# TOWARDS A SECURE UNIKERNEL

**(A Rust Based Kernel)**

*A thesis submitted*

in Partial Fulfillment of the Requirements

for the Degree of

Bachelor of Technology

by

**Katta Vikas Reddy**

**EE12B033**

*to the*

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

May, 2016

# CERTIFICATE

It is certified that the work contained in the thesis titled **TOWARDS A SECURE UNIKERNEL**

**(A Rust Based Kernel)**, by **Katta Vikas Reddy**

**EE12B033**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

---

Prof. Chester Rebeiro

Department of Computer Science & Engineering

IIT Madras

May, 2016

# ABSTRACT

Name of student: **Katta Vikas Reddy**

**EE12B033** Roll no: **EE12B033**

Degree for which submitted: **Bachelor of Technology**

Department: **Electrical Engineering**

Thesis title: **TOWARDS A SECURE UNIKERNEL**

**(A Rust Based Kernel)**

Name of Thesis Supervisor: **Prof. Chester Rebeiro**

Name of Thesis Co-Guide: **Prof. Devendra Jalihal**

Month and year of thesis submission: **May, 2016**

In this project we developed a kernel in Rust programming language, which is directed towards building a Unikernel. In this thesis we will look at some of the security threats for Operating systems, followed by discussion of requirement of Unikernels in the cloud architecture. We chose Rust as the choice of language, and there is a chapter on why Rust is the most exciting language for building Operating system like software. The kernel is built up to the stage of memory allocation algorithm, and we present the kernel in more detail in a chapter.

# Acknowledgements

I would like to thank all the people who helped me during my project and thesis.

# Contents

# List of Figures

# Chapter 1

# Introduction

Computer technology has made a phenomenal progress in the last six decades since the first general purpose electronic computer was created. The role of operating systems has been very vital in the journey of computers from being just a scientific tool to being a necessity in every walk of life. Operating systems have evolved a lot from simple batch processing systems to the present library operating systems. After 1970's the development of operating systems has been guided mostly by the rapid advent of faster processors and cheaper memory, and has been oriented towards the development of desktop operating systems. Internet in the 1990's opened up the new area of cloud computing. Server operating system for the cloud, were developed over model of already existing kernels as the differences in requirements were marginal compared to the desktop operating systems. Over the last two decades internet has really caught on and now a lot of computing is being done over the servers, paving way for the cloud computing. Security and efficiency are the two important reasons, for the search of alternatives for server operating system, though the virtualization and advancement of host operating systems has made cloud computing quite effective. Unikernel architecture will be discussed in a later chapter, and we will see how it can be a better solution. Before that we present the security threats of operating systems, and how the programming languages can help us overcome some of the threats. Rust, which is quite new seems to be a better choice for the kernel development and more information about Rust is presented in 4th chapter. In the next section, we will look

at some of the general security threats for operating systems.

## 1.1 Operating systems security

Security has been and still remains a major concern for operating system developers and users alike. Security is, in general, keeping unauthorized systems from accessing the system, and preventing them to exploit the resources. It is traditionally defined by three attributes of confidentiality, integrity and availability. As in the OS literature, confidentiality is the prevention of unauthorized disclosure of information. integrity is the prevention of unauthorized modification of information, and availability is the prevention of unauthorized withholding of information or resources.

One common view of operating system is that it is a virtual or extended machine, i.e., it masks the details of the underlying hardware from the programmer and provides the programmer with a convenient interface for using the system. It can also be viewed as resource manager, which is responsible for fair resource sharing between different processes in the system. So when multiple users are active within a system, the operating system must ensure protection of one user from another. It should also ensure the protection against unauthorized users. Permissions in the systems are based on identity, which in turn are based on authentication. Passwords, voice recognition, fingerprints, face scanning and eyeprints are some of the common authentication measures. Cracking the authentication measures is usually difficult, so the hackers resort to other ways for gaining the access of the system. We will present some common attacks and their impact in the recent years.

As said previously, authentication attacks usually lie at the bottom of the priority list of the hackers. The major problems are targeted attacks like, trojan Horses, login spoofing and buggy software. Trojan horses are basically programs that are disguised programs, meant to harm the system code and resources. Authenticated users are usually enticed with attractive offers to run the disguised program that may affect the system, more commonly for the data. There are a lot of malware attacks, where the word malware stands for malicious software. Malware includes computer viruses,

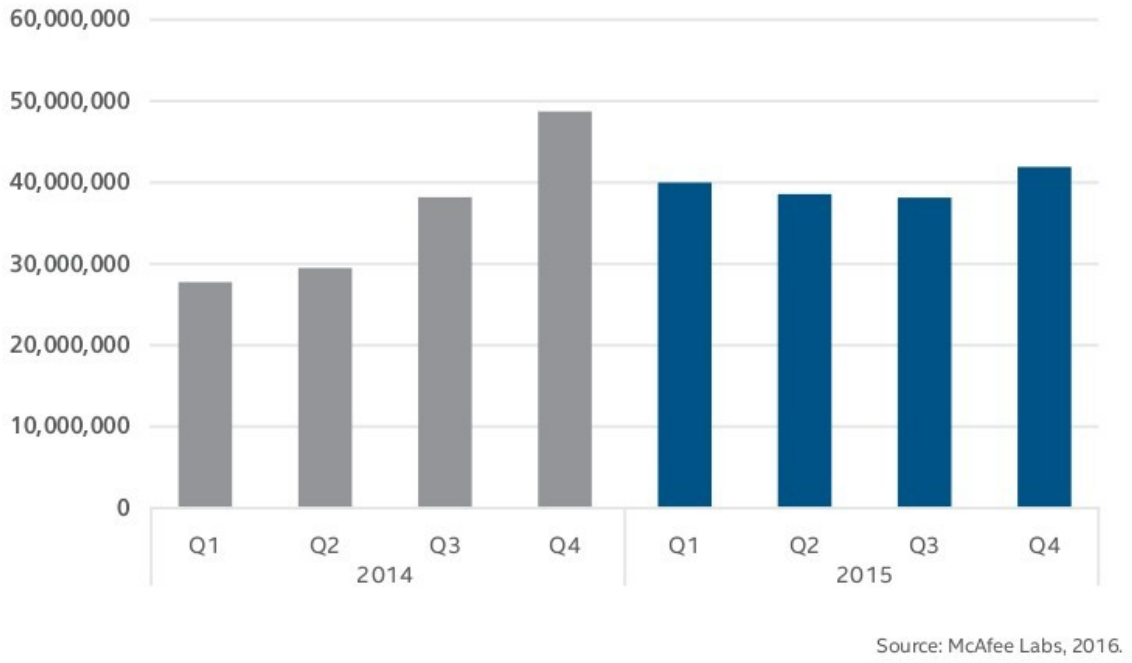


**Figure 1.1:** New Malware

worms, spyware and other malicious and unwanted software. Computer viruses are the programs that embed themselves in some other executable software, whereas a worm is a stand-alone malware program that actively transmits itself over a network to infect other computers. Some of the most common malware are rootkits and autorun malware. Rootkits, or stealth malware, are designed to evade detection by anti-virus softwares and reside on a system for prolonged periods. Autorun malware, are the kind of malware that often hides on USB drives and can allow an attacker to take control of a system. There are also attacks like backdoors attack, which is a method of bypassing normal authentication measures, usually over a network such as internet. The figures were from the data collected by the Mcafee anti-virus software company. As we can see in the figure, total number of new malware attacks in the recent years is very high, ranging in tens of millions, while the total number of malware is in the range of half a billion.

