A Kernel-less System for Fast Application Control of Low Level Resources

Authors: Hamidreza Bazoubandi, Francesco Allertsen

Supervisor: Prof. Herbert Bos
Second Reader: Tomas Hurby

A thesis submitted in fulfilment of the requirements for the degree of Master of Parallel and Distributed Computer Systems in the System Group Department of Computer Sciences

August 2013
Declaration of Authorship

I, Hamidreza Bazoubandi, Francesco Allertsen, declare that this thesis titled, ’A Kernel-less System for Fast Application Control of Low Level Resources’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:
Abstract

Faculty of Sciences
Department of Computer Sciences

Master of Parallel and Distributed Computer Systems

A Kernel-less System for Fast Application Control of Low Level Resources

by Hamidreza Bazoubandi, Francesco Allertsen

Traditionally, operating systems are implemented as a privileged layer of software, called the kernel, running underneath the user applications. This setup prevents applications from directly accessing the hardware and makes development and portability of applications simple. However, there is an increasingly large set of applications for which this is no longer a benefit. They cannot rely on the generic mechanisms of the underlying operating system as it does not have enough knowledge to provide optimal service to them. The kernel—in its zeal to shield the hardware from such user applications—instead causes suboptimal execution. In this paper, we advocate a design for multicore chips which removes the operating system from the cores that run such applications. Rather, the operating system will run beside them on other cores, together with legacy applications. We use virtualization to provide isolation for the new class of applications and we show that these applications can take advantage of the new environment to make better decisions than the operating system for sensitive tasks while being still able to use services of the operating system for other generic tasks. We evaluated our design and show that it achieves low latency and reduces timer jitter to fewer than 20 machine cycles, which is two orders of magnitude better than Linux can do. For certain real-time applications, this short, predictable latency is of great value.
Acknowledgements

We would like to kindly thank all people who helped us throughout this project. Special thanks to our supervisor Prof. Herbert Bos, and the second reader Tomas Hurby, without whom this project could never be done. We also would like to thank Dirk Vogt who generously shared his knowledge and experiences with us, and also Andrei, Remco, Aggelos who patiently hosted us in their office.
Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

List of Figures vi

List of Tables viii

Abbreviations ix

1 Introduction 1

1.1 Operating System as Process Isolation Provider 1

1.2 Operating System as Resource Multiplexer 3

1.3 Discussion 4

1.4 Contributions 5

2 Operating System Designs 6

2.1 Monolithic systems 7

2.2 Microkernel-based systems 8

2.3 Exokernels 9

3 Design of Savannah 11

3.1 The system 13

3.2 Message passing 13

4 Implementation 15

4.1 Cpuinfo Structure 15

4.2 Booting Savannah 18

4.2.1 GRUB 19

4.2.2 Stage1 20

4.2.3 Stage2 20

4.3 Virtual Machine Monitor (VMM) 23

4.3.1 VM Environment 25

4.4 Inter Process Communication 25

4.4.1 Message Passing 26
4.4.1.1 Architecture ........................................ 26
4.4.1.2 How to use the API ............................... 28
4.4.2 Channel Communication .............................. 29
  4.4.2.1 Architecture ...................................... 30
  4.4.2.2 How to use the API ............................... 31
4.5 Process Manager ......................................... 32
  4.5.1 PM vs. MM ........................................... 32
  4.5.2 Memory Manager ..................................... 33
    4.5.2.1 Two Step Page Translation .................... 34
    4.5.2.2 Cache Policy ................................... 35
    4.5.2.3 Memory Chunk (De)Allocation .................. 35
  4.5.3 Process services ................................... 39
    4.5.3.1 fork() ........................................... 39
    4.5.3.2 exec() ........................................... 42
    4.5.3.3 exit() ........................................... 45
    4.5.3.4 waitpid() ....................................... 45
4.6 File System ............................................... 46
  4.6.1 Implementation ..................................... 46
  4.6.2 Creation ............................................. 47
4.7 File Server ............................................... 48
  4.7.1 File Server services ............................... 48
    4.7.1.1 open() ......................................... 49
    4.7.1.2 read() ......................................... 50
    4.7.1.3 write() ......................................... 50
    4.7.1.4 close() ......................................... 51
    4.7.1.5 load() ......................................... 51
4.8 User Library ............................................. 51
  4.8.1 Driver in User Library .............................. 52
  4.8.2 Network Stack ...................................... 52
  4.8.3 Special Thread Library ............................. 53
4.9 Drivers .................................................... 54
  4.9.1 Console Driver ..................................... 57
  4.9.2 Keyboard Driver .................................... 58
  4.9.3 PCI Driver .......................................... 61
  4.9.4 E1000 Driver ........................................ 62
4.10 Source Code .............................................. 64
4.11 Compilation .............................................. 65

