dissertation contains all the required features.

23rd April – 15th May

- Buffer time to allow for recovery from earlier delays.
- Proof read and finalize the dissertation.
- Exam revision.

Deadline (15th May): Dissertation submission.