As mentioned earlier, a lot of malware is created to take assistance from Buggy software for the intended attacks. One of the most commonly exploited bug is the buffer overflow. It is usually either the stack-buffer overflow or the heap-buffer

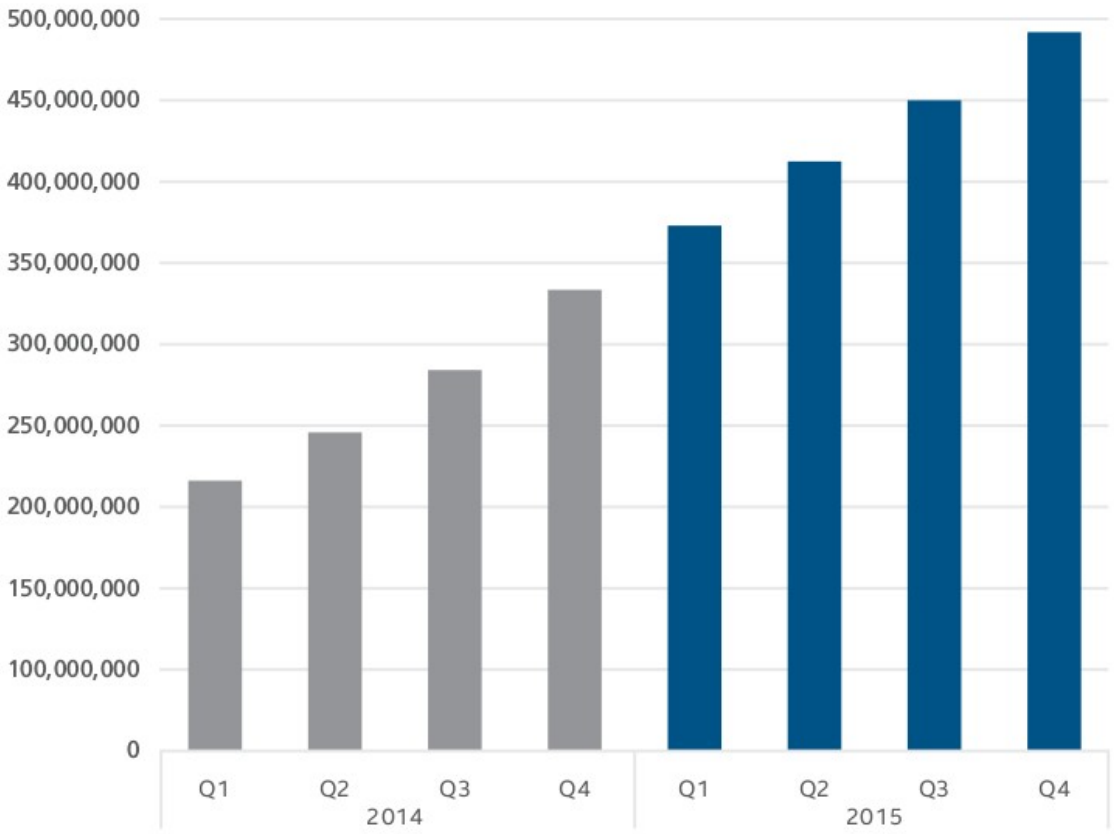


**Figure 1.2:** Total Malware

overflow. More information about buffer overflow attack is presented in the second chapter.

## 1.2 Impact of Languages on Security

Most of the large and popular software like operating systems are written in C, which doesn't have the memory checks like bounds checking for built-in buffer type like arrays or the garbage collection that is present in the modern high level languages. This is the reason for a lot of buggy software. Many new modern languages like Java came up with features like garbage collection for the memory checks, but they can't be used for the system level programming as they doesn't offer the control that is offered by C. The issue of security is also quite important for web browsers as a lot of web-based malware attacks occur. Mozilla Organisation employees initiated the creation of a new memory safe language, called Rust to use internally. But

the features they proposed, which we will discuss in a later chapter are relevant in almost every other software development. The development was widely accepted by the open-source community, and a lot of work has been put in developing Rust, a memory safe language, which can be used for systems programming as well for other high level purposes. In chapter three, memory safe features of Rust will be presented.

## 1.3 Organisation of the report

The second chapter presents memory overflow vulnerabilities and current state of the art countermeasures. The third chapter discusses Rust programming language. A brief introduction to Unikernels is in chapter 4. Chapter 5 and 6 presents our work on the Rust kernel, which we call oxide while chapter 7 concludes the report.

# Chapter 2

# Security threats in Operating systems

In the previous chapter we discussed briefly about the security of large software and in particular, hat of the Operating systems. Though there are different kinds of security threats, we won't be looking at all classes of them. In this chapter we focus on the security threats in an operating system that are related to the memory buffer overflow. This is because, it can form a strong foundation, to realise the need for a new memory safe language.

## 2.1 Buffer Overflow Attack

Buffer overflow occurs when a computer program overruns the buffer and writes in the locations adjacent to the buffer. Usually checking bounds should be done to know the limits of buffer size the program is permitted to access. These overflows usually occurs when the data written exceeds the size of the destination buffer and our program does not implement checking bounds of the buffer. Stack-buffer overflow and heap-buffer overflow are the types that are encountered. This flaw has been used by the hackers to exploit and gain access to the system. Languages like C and C++ do not implement bounds checking for built-in buffer(array) accesses, thus allowing the buffer overflows. Following sections discuss the various cases of buffer overflow,

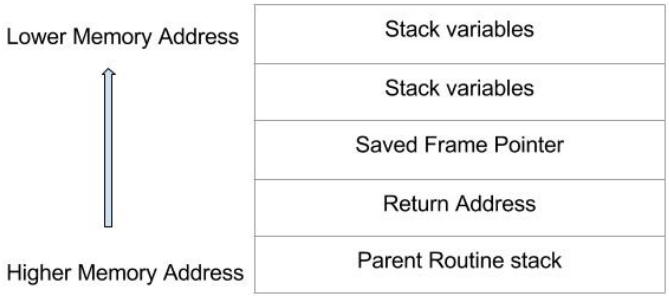


**Figure 2.1:** Stack structure in a program

namely stack-buffer and heap-buffer overflow.

### 2.1.1 Stack-buffer overflow

This occurs when the program being executed writes in the memory locations adjacent to the object area into the stack. Stack is usually a fixed size buffer which contains statically allocated program variables, saved frame pointer and return address of the caller routine among other important things. The program most likely crashes when stack is overwritten as it usually overwrites things like caller routine return address on the stack.