5 Evaluation .................................................. 68
  5.1 Inter Process Communication .......................... 68
  5.2 Interrupt latency ..................................... 69
  5.3 High Frequency Trading ............................... 72

6 Conclusions ............................................... 75
List of Figures

2.1 Different system architectures ........................................... 6

3.1 Architecture of Savannah – The thick arrows denote VMMs starting VMs, solid arrows represent pre-established communication, dashed arrow is for communication on demand and dotted arrows show direct access of each VM to the raw hardware. ........................................... 11

3.2 Life cycle of a VMM and its VM - (a) idle VMM, (b) VMM starts a new VM, (c) VM runs and VMM does not interfere, (d) VM exits and VMM takes over again and reports to PM ........................................... 11

3.3 Direct access to devices vs. access via the system .................. 12

4.1 Grub loads modules into the main memory ............................ 19

4.2 Steps of the booting procedure ........................................... 21

4.3 Code of a VMMs’ main loop .............................................. 23

4.4 Life cycle of a VMM and its VM ........................................... 24

4.5 Inter Process communication using Message Passing ............. 27

4.6 Creation of the channel .................................................... 30

4.7 (a) EPT violation scenario. (b) Page fault scenario ............... 34

4.8 Memory Layout After Booting ............................................ 36

4.9 Memory Layout ............................................................... 36

4.10 Memory allocation steps ................................................... 37

4.11 Fusion of memory blocked ................................................. 38

4.12 Pseudo code of forks’ wrapper function .............................. 40

4.13 Routine fork_internal sends the beginning address of register vector to the PM ........................................... 40

4.14 Stack layout of a forking process ........................................ 41

4.15 Routine exit() sends the termination request along with the exit value to the PM ........................................... 45

4.16 File System layout ............................................................ 46

4.17 Structure of the header .................................................... 47

4.18 Structure of the file descriptor ........................................... 49

4.19 Network stack per process combined with NIC supporting virtualization ........................................... 53

4.20 Networking in Savannah if NICs do not support virtualization ... 53

4.21 (a) Steps required to handle a key stroke in MINIX3 (b) Steps required to handle a key stroke in Savannah ........................................... 56

4.22 Communication between applications and console driver ........ 57

4.23 Pseudo Code of a TTY driver in MINIX3 .............................. 58

4.24 Snippet of code from keyboard driver in Savannah ............... 60

4.25 Main function from keyboard driver in Savannah ................. 61
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.26</td>
<td>Simplified code of the e1000 driver main function</td>
<td>63</td>
</tr>
<tr>
<td>4.27</td>
<td>Source code structure</td>
<td>64</td>
</tr>
<tr>
<td>4.28</td>
<td>Potentially vulnerable code to optimization</td>
<td>66</td>
</tr>
<tr>
<td>5.1</td>
<td>Jitter of Savannah, Linux and Linux-RT</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Breakdown of latency in HFT benchmark</td>
<td>73</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Source code statistics .................................................. 64

5.1 Comparison of IPC notification in Savannah ....................... 69

5.2 Comparison of jitter of timer functions in Linux, Linux-RT and Savannah (in µs) ............................................. 70
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APIC</td>
<td>Advanced Programmable Interrupt Controller</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>EPT</td>
<td>Extended Page Tables</td>
</tr>
<tr>
<td>FD</td>
<td>File Descriptor</td>
</tr>
<tr>
<td>FS</td>
<td>File Server</td>
</tr>
<tr>
<td>GDT</td>
<td>Global Descriptor Table</td>
</tr>
<tr>
<td>IDT</td>
<td>Interrupt Descriptor Table</td>
</tr>
<tr>
<td>IO</td>
<td>Input and Output</td>
</tr>
<tr>
<td>IOAPIC</td>
<td>Input/Output Advanced Programmable Interrupt Controller</td>
</tr>
<tr>
<td>IPC</td>
<td>InterProcess Communication</td>
</tr>
<tr>
<td>IPI</td>
<td>Inter Processor Interrupt</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt ReQuest</td>
</tr>
<tr>
<td>LAPIC</td>
<td>Local Advanced Programmable Interrupt Controller</td>
</tr>
<tr>
<td>MMIO</td>
<td>Memory Mapped IO</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PM</td>
<td>Process Manager</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric MultiProcessor</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Monitor</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Operating system is the most important piece of software running on any computer. The performance and the behavior of the operating system affects all applications running on the computer. Therefore, it’s very important that an operating system provides an environment in which all user applications can run with the maximum performance and efficiency.