An exploit that intentionally uses stack overflow is called stack smashing. In case of a deliberate attempt of stack smashing the user can overwrite the buffer with executable code, which can be used to gain access of the computer, if the program was running with special privileges. More details about some stack-smashing exploits is presented in further sections.

### 2.1.2 Heap-buffer overflow

Objects in the heap are dynamically allocated, generally using a function such as malloc() during the runtime. Most heap implementations maintain the meta data of the allocated chunks of memory just before and after the area. Heap exploits generally target that meta data to make the memory shortage crashes or change the funciton pointers overwriting them with the pointers to the functions.

## 2.2 Attacks based on the stack-buffer overflow

If there is a privilege to run the code from the stack area, then the exploits can be simple. The simple buffer overflow to overwrite the static program variables on the stack, followed by overwriting the return address is mostly sufficient to gain the access of the system. As the executing routine returns, the code placed by the earlier program, by buffer overflow is run. This is generally shell code. If the program was running with special privileges, then this code can be used to gain the access of the system.

An approach to prevent the previous attack is to make the code non-executable from the stack. So an attacker can't run the shell code from the stack. The following attacks shows how the hackers can get around to access the system if the stack is made non-executable.

### 2.2.1 Return-Oriented programming

When there is non-executable stack, this method is used to exploit the system. In this method the attacker gains the control of the stack and carefully executes a sequence of machine instructions called generally as 'gadgets'. Each of these gadgets typically ends in a return instruction, and are located either in the user program or the shared library code. Return-to-libc is an example of Return-Oriented programming, where libc is a standard library in C. Generally libc is targeted as it is almost always linked with programs. The hackers in this kind of exploits either rely completely on the

system calls done by the standard library functions or use the program code.

## 2.3 Attacks based on heap-buffer overflow

The overflow generally targets the meta data of the heap, changes the function pointers and uses the resulting pointer linkage to overwrite the function pointer the attacker wants to run. Even if the programs might not have a lot of function pointers explicitly, the runtimes in languages can induce them, making them potential attack areas.

## 2.4 Mechanisms to address the buffer overflow attacks

### 2.4.1 Non-executable stack

Some processors have a bit for marking the non-executable areas in the memory. NX (No-execute bit) or XD(execute disable) bit are the examples. For example the 63rd bit of a page address is used as NX bit in the intel 64-bit processors. This is generally done to avoid the buffer overflow attacks, where the executable code may be written to the stack. But this method does not prevent the return-to-libc attack, as the existing executable code is used in this case.

### 2.4.2 Stack-smashing protection

Placing an integer value before the return address is one of the few methods, that is used to check the buffer overflow. That integer is chosen randomly and is called a canary value. Corruption of canary value generally indicates the buffer overflow. This mechanism can only be used with stack to check the canary value before giving the control back to caller.

### 2.4.3 ASLR

In attacks like return-to-libc executable code from specific libraries is targeted. Address space layout randomization, is randomly arranging the key data areas, such as the base of the executable binaries, the positions of the stack, heap and libraries. For example, if the attacker injected shell code into the stack, the attacker must find the stack address first.

### 2.4.4 Modern 64bit mechanisms

In modern 64bit processors, the calling convention of a function is changed. So, it prevents the return-to-libc attack because they have to write the first argument to the register, which is generally not very easy. Also the system libraries are modified to reduce the programs that typically support the attacks.

Though there are a lot of countermeasures, for the exploits, we can see that they were developed only after the exploits came into light. If we could prove the safety of the software, even before it can be a lot better. We don't do that, but we will see some exciting alternatives for both the current architectures as well as programming languages. Next chapter discusses the features of the rust programming language and how it offers security and control.

# Chapter 3

# Rust

Rust is a systems programming language developed by Mozilla and targeted at high performance applications. It was started by Graydon Hoare, a Mozilla employee, in 2006 as an independent project. This was later taken up by the Mozilla as a full sized project to use it in the development of their new browser, Servo.

Rust is a systems language for writing high performance applications that are usually written in C or C++. It was developed to prevent some of the problems related to invalid memory accesses that generate segmentation faults. Rust's syntax strongly resembles C, but there are remarkable differences between them. Some of them are: 1) No buffer overflow 2) Immutable by default 3) A non-blocking garbage collector like mechanism 4) Generics 5) OOP support through method implemented structs 6) Higher order functions like closures 7) Pattern matching.

Rust 1.8.0 is the latest stable version and it is evolving rapidly with a lot of support from the open source community. In this chapter we will look at the most exciting features of Rust in relevance to memory safety. We will start that discussion with memory safety of other popular languages.

## 3.1 Memory Safety

A program is said to be memory safe if all of its possible executions are memory safe, and a language is said to be safe if all possible programs in the language are memory

safe.

Some of the most common problems that can occur when the programs are not memory safe are: 1) Buffer overflow: This was discussed in the previous chapter. 2) Null pointer dereference: This can cause an exception or the program corruption. 3) Dangling pointers: If an object has an incoming reference from a pointer, and the object gets deallocated without changing that pointer value, then that pointer points to a deallocated memory and is usually called as dangling pointer. 4) Use of uninitialized memory: Use of variables that has not been assigned any value can cause some unexpected behavior as it may contain some undesired value or in some cases a corrupt value. 5) Illegal freeing of an already-freed pointer(called Double freeing), or a non-dynamically allocated pointer. 6) Data races: This occurs when two or more threads access a resource of whose value can be altered.

We use the term 'aliasing' if more than one pointer references the same object. In this context the word 'mutation' is used whenever some changes are made to the object.

Memory safety issues in programs occur because of the combination of two things, aliasing and mutation, in a certain order. Aliasing creates a lot of hidden dependencies which are not obvious. When a variable is changing the location of its memory contents it could update its variable status, but it does not know about the other pointers which are hanging in the system at large. Mutation is changing the memory contents of a variable, either adding the memory or freeing up the memory or changing the values of the variables. If memory allocation is static and we are just reading it, then it is fine, but memory allocations dynamically change when we are mutating the memory contents. It causes the memory to be freed and is a source of dangling pointers.

Languages like C and C++ offer complete control to the programmer. They support pointer arithmetic, allow casting of pointers, and allocation and deallocation of memory by the programmer. Common problems like dangling pointers, iterator invalidation, double free are the result of offering the complete control. Basically

raw pointers imply a lack of memory safety as they can be misused easily. Array is the inbuilt-buffer type data structure in C, and there is no bounds checking for the array type. So this can result in buffer overflows and segmentation faults. There is also no automatic mechanism for deallocating the memory. In C we have to use free() and in C++ it is delete() to free the object memory allocated from the heap. So, there is also a chance of memory leaks, depending on the programmer.

In languages like C++ a lot of conventions about memory safe programming has been developed to achieve the above objectives. In C++ the conventions like passing a const reference when the purpose is to just read, and passing the l-value, r-value references have been developed, but it is up to the programmer to follow them. In other words they are not compiler enforced steps. Next we look at how high level languages tackle the issue.

### 3.1.1 Garbage collection

Every object in Java is created using new() operator, which gets the memory inside heap. The language runtime maintains the reference count for each object, and an object gets destroyed automatically after its reference count drops to zero. The object memory inside heap gets cleared. So there is no chance of dangling pointers.

Modern languages like Java, Ruby and Python ensure memory safety using references and garbage collection. It frees the programmers from manually dealing with memory deallocation, thus removing the problem of memory leaks. Garbage collection needs the language runtime and it comes at the cost of performance.

The next few sections discuss how Rust can be better at addressing those memory issues listed above.

## 3.2 Memory safe features of Rust

It is built with the objectives of safety, performance and concurrency. By default it doesn't use garbage collection, and so allows us to take control about the stack and

heap allocations.

Rust ensures the memory safety of the program at compile time. It has a very light runtime like C, and so it can run in very constrained environments like embedded systems or real-time systems. Thus the final rust code is as fast as C/C++, but a lot more safer.

While the programming languages like Java use garbage collection at the runtime, Rust simply does all the work at the compiler time. During the compile time, it uses different pointers to prove the owner at any point of the program. It goes to the extent of not compiling the code if it can't prove that the code is guaranteed to be memory safe.

Rust solves a lot of memory safety problems by codifying and enforcing the safety patters, and use the type system to implement the safety patterns. The following are the safety patterns adopted by the Rust:

Ownership: When we hand over the ownership of the data structure to a new variable, we are giving all the rights to that variable, the variable that handed over the data structure is no more relevant in the context of the data structure. When no one has the ownership to that data structure, we can free up the memory. This is how rust does memory management. There is no aliasing here, and so there is no problem of mutation.

Shared borrow: In this case we give the references to the data structure but shared borrow entitles them only to the read level rights. Rust allows many shared borrows at a single time. In this case we have aliasing but there is no mutation, because the references we are handing out are read-only.

Mutable borrow: In this case we are handing out the mutable reference of an object. So there is aliasing and also mutation, but we are safe here because we are handing out the mutable reference to just one object. Therefore the only object who can mutate the object is the one who has the mutable reference with them. The object which has mutable reference can lend it to someone, and they get freed after they hand over the mutable reference, except the original owner who stays till the

end to take the mutable reference from some other object.

When we are allocating the dynamic memory in the rust, it is placed just exactly as in the C language. It store the variable like data, length, capacity on the stack and the data points to the memory on the heap.

When we create a vector of vectors and when we handle out the ownership to the new variable, it create all the auxiliary data on the stack newly, points the data to the old vector contents and frees the auxiliary variables on the stack. If we handle that new variable to a function, the variable contents get destroyed by the time the function is executed. This prevents things like ”using after free” and ”double moves”. When we are handing out the shared immutable reference we don’t recreate the stack contents but just give the pointer up into the stack. The pointer can be used to read the contents of the memory and when the function exits we can have that borrowed reference destroyed.

If we create a function which takes a shared reference of a vector, and a mutable reference of a vector and then iterates across them, and if we pass the same vector’s shared and mutable references to that function, the mutable reference may cause the freeing of the vector and the shared references may now be the dangling pointers, and this also causes iterator invalidation as call it in C++. This is one of those problems that garbage collector does not properly handle, i.e., in languages like Java it may cause undefined behavior, it is expected to throw an exception but it is not guaranteed. This is not at all possible in rust because if we have to do that we have to create a shared reference and a mutable reference to that data, but if we try to do that then the compiler throws an error that we can not create a shared and mutable reference at the same time.

Compiler uses the concept of lifetime to enforce these safety patterns. It throws error like ”while something is borrowed it can’t be mutated”, or ”You can’t have a shared and mutable reference at the same time”. Every reference has lifetime as part of it’s type and we can’t use the reference outside it’s lifetime. If we are declaring a borrowed reference to an element within a loop then the lifetime of the variable can

only be confined within the loop. Outside of that loop the borrowed pointer is not valid.

Safe pointers into the stack: The concept of lifetime solves the problem of safe pointers into the stack. If we have a vector in a function whose lifetime is confined to that of the function and when the function is exited that vector is freed, because it's lifetime is confined to that of the function. So if we declare a borrowed reference to that vector and try to return that reference to a variable outside of the function, we get a compilation error. That return would only be valid if we handle out the reference whose lifetime exceeds that of the function.

## 3.3 Other exciting features of Rust

The range function returns an iterator, and it is used in the for loops. Inherently iterators are more safer than direct indexing and so they are extensively used in Rust.

It solves the problem of null value return checking a special enum called Option. Some and None are the two possible values of Option. Options are considered to be the safer alternative to the null pointer checks. Because of the generics, Option enum can hold any type of object.

There is a C like 'if' condition, but there is a more safer option called 'match'. Like switch in C match also checks for multiple conditions, but it is better in terms that it forces the programmer to account for all the code paths possible.

For the stack overflow, we can create a guard page and also use stack probes if the amount of buffer write is more than one page so that it doesn't skip the guard page.

Rust has buffer overflow checking at runtime. It panics if we try to access a memory location which is out of the bounds. This way it also panics when there are segmentation faults with a clear message.

It has support for foreign function interfacing, so it is very easy to interface with the existing C libraries. It doesn't have classes, but it supports OOP with structs

which has method implementations.

## 3.4 Selection of Rust

This project was initiated because the features of Rust looked promising for large software development, though a lot of features are still unstable. In the next chapter we present the need for the Unikernels, and how their architecture can be useful for the cloud computing.

# Chapter 4

# Unikernels

As mentioned earlier, this chapter presents a discussion about the cloud architecture and Unikernels.

## 4.1 Cloud Architecture

Software today are essentially applications sitting on top of operating system. There are a lot of layers of software between the application layer and the physical hardware. The sizes are beefy resulting in long boot times and inefficient memory usage, though most of the applications which are being shipped are single purpose applications. In addition to the previously mentioned drawbacks, large size software are more vulnerable to attacks owing to their larger attack surface.

The reason for this is that the software for cloud is built in the same way it used to be done for the desktop applications. If we disentangle the applications from the operating system, or divide the functionality of operating systems into modular libraries where each modular library in its own right can be shipped with very less dependencies on the other libraries, we can do a lot better. With modular software development we can ship only the libraries we need, and in turn this also makes cross platform targeting very easy. So there is scope of improvement in the architecture to make the cloud computing more efficient.

In the next section we will look more about the current cloud architecture, starting

from the virtual machines and then we follow it up with, how it is evolving to suit the application needs.

### 4.1.1 Virtual Machine

To use the hardware efficiently, a virtual layer is created between the hardware and the operating system that a user uses. This layer has the tools to present each operating system with all the virtual hardware they require. They are also called as Guest OSes, but in general it can be used for anything that runs on a virtual layer. In the next section we look at the hypervisor, which acts as the virtual layer.