Traditionally operating systems play two major roles in a computer:

- Provider of process isolation
- Resource multiplexer.

In this chapter, we are going to argue that traditional operating system designs do not give all applications equal chance to run with the maximum performance, and therefore we believe with all new advances in technology, the definition of operating system role in a computer should be rethought and we try to draw a new role for the next generation of operating systems.

1.1 Operating System as Process Isolation Provider

Operating systems provide an API to their user applications and shield them from each other. In order to achieve this goal, CPU provides two different execution modes: Less privileged mode which is also known as user mode, and a privileged mode, also known as kernel mode. In an operating system, user applications run in less privileged mode, which means they are not allowed to use some features of the underlying CPU. These features are reserved only for privileged software: the kernel of the operating system. Kernel
can change the execution environment and directly interact with external devices using these privileged CPU features. Depriving user applications from accessing privileged features guarantees that the applications can safely share the hardware resources. Thus, the operating system serves as the arbiter of who is allowed to use what and when.

But this generic definition for the role of an operating system implies an environment that is a compromise between the needs of various types of applications. This might be fine for a large range of applications, but it is rarely optimal for all of them and especially not for applications with unusual and specific requirements. For instance, real-time applications suffer from general purpose operating systems’ desire to keep everybody happy. The jitter on timers – of critical importance for real-time systems – on general purpose systems can be tens of thousands of cycles. Likewise, applications that have strict demands on latency of network processing (like high frequency trading applications) cannot easily run on a general purpose system that does not guarantee low and predictable latencies. If it takes more than 10 microseconds to transfer a packet from the network card to the application, it is much too long for some applications.

Operating system developers attempt to mitigate the issue by providing some knobs that applications can turn or parameters they can set to influence system behavior. Nevertheless the system largely remains a compromise. In all traditional operating system designs, a process is an opaque entity – a black box – which the operating system observes. Subsequently it makes its decisions based on these observations.

As a matter of fact, only the applications themselves know what their requirements are and how to achieve their goals. Applications want the operating system out of their way when they do not need it. They may even want guarantees that they can run without preemption at critical moments. They may want guarantees of low jitter and low latency. In such cases, purpose-built systems [1–3] and lightweight runtime environments [4] step in. Unfortunately, developers have to pay an increased price for porting their applications to such systems and they lose convenience in places where the performance is not critical and where simple and standard programming interfaces would be great assets.

With the advent of multicore processors and the (re)invention of virtualization, we believe that it is possible to design operating systems in such a way that they can provide the best from both worlds – general purpose interfaces and dedicated resources with low level access to them.

Giving an application its own core or cores has the benefit of no interference by other applications. This has been possible in many operating systems for a long time. However, the application still needs assistance from the kernel to schedule its threads, setup timers, receive external events and interact with hardware. Soares et al. [5] demonstrated that the
interaction with the operating system itself, by the means of system calls, has a negative impact on the optimal performance of applications. For instance, their experiments proved that a single \texttt{write()} system call in Linux can evict up to $\frac{2}{3}$ of L1 cache and TLB entries. This would cause a significant performance penalty both for kernel and the user applications. They proposed that it is better to run user applications on different cores from the operating system in order to prevent the user applications and the operating system competing for the CPU high performance data structures such as cache and TLB.

Once the application has its own core(s), it can perform scheduling of its threads the way it wants to. For instance, it can decide to let each thread run undisturbed as long as it can and context switches only when a thread makes a blocking call. Similarly it can decide that some threads are preemptible and some are not, or that some threads must run immediately when they become runnable. Conventional operating system designs do not have the application-specific knowledge to do this.

To implement efficient low latency scheduling, the application must have access to efficient timers and interrupt handling. In general, operating systems do not offer that and handle both timers and interrupts themselves. This means that setting a timer requires a system call and the system makes an expensive up-call to deliver the event. The system hides all other interrupts and external events by higher level APIs, for instance, for reading and writing files.

Belay et al. [6] demonstrated that some applications can benefit from running within a virtual machine environment which provides the same isolation as ordinary process, while it provides the applications access to features of the CPU that traditionally only the operating system can access. However, the application still executes as an ordinary process which shares a multiplexed CPU with other applications on top of the Linux kernel. In addition, this design dramatically increases the overhead of accessing to the operating system services since for every system call the process first needs to exit the virtual machine, and only then trap to the kernel as an ordinary process and after receiving the response from the operating system it has to enter the virtual machine again before being able to resume its execution.