### 4.1.2 Hypervisor

Hypervisor can be described in general as virtual machine manager. It is a program that allows multiple operating systems to share a single hardware host. Each operating systems appears to have the host's processor, memory and other resources, all to itself. However, the hypervisor is actually controlling the host processor and resources, allocating what is needed to each operating system in turn and making sure that the guest operating systems don't disrupt each other. There are two types of hypervisors. Type 1 hypervisors are those where they are the layer that lies directly on top of physical hardware, whereas in case of Type 2 hypervisors, a host operating system lies on top of hardware, followed by the hypervisor. Vmware vSphere, Citrix XenServer and Microsoft Hyper-v are a few examples of Type 1 hypervisor, while Vmware workstation which is generally used on desktops and for small scale purposes is an example of Type 2 hypervisor.

Hypervisors are the base of virtualisation in cloud, and on each hypervisor, there are paravirtualisation tools and drivers so that each virtual machine running on the hypervisor, can feel as it is running on an independent host.

Though hypervisors provide virtualisation to a good extent, the guest and the host OS are generally traditional OS's like Linux. They are not designed to take more payload, and so are limited to around tens of VMs typically.

The main drawback of this system is that in cloud we use a guest OS mostly for a single application, but as there are no alternatives, we are doomed to run the entire guest OS leading to larger footprints of memory, and also a high booting time per application. Security is also a compelling issue, because there is a lot of attack surface as it is with a normal desktop operating system.

Recently, the cloud computing industry has really scaled up, and the need for a new architecture led the research and we see new things like containers. In the next section we will look more about the containers.

## 4.2 Container

A container is a form of operating system virtualization that is more efficient than the typical hardware virtualization. It provides the necessary computing resources to run an application as if it is the only application running in the operating system. They are specifically geared towards efficiently running a single application. An application is completely wrapped into a piece of software and directly runs on the hypervisor. Conceptually a container is like a VM. Unlike a virtual machine, in a container we doesn't run the complete instance or image of operating system, with all the kernels, drivers, and shared libraries. Instead of that, an entire stack of containers, which can be a very large number can run on a single instance of host operating system, in a tiny fraction of a footprint, when compared to a VM running the same application.

Though they are better compared to the previous Guest OS systems, it still is not a complete solution because the containers usually carry the entire system libraries with them leading to large memory footprints. Better alternatives like customized operating system have created curiosity as they can reduce the costs and also improve the efficiency and security very much. As we see in the next section Unikernels are a step in that direction.

## 4.3 Unikernels

Unikernels are the new age software which are designed to be shipped with very less amount of extra software. They can be described simply as library operating systems. They are basically applications compiled into their own specialized OS that can be readily deployed on the hypervisors. They are usually virtual machine images, as they are designed to run on top of a virtual layer like Xen hypervisor just to make them uniform across different hardware platforms, as Xen provides a uniform interface for the modules to be written. They have application code specific for the deployment. They are composed of a modular stack which consists of the system libraries based on the configuration specified for the application.

For a unikernel to be compiled with its own modular libraries, there has to be modular libraries with very less dependencies between them. It has to have atleast the following modules, the network stack, the file system stack, modules for the support of user proceses and kernel threads, the language runtime depending on the language the modules and the application are built in, and on the top of it, we need to have the configuration files necessary to compile them together. Any application code, compiled together with its necessary modules as stated in the configuration file, can be called as an Unikernel.

They provide all the advantages of virtual machines and containers. They have considerably lower overhead, which leads to more agile and lower-cost cloud computing. The size of the Unikernels can be very small with the above architecture, and this helps in that the apps can be moved around faster and more cost effectively giving a huge profits with network bandwidth.

There are few places where the unikernels are desired, and we see some of the examples here. If an application needs much disk and processor, unikernels are not of any great importance in those situations. It is a new system, and for large applications, we can't use the already existing libraries for other operating systems and this can be discouraging.

There are a few Unikernels that are already created. ClickOS, HalVM, MirageOS and LING are some of the examples. They differ in things like the choice of programming language, or their focus on certain things. Some are designed only for network processing, while some are for the specific features like safety, security or speed.

Using unikernels the binaries of simple applications like static websites becomes very small. Its basically a virtual machine for every URL. The size of these unikernels also helps in the stream of internet of things, as a lot of devices that have constrained environments can now run the unikernels with only the stack they need and nothing more.

In the next section we will look more about the Unikernel security.

### 4.3.1 Unikernel Security

Linux operating system or the existing Linux containers and Docker images rely on a fairly heavyweight core OS to provide critical services. This implies that a vulnerability in the Linux kernel affects every Linux container. In case of Unikernels, they include only the minimal functionality and systems needed to run an application or service, all of which writing an exploit to attack them much more difficult, but definitely not impossible. The virtual machines are strongly isolated and this guarantees hardware virtualisation and a trusted computing base that is orders of magnitude smaller than that of container technologies. There is no shell in the Unikernels, and because of the absence of exec() we can't even execute a new process for exploiting the system. They allow for careful management of particularly critical portions of an organization's data and processing needs.

The figure below clearly summarises the aforementioned points.

So we see that the Unikernels can be the real game changers in the cloud computing. We plan to build a Unikernel system in the Rust language as mentioned earlier. It is a long process and as a part of this project we only build a minimal kernel in Rust. In the next chapter we will see the current work done and the future

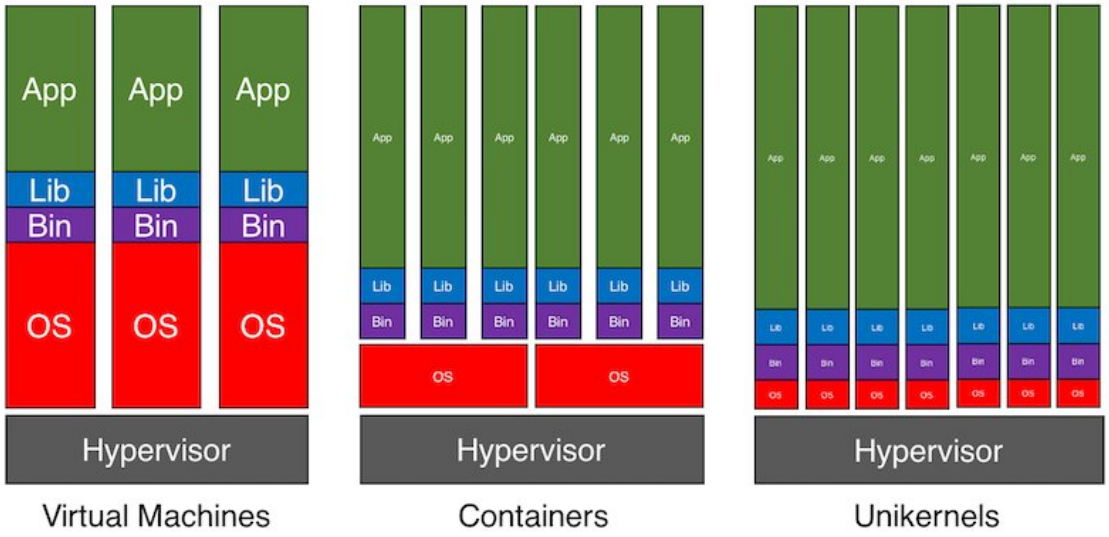


**Figure 4.1:** Virtual Machines vs Containers vs Unikernels

directions.