### 1.2 Operating System as Resource Multiplexer

Traditionally, resources were expensive, limited and therefore it was a must to share resources among multiple processes. This is not true anymore, since computing resources are becoming cheaper, more powerful, and more abundant. We now have large amounts of memory and CPU cores are becoming plentiful. Therefore we can relax the hard
requirement of multiplexing and dedicate some resources for exclusive use of a single application.

As virtualization is penetrating into many areas of computing machinery, the devices take over the role of the operating system as the multiplexer. Therefore we can give an application the illusion that it is the exclusive owner of a resource without the kernel acting as the middleman. The hardware does it instead and since the hardware does it more efficiently, the remaining role of system software is to configure the devices. It does not take program execution. This is in contrast to conventional designs, where hundreds or even thousands of user-to-kernel mode context switches are needed to handle system calls. Network cards have pioneered this approach and other types of devices follow [7]. However, this is not a requirement. If there are multiple devices of the same kind in the machine, dedicating some to an application is also an option. Similarly, it is also fine when there is no other application that needs a particular device.

The hardware support for virtualization in new processors provides much more flexibility in terms of protection and isolation compared to the legacy ring protection mechanism in the x86 architecture. That architecture has a fixed set of features accessible in each ring. On the other hand, virtualization allows much finer grained control over what an application can access and what is prohibited. For instance, it is possible to selectively grant or restrict VM access to some privileged instructions such as HLT or MONITOR/MWAIT, whereas in the ring protection scheme, there is no way to give a process in unprivileged mode direct access to such instructions, even if there are no security and isolation violation concerns. For instance, letting an application halt a CPU is possible when there is no other application that shares the core and thus it would not be able to use it.

1.3 Discussion

With all these advances in hardware virtualization, the role of the kernel as a provider of process isolation and resource multiplexer is fading. We argue that future operating systems do not need to have a kernel per-se. A thin layer that runs only when it starts a process and after it finishes to clean up is the only privileged software we need.

In this Master thesis we propose a new design for the operating system which adopts some of the ideas about the novel use of virtualization from Dune [6], combine them with ideas from variety of other systems and add some new ones. The resulting system, Savannah, represents an extreme point in the design space of operating systems with excellent support for meeting demanding application-specific requirements.
1.4 Contributions

Major contributions our works are the following:

1. We reduced timer jitter from (tens of) thousands of cycles to fewer than 20.

2. We can deliver hardware events (such as interrupts) to applications with extremely low latency.

3. We demonstrate latency of the network stack of less than 1 microsecond.
Operating System Designs

Operating systems come in many kinds and flavors. In this section we look at successful system designs and discuss their flaws to motivate the need for rethinking current designs in the light of new hardware and new applications.

![Diagram of different system architectures](image)

**Figure 2.1:** Different system architectures
2.1 Monolithic systems

The most popular and widespread architecture is the so-called monolithic system. It is used in many commodity operating systems such as Linux, Windows, BSD, and more. The key concept in this architecture is that the whole operating system is a single software entity, called the kernel, running in the privileged mode of the CPU. User applications, as depicted by Figure 2.1(a) run ‘on top’ of this kernel. It is a successful, well-studied model. Its primary advantage is high performance. Since the entire system is a single piece of code running in a single address space, there is minimum overhead in communication within the kernel; all data structures are shared and accessible by any part of the code which needs to use them. However, this approach also has its downsides. First, a single bug in one line of code in a minor device driver can bring the entire computer to a grinding halt. Second, as multicore chips become the norm, the kernel has to protect the shared operating system data structures from concurrent access. Fine grained high performing locking schemes are complex, difficult to understand and error prone.

Since the applications run ‘on top’ of the kernel, whenever they request a service from the system, they ‘trap’ into the kernel and the kernel executes on their behalf on the same CPU. As demonstrated by Soares et al. [5], this performs suboptimally due to the many context switches, which negatively affect the performance of the TLB, cache, and other items. To eliminate these context switches, in our design we run the operating system on different cores than the applications.

Due to their monolithic nature, such systems provide only a single set of policies for all applications, which never suits them all. It is possible to configure the system when building it or use various run-time knobs to tweak it, but the one-size-fits-all model is still never ideal for all applications. This is especially true if the applications have radically different needs and hard requirements, for instance, real-time applications. The need to deal with stringent application requirements can even lead to forks of the software. An example is RTLinux [8], a derivative of Linux for real-time applications. Although RTLinux is an improvement for real-time, it is far from being a system which can safely provide real-time guarantees as well as run ordinary processes. For the real-time tasks to meet the deadlines, RTLinux runs them as modules within the kernel.

This is a shortcut for the tasks to reach devices, however, it is a security and reliability nightmare. The real-time tasks are not isolated from the rest of the system and any bug is a potential threat to the stability of the entire system.

RTLinux is a discontinued project, but there are other similar active projects like Xenomain [9] and RTAI [10]. Both projects take Linux and modify it to introduce a layer that intercepts interrupts and scheduling to give better treatment to real-time tasks.
There is an initiative to turn mainline Linux into a real-time capable system \[3\]. Although this effort improved many sensitive parts of Linux, it proved to be too disturbing and controversial that only parts of the patch were merged in the mainline and the authors of the real-time patch \[11\] still maintain it outside the Linux source tree.

### 2.2 Microkernel-based systems

Microkernel-based systems address some of the downsides of the monolithic system design. They build on a thin layer of privileged code, the microkernel. The rest of the operating system functionality is implemented by multiple processes running in user space.

Some systems run the operating system in user space as a single process, often as a paravirtualized instance of a monolithic system (e.g., L4Linux \[12\], L4Android \[13\], etc.). This setup makes it possible to run legacy applications as clients of the monolithic system while drivers or real-time or high-performance applications can run as native processes of the microkernel. It is a setup similar to RTLinux with the big advantage of isolation and protection of the real-time code. As multicore processors are abundant, such systems can run the real-time and latency sensitive code on a dedicated core where the rest of the system does not interfere. Nevertheless, the microkernel on such cores still prohibits direct access to some hardware features like device interrupts. In addition, when such an application needs access to some operating system service, it uses the kernel to pass the requests to the system and to deliver its reply.

Other systems like HelenOS \[14, 15\], QNX \[16\] and MINIX 3 \[17\] stretch the microkernel design to an extreme and implement the operating system as a collection of user space processes, both servers and drivers, hence we call such systems multiserver systems. This design, presented in Figure 2.1(b), allows large degree of flexibility since the system is not a single component. It is possible to replace parts of the system according to the current needs or have the same functionality, for example, scheduler or memory manager, running in several instance to provide for each type of applications the optimal service they need. This has important reliability advantages. However, it also has performance impacts which are possible to overcome using multicores and dedicating cores to system processes. The authors of NewtOS \[18\] also argue for minimal role of the kernel, however, due to the fact that ring protection does not allow the fine grained isolation settings, they must use the kernel in some corner-case situations. In this thesis we show that our system avoids this.

These processes need to act with low latency. Drivers, especially, can benefit from direct access to the hardware. In the current multiserver systems, it is possible to implement
a driver in user space as devices are largely memory mapped. However, the operating system still handles interrupts and delivers them with a delay to the driver. Moreover, the driver receives the interrupts by the means of a message only when it explicitly asks for it. This can further delay its delivery. At the end of the day, the microkernel running on a driver’s core only handles interrupts and the driver uses it to block as it cannot stop its dedicated CPU on its own. We argue that the microkernel, albeit tiny, still presents clear overhead. With current hardware and our design we can do better.

A multikernel [19] is similar to a microkernel, however, it does not run as a single kernel across all cores in the system; rather it runs a separate kernel for each individual core (Figure 2.1(c)). This design scales better since there is no contention in the kernels and communication between the kernel is never by sharing, but by the means of explicit message passing.

### 2.3 Exokernels

In some ways, the closest existing system structure to ours is an exokernel [20]. As Figure 2.1(d) shows, this approach eliminates most of the kernel as the remaining kernel is responsible only for isolating individual applications and separating the hardware resources they need while letting the applications access as much of the raw hardware as possible. It is usually accompanied by a library which makes use of the raw hardware more pleasant for the programmer without hiding any details. In addition, exokernels usually provide a rich library which offers an API like POSIX to make development and porting of applications from other systems simpler. This library, linked with the applications forms a so-called library operating system. Such a design is popular in the embedded world. Nevertheless, without virtualization, the kernel is still active as the applications cannot have full direct access to all features of the CPU nor to other peripheral devices due to the fact that they could escape from the isolation.