# Chapter 5

# Bootloading and Kernel Initialization

The kernel we created in this project is a 64 bit kernel which can be booted either on bare metal or in Qemu virtual manager. This chapter presents the initial stages of this minimal kernel creation followed by discussion of booting and also the capabilities of the kernel.

## 5.1 Overview of Booting

When the computer is powered on, it loads the BIOS from a fixed flash memory. It runs self test and initialization of the hardware, then it looks for bootable devices. If it finds a bootable device, it transfers the control to its bootloader. Bootloader is a small portion of executable code stored at the device's beginning. The bootloader determines the location of kernel image on the device and loads into memory. It also switches the CPU to the protected mode because x86 CPUs start in real mode by default.

### 5.1.1 Bootloader

This kernel uses a Multiboot-compliant bootloader called GRUB 2. Multiboot specification is a bootloader standard, and we have to indicate that our kernel

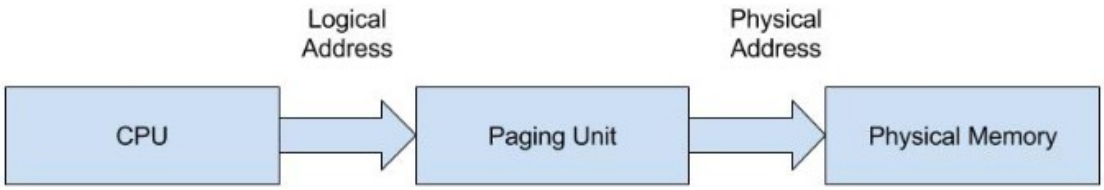


**Figure 5.1:** x86-64 Memory access in long mode

supports Multiboot, so that any Multiboot-compliant bootloader can boot it. The support for Multiboot specification is given by the multiboot_header section in the mutliboot.asm file. It is placed at the beginning of the kernel, by specifying so in the linker.ld file. More about linker file can be found in the section.

### 5.1.2 Linker

We build an ELF executable, to boot later through GRUB, and need to link the object files together. The executable first section is loaded at 1 MB, as we don't want to load the kernel below 1 MB mark because there are many special memory areas like VGA buffer at 0xb8000. It begins with the read-only data, starting with the mutliboot header. Read-only data section is followed by the text section, which is the executable part of the kernel. This is then followed by placing the data and bss sections. All the sections are planned to be page-aligned, with the page size of 4K. This might create more spaces between the sections but the reason for making them page-aligned is that when we setup identity paging, we create 4KB size pages, which we require to be page aligned.

### 5.1.3 Paging

Paging is a memory management scheme that separates virtual and physical memory. The address space is split into equal sized pages, which is 4096 bytes in x86 in long mode. A page table specifies which virtual page points to which physical page. Page table in 64 bit, x86 consists of 4 levels of intermediate tables. They are 1) Page-Map Level-4 Table(PML4) 2) Page-Directory Pointer Table(PDP) 3) Page-Directory Table

(PD) 4) Page Table (PT). This is a general extension of Page directory in 32 bit x86 system as we can see from the naming conventions. Each page table contains 512 entries and one entry is 8 bytes, so they fit exactly in one page. The conversion of the virtual address to physical address is almost similar to that of the 32 bit systems.

### 5.1.4 Assembly part of the kernel

The executable starts in boot.asm and we specify that the instructions followed until switching on the protection mode are the 32 bit instructions. A stack is initialized with the size of 8KB, which grows downwards, and then update the stack pointer with the stack top. Since we have a valid stack pointer, we can call the functions. Initially, we begin with the error check functions as they can be of great help for debugging. Then a check can be done for all the compatibilities with the processor it is supposed to run on like multiboot specification, cpuid and long mode support. This is followed by the paging setup.

We discuss more about paging in the next section. Initially the identity paging is setup with 2MB huge pages, and create the respective pages just above the stack area. Paging is now enabled and the processor is switched to long mode. Paging is enabled by writing the address of P4 table to the CR3 register. Then the long mode is enabled after first enabling the PAE and setting up long mode bit in the EFER register. Though the segmentation is no more used, we still have to write the 64 bit GDT tables, because the GRUB sets up a valid 32 bit GDT initially.

### 5.1.5 Calling Rust code from Assembly

If we have a funtion main_rust to be called, we declare the function with as pub extern. In the file that we want to call, we specify that function as extern main_rust to help the linker. In the rust code we don't use the standard std library. So, for some elementary functions like memcpy, memmove, memcmp and memset from the libraries like libc we get linker errors. We use the rlibc crate which has a partial implementation of the libc library.

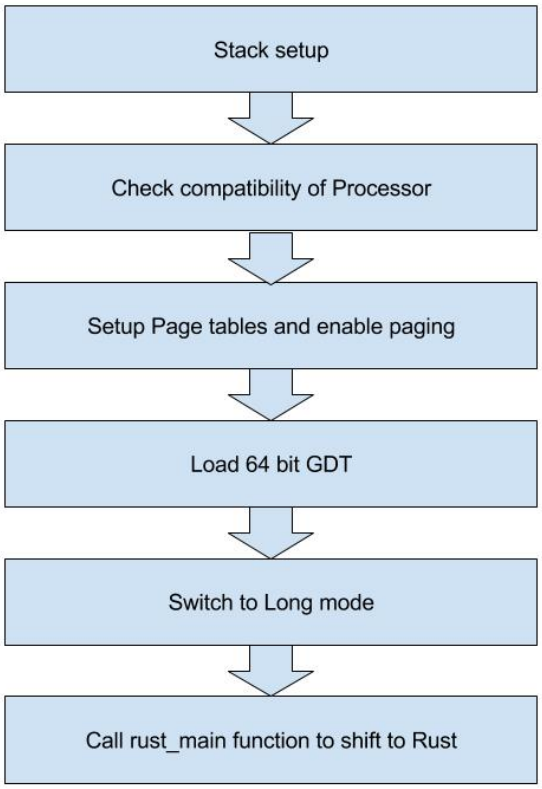


**Figure 5.2:** Assembly code before kernel execution

Initially the VGA buffer memory was edited through assembly routines to print on the screen. Then the interrupts module was written to take keyboard inputs.

### 5.1.6 Interrupts

The interrupt code starts with creation of space for the interrupt descriptor table. Then the interrupt descriptor table is loaded, using the lidt instruction. Then PIC was remapped to change the BIOS defaults. Interrupt service routines were written for keyboard and timer, and placed the addresses of ISR handlers in the descriptors. Then the interrupts supported were enabled in IRQ mask.