One of the first exokernels that took advantage of virtualization was the SUN hypervisor [21] shipped in the SUN’s SPARC T-series servers, also called Niagaras. Since these processors have an additional hyperprivileged level, the CPU can safely execute the vast majority the privileged instructions without trapping into the hypervisor. Although the hypervisor is part of the SPARC v9 architecture, it largely stays out of the way of normal software execution. It only initially partitions the machine and virtualizes access to the cores’ TLB and a few more very low level functions. With the progress of the Intel virtualization and since the x86 TLB is filled automatically by the hardware, it is possible to push the virtualization software even further away. The disadvantage of the SUN hypervisor is static partitioning of the machine, which is quite limiting in case the
workload changes and the mix of and requirements of the applications as well. Since our design is not part of the architecture specification, rather, the virtualization is adopted as an extension of our operating system, it allows fully dynamic assignment of memory and cores at runtime.
Chapter 3

Design of Savannah

Currently, our system, called Savannah, targets the 64-bit x86 architecture. However, this approach is applicable to any virtualizable architecture. Figure 3.1 shows the architecture

Figure 3.1: Architecture of Savannah – The thick arrows denote VMMs starting VMs, solid arrows represent pre-established communication, dashed arrow is for communication on demand and dotted arrows show direct access of each VM to the raw hardware.

Figure 3.2: Life cycle of a VMM and its VM - (a) idle VMM, (b) VMM starts a new VM, (c) VM runs and VMM does not interfere, (d) VM exits and VMM takes over again and reports to PM.
The key component of the system is a Virtual Machine Monitor (VMM). Each core runs a single VMM, much like the Barrelfish multikernel [19] runs a single kernel on each core. The key principle is that all code of the operating system itself as well as all applications run within a Virtual Machines (VM). The "guest operating system" (really just a library) and the application can be linked together (as in the exokernel model) and run in kernel mode. In this case, hardware resources dedicated to this core can be directly accessed, without any interference from the VMM. This is the case of the VM running on the core \( W \) in Figure 3.3. Alternatively, there may be some cores running a legacy guest operating system and supporting multiple user processes. Any mixture is possible, something not doable with the exokernel model. In addition, some of the central components of the operating system, such as the process manager and file system, can run on dedicated cores with their own VMMs and protected from applications and each other. The second part of Figure 3.3 shows a user VM on the core \( Z \) which uses the network stack (core \( Y \)) and a driver (core \( X \)) to access the network. In this case, only the driver has direct access to the network interface card.

Although it may seem like the VMM serves the same role as a kernel in other operating systems, the VMM has very limited number of responsibilities. Its role resembles, to a great extent, the role of an exokernel. It is only responsible for isolation, launching VMs and handling their crashes. Unlike an exokernel, once a VM is launched, VMM is out of its way and does not interfere during in the execution of a VM. We configure the VMs so that the code inside can do also anything it could do on a dedicated raw hardware. The VMM takes over the control of the core only in case the code inside the VM does something wrong. It causes an exit from the VM and VMM reports it as an error to PM and does not let the VM run any more. Figure 3.2 shows the life cycle of both a VMM and its VM. The dashed border of the VMM in Figure 3.2b denotes that the VMM is present, but idle. Since only one VM can run on each core, there is no need for VM exits due to scheduling multiple VMs on a shared core. In addition, the VMs communicate by the means of message passing, using mailboxes and channels in memory shared between
them without any assistance from the VMM as the communication is created once when launching the VMs.

By default, there is one VMM per core. However, it is possible to span a VMM across multiple cores to support applications that run multiple threads and benefit from parallelization.

Using the VMs to implement processes has the following benefits, which are key to the success of our design:

- Direct access to virtualized low-level resources
- Dedicated core without context switching
- Direct delivery of interrupts and exceptions

### 3.1 The system

For implementing the operating system’s functionality, we adopted the multiserver approach. Currently, we have two operating system servers, each on a separate core running a single thread. They are the process manager (PM) and file server (FS). The former is responsible for managing processes and their memory, while the latter implements an interface to storage and devices. Our system also features independent isolated device drivers. We isolate the drivers since drivers provided by third-parties tend to be the primary source of bugs and problems for operating systems. Applications or specific subsystems (e.g., a generic network stack) can use the service provided by the drivers. On the other hand, a driver is not necessary if an application provides its own driver within the environment of its VM as we allow applications direct and exclusive access to hardware which they request.

### 3.2 Message passing

Since all entities of the system as well as applications run isolated on different cores, the traditional way of communication between the applications and the system by the means trapping from the user space to the system’s kernel is impossible in our case, so we provide intercore message passing.

There are two ways of message passing. One way gives each applications’ VM an inbox and an outbox when the system creates it and VMs can notify each other that it has a
message for it in its outbox. PM maps these mailboxes in its address space and in the FS’s as well. Each application can only write to its outbox and read from its inbox. We call the pair of inbox and outbox a communication channel and they are represented by the solid arrows in the upper section of Figure 3.1.

Since we cannot anticipate what other VMs/processes would like to talk after it starts, we can establish further communications channels on demand. For instance, when an application opens a device, FS sets up a message channel between the application and the device driver. This way, FS does not need to serve as a relay and the communication has less latency without the extra hop. Also if two applications want to talk to each other, they can ask the core system to establish such a channel (the dashed line in Figure 3.1), a shared piece of memory. We present the details on both kinds of the communication in Section 4.4.

Although VMMs do not run during the normal operation of their VMs, they do communicate with PM before they start the VM and also after the VM exits when the process either finishes its work or crashes. Therefore we setup a communication channel between PM and each VMM in the same way.
Chapter 4

Implementation

4.1 Cpuinfo Structure

The cpuinfo is the most important data structure in Savannah that keeps all the information about each CPU. The information stored in this data structure can be put into four main categories:

- information about the processor
- information about the VMM
- information about the VM
- information IPC data structures

The following code snippet shows the definition of this structure.

```c
struct cpuinfo {
    uint8_t lapic_id; // Local APIC ID
    uint8_t cpuid; // Local CPU ID
    volatile uint8_t ready; // Set when the CPU is ready after booting
    struct vm_proc vm_info; // Information about the VM
    struct vmm_proc vmm_info; // Information about the VMM
    void *vm_args; // Arguments passed to the VM
    uint64_t ncpus; // Number of CPUs in the system
    uint64_t cpu_freq; // Frequency of the CPU
    uint64_t bus_freq; // Frequency of the BUS

    /* IPC */
    struct message *msg_input; // Pointer to the inbox
    struct message *msg_output; // Pointer to the outbox
    bool *volatile msg_ready; // Pointer to the notification bitmap
};
```
The processor related information stored in the `cpuinfo` structure includes some general information about the CPUs such as total number of CPUs, the bus and the CPU frequency. It also includes two fields for the unique IDs that the CPU can be identified with. There are two different fields for the CPU ID in this data structure. The `lapic_id` field contains the ID assigned to the core by the processor. This field is very CPU specific, because some processor assign IDs that can not be changed by the software, and some allow the IDs to be assigned. Therefore, we considered another field named `cpuid` that contains the ID assigned to the core by Savannah. Both of these fields are used extensively in our source code. In some sections of the operating system where we program the hardware to interact with a particular core (for example to send an IPI to the core), we need to use the `lapic_id` field, because that is the ID with which the hardware addresses that specific core. But in other parts of the operating system for example when the PM send a message to a VMM, `cpuid` field is used.

The other two important fields in the `cpuinfo` structure are `vm_info` and `vmm_info` that contain all the information about the VM and VMM belonging to the CPU. The `vm_info` and `vmm_info` are of types `vm_proc` and `vmm_proc` respectively. The definition of these types are the following:

```c
struct vm_proc {
    struct regs vm_regs; // VM’s registers
    phys_addr_t vm_start_paddr; // VM’s physical start address
    phys_addr_t vm_end_paddr; // VM’s physical end address

    virt_addr_t vm_start_vaddr; // VM’s virtual start address
    virt_addr_t vm_end_vaddr; // VM’s virtual end address

    /* Physical addresses of the sections */
    phys_addr_t vm_code_paddr;
    phys_addr_t vm_data_paddr;
    phys_addr_t vm_rodata_paddr;
    phys_addr_t vm_bss_paddr;
    phys_addr_t vm_stack_paddr;

    /* Virtual addresses of the sections */
    virt_addr_t vm_code_vaddr;
    virt_addr_t vm_data_vaddr;
    virt_addr_t vm_rodata_vaddr;
    virt_addr_t vm_bss_vaddr;
    virt_addr_t vm_stack_vaddr;

    size_t vm_code_size;
    size_t vm_data_size;
    size_t vm_rodata_size;
    size_t vm_bss_size;
    size_t vm_stack_size;

    phys_addr_t vm_page_tables;
    phys_addr_t vm_ept;
};
```
The last category of the information in the `cpuinfo` structure belongs to IPC. Both VMM, and VM running on a CPU need to communicate with the other components in the
system. Therefore, apart from the IPC channels created for the VM that we will discuss about in details in Section 4.4.2, there are some IPC data structures that belong not only to the VM, but also to the VMM. These data structures are created during the boot process as Section 4.2.3 explains in more details.