### 5.1.7 Paging

Initially paging is setup after identity mapping the first one GB memory, using 1 each of the four page tables that are needed for paging setup in 64 bit kernels. This method can only be used for identity mapping and so we can't use for the separate

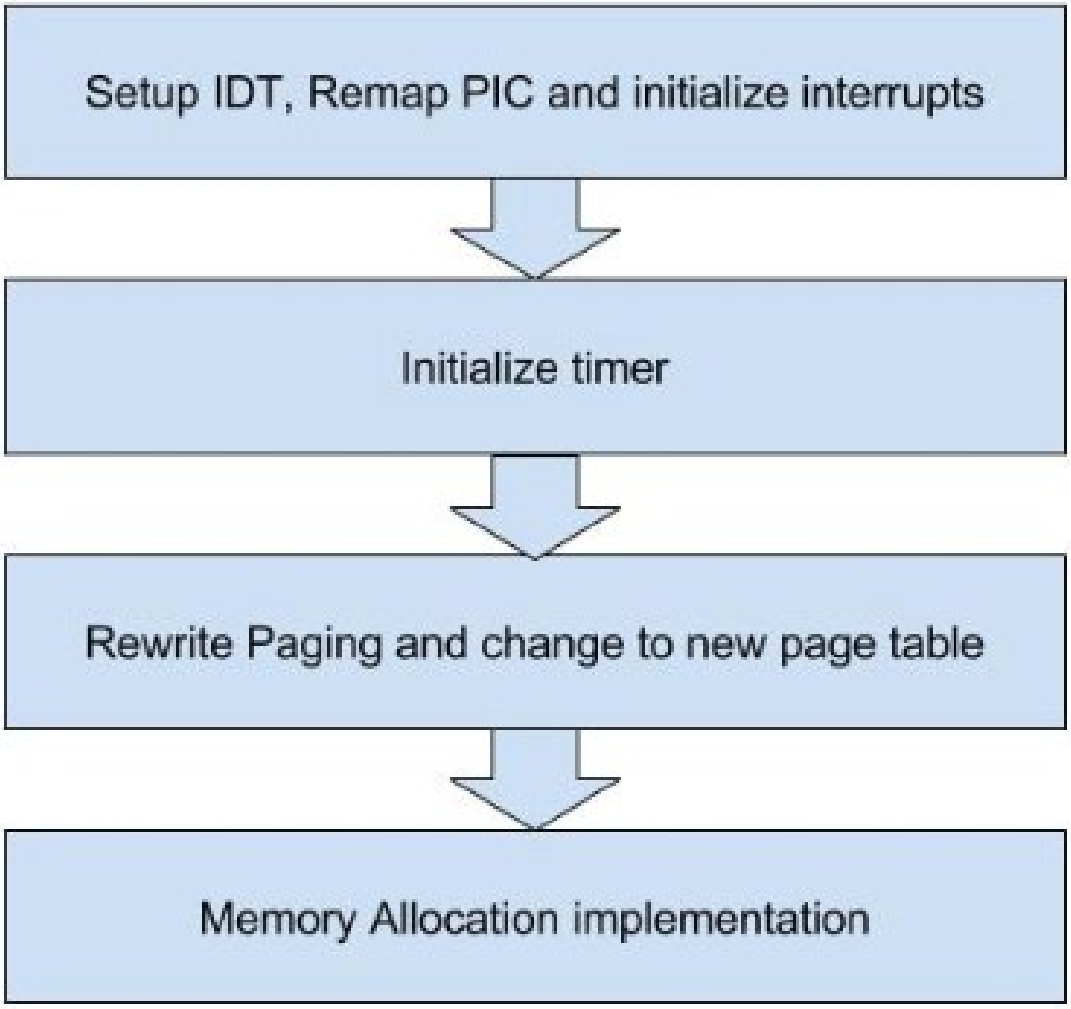


**Figure 5.3:** Code Sequence in Rust

virtual address access for each process. So, we have to setup paging separately and write a frame allocator. This part of the kernel was done. First the frame allocator was written which uses the bitset mechanism to know about the frames that are allocated.

This was followed by the dynamic memory allocation algorithms which we will see in the next chapter. There we will look at the Doug Lea's dynamic memory allocation algorithm that was used in this kernel.

# Chapter 6

# Doug Lea's Memory Allocation Algorithm

This algorithm was started by Doug Lea in 1987, and this has evolved with the help of open source contribution ever since. This memory allocator provides implementation algorithm for standard routines like malloc(), free(), and realloc(), as well as a few auxillary routines. This is popularly called as Doug Lea's malloc, or dlmalloc for short. It's implementation in C serves as the default native version of malloc in some versions of Linux. This chapter presents a description of some of the main design goals, algorithm, and implementation considerations for this allocator.

## 6.1 Goals of a memory allocator

A good memory allocator needs to balance a number of goals: It should not rely on few system-dependent features, such as system calls, but still provide optional support for features found only on some devices. It should be in conformance to all known system constraints on alignment and addressing rules. It should not waste space, so it should obtain as little memory from the system as possible. It should minimize the fragmentation. As it can considerably slow a program, if there are many calls to the allocator, it should run as fast as possible. It is always better to allocate chunks of memory that are used together, as close as possible, so as to minimize

the page and cache misses during the execution. Summarizing the above, we can say that minimizing space by minimizing wastage, due to things like fragmentation, must be the primary goal in any memory allocator.

To see an example, the fastest possible versions of malloc() can be the one that always allocates the next sequential memory location available on the system, while the corresponding fastest version for free() can be the one where it doesn't free at all. As exciting as it might look, this doesn't work in practice, it is pretty easy to run out of memory this way. So, there should always be a space-time trade-off to make an optimal allocator. In the next section we look at the Doug Lea's allocator algorithm in detail.

## 6.2 Memory allocator algorithm

The two core elements of the malloc algorithm are the blocks and the holes. They are the names for the chunks of memory, where a hole is used to indicate a free memory chunk, while a block is memory area that is allocated for the program. These chunks of memory carry around with them size information fields both before and after the chunk, which are called as boundary tags. In general, we include the size of these boundary tags while reporting the size of either holes or blocks. In the next part we will look more about the boundary tags.

### 6.2.1 Boundary tags

There are two kinds of boundary tags, one which is placed before the memory chunk, called header, and another which is placed after the chunk, called footer.

A header is a data structure that consists of the three fields, size which holds the size of the chunk, a magic number which holds a randomly chosen number which is consistently maintained across all the holes and blocks, and a field that holds the bit to say if that chunk is either a hole or a block. Similarly, the footer is placed at the end of the chunk, which contains two fields, one is the magic number, which is

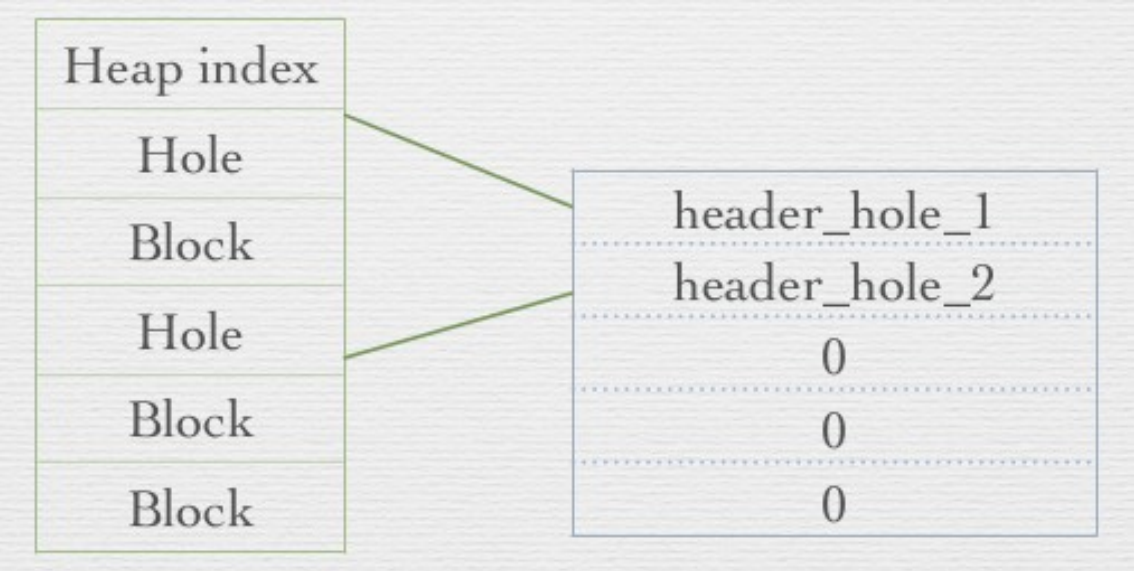


**Figure 6.1:** Memory Layout and Heap Index

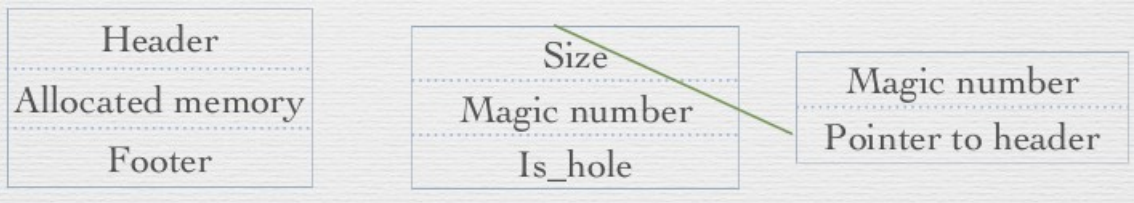


**Figure 6.2:** Layouts of memory chunks, header and footer

the same as in header and is also consistently maintained across all the chunks, and the other is the pointer to the header of the chunk, whose use we will see later. The footer is generally is used for unifying chunks as we see in the later part, but it is sometimes omitted in blocks of chunk that are allocated, to reduce the wastage of space. Though it seems like a good trade-off, it can mislead in the error detection.

### 6.2.2 Arrangement of memory chunks

They are placed such that two bordering holes can be merged into a one larger hole, as it minimizes the number of unusable small holes. Both holes and blocks are bordered together such that, all the memory chunks can be traversed from any known chunk in either forward or backward direction.

### 6.2.3 Allocation algorithm of blocks

There are many versions of allocation algorithm which are based on some small trade-offs. The type of alloc() used in this kernel does not round up block size to powers of two, neither for large nor for small holes. This is because, we can later coalesce the neighbouring chunks, on a free() irrespective of their size. This reduces fragmentation, remarkably as there is never a memory chunk that can't either be a block or a hole.

Very large chunks of memory can be allocated by this implementation, and it has a great advantage that these chunks are returned to the system immediately when they are freed. In our implementation, we create an ordered array of pointers, which is usually very large. It holds the address of the holes in the memory in that ordered array, and all the elements are sorted by the address of the holes. The area where the ordered array is maintained is generally called as heap index, as it indexes all the holes in the heap.

Whenever a memory of a certain size is requested through the alloc algorithm, we go through the heap index. Each element in the index, is the starting address of a hole. As mentioned earlier, we place the header struct at the start of the hole. So from a field of that struct we can know the size of that hole. If the size of that hole is found to be more than the requested size, we can check other required parameters like if we can return the requested size memory, making the block to be page-aligned. If we find that the hole size is far larger than the requested size, then we will split the allocated area into two holes, and the first hole of the required size is returned to the user.

### 6.2.4 Freeing memory

Whenever a block of memory is freed, it is made as a hole by writing the appropriate fields in its header as well as the footer.

An index pointing to the new hole is also added to the heap index. Fragmentation

occurs if we just free only a certain block and make it as a hole. So, whenever we free a block we check with the previous memory chunk and the next one, to see if we can make a larger hole. We check if the previous memory chunk is a hole or not, by using the footer of the previous chunk, which lies just before the header of the present block. If we know that it is a hole, then we unify both of them. We call this iteratively to unify all the left side free memory. Similarly, the header of the next memory chunk is placed right after the footer of the present memory chunk. So, we traverse to the header of the next memory chunk and then see if that area is a hole. If is a hole, then we unify right side. We do this for each free, so in general a single left or right unify is sufficient to create a larger hole, if there is a possibility.

This allocator was written for the kernel, completely in Rust. In this chapter we conclude the thesis and present the future directions.

# Chapter 7

# Conclusions

Our implementation is maintained at

```
git@bitbucket.org:casl/oxide.git
```

## 7.1 Scope for further work

As mentioned earlier Mirage OS is one of the unikernel systems currently available which is completely built in OCaml language. It uses garbage collection for memory management, which can't be used for low-level system development. Mirage OS therefore uses a small kernel called Mini-os written in C, which has the system level code. The kernels written in Rust provide a great hope for replacing Mini-os. If we can also develop modular drivers like network stack and file system stack, we can complete a full Unikernel stack to be used in the cloud.

In the long term, we hope the community will find this system useful and will contribute to the project to solve large practical problems.

# Appendix A

# Setup

We need the following things to run this kernel: Qemu, the virtual environment setup, ld, the linker, nasm, the assembler for x86 assembly, and a nightly compiler for Rust. To compile, we can simply run the command 'make' and to run the kernel, we have 'make run' command.

Unfortunately we have to use a lot of features that are not quite stable as yet, so we use a nightly compiler as it allows the usage of a lot of unstable features which are expected to be standardized soon. So, installing the stable compiler doesn't work, and we need a nightly compiler. There is also a beta compiler, but we won't be using that either. If we install individual compilers, we can't switch between the compilers and so we need an additional tool called multirust. Using multirust we can switch between the type and version of compilers seamlessly. If you have an already installed version of Rust, then you need to un-install that version completely before installing multirust. With multirust we can use different type of compiler in each directory we work. It is enough if we run the command 'multirust override nightly', once in the current directory to use nightly compiler in that directory.

# References


[1] Anil Madhavapeddy, David J. Scott. "Unikernels: Rise of the Virtual Library Operating System. (English) [On Distributed Computing]". In: *ACM Digital library* 11.11 (2014).

[2] Nergal. *The advanced return-into-lib(c) exploits*. URL: `http://phrack.org/issues/58/4.html`.

[3] Doug Lea. *A Memory Allocator*. URL: `http://g.oswego.edu/dl/html/malloc.html`.

[4] OSDev Wiki Community. *OS Dev Wiki*. URL: `http://wiki.osdev.org/Main_Page`.

[5] Phil Opperman. *A minimal kernel*. URL: `http://os.phil-opp.com/multiboot-kernel.html`.