Each `cpuinfo` structure is allocated within a 4KB memory page. This is necessary to guarantee the privacy and security of the information in the structure. The `cpuinfo` of each core, is only accessible to the VM, and VMM running on that same core. More specifically, the VM, and VMM, have read-only, and read-write access to the information available in the `cpuinfo` structure respectively. In addition, operating system servers have read-write access to the `cpuinfo` structures of all CPUs.

### 4.2 Booting Savannah

The goal of booting process is to setup the environment for the rest of the operating system to begin execution and serving the request. Therefore, the design and complexity of the booting process directly relates to the design of the operating system. Often monolithic operating systems have a simple booting process because, in a monolithic design, all the operating system services are accumulated in the kernel, and as soon as the kernel is loaded in memory and its execution environment is set up, it can begin serving the user requests. Microkernel operating systems however, need a more complex booting process, because in such operating systems, the kernel itself is very minimalistic and simplistic and does not provide enough functionalities for the system to stand by itself, and therefore, to boot up a microkernel operating system, in addiction to loading the kernel, operating system servers must be loaded into the memory as well, and the environment must be set up such that these components can cooperate with each other before the whole system can begin functioning.

Booting Savannah has some similarities to booting microkernel operating systems, however, Savannahs’ booting process is more sophisticated and is comprised of the following steps that we are going to explain in detail in this section:

- GRUB
- Stage1
- Stage2
4.2.1 GRUB

GRUB [22] is a powerful, free, and open source boot loader widely used by many operating systems like Linux, BSD, and etc. Grub is highly portable, and configurable, and provides very nice features that make it interesting for boot process designers. We chose GRUB for the very first step of booting Savannah for two reasons: first, it collects and provides some basic information about the hardware features such as the size of available memory that otherwise require large amount of effort to acquire, and second, it facilitates loading operating system components. That is GRUB has the capability of loading some modules along with a special module (GRUB calls it kernel) which will take the control of the processor immediately after the GRUB finishes its job. In our case, the kernel that GRUB loads in the main memory is the stage1 boot program explained in Section 4.2.2. Additionally, as Figure 4.1 shows, it loads some other essential components of Savannah which are:

- ELF file of stage2 boot program
- ELF files of the main servers: PM, FS and INIT
- ELF file of VMM
- Ramdisk

Grub runs on the first core of the first processor known as the bootstrap core or the bootstrap processor. It begins its execution in 16-bit real mode and then switches to 32-bit protected mode. Therefore, the special module that GRUB loads to take the execution control after GRUB, must start in 32-bit protected mode. That is exactly what the boot stage1 program does.
4.2.2 Stage1

After the GRUB stage is done, it moves the control to the program (the kernel in GRUBs’ terminology) it already loaded to deliver the control to when it finishes the job. We named this program the first boot stage or boot stage1. The boot stage1, is written compiled, and linked to function in 32-bit protected mode. It’s a rather small program that has the following responsibilities:

- taking the addresses of all loaded modules from grub, and move them into the pre-defined memory locations
- check to make sure that all required CPU features and extensions are available
- check to make sure cache is enabled, and enable it if it is disabled
- Load the second boot stage program by reading, and parsing its loaded ELF file, and copying program segments in the appropriate virtual addresses mentioned in the ELF header.
- get some basic information from grub such as memory size
- give the control to the stage2 boot loader and pass in the required parameters include the available memory size, and the start addresses of all operating system components (servers, drivers, ...).

After performing all the aforementioned steps, the boot stage1 delivers the execution control to the second boot stage program that we are going to explain in detail in the Section 4.2.3.

4.2.3 Stage2

The boot stage2 program is the final step in booting Savannah. Therefore, it has to build up the appropriate execution environment for the rest of the operating system to begin functioning. This role defines a very broad set of responsibilities for boot stage2 ranging from detecting and booting other cores to running a VMM on each core, to loading and running operating system servers. As a result, the second boot stage is a large program of more that 3000 lines of code.

The boot stage2 takes the execution control from the first boot stage program in 32-bit protected mode. The very first parts of the boot stage2 is written in x86 assembly language. This short assembly file first maps 1GB of memory with 2MB pages. This
is essential because the boot stage 1 was operating on the memory setup that had been created by the GRUB, and before switching to the 64-bit mode, paging must be setup and enabled. Therefore, one of the first actions taken by this assembly code is to create page tables and then switch to 64-bit mode. Additionally, as the last step before jumping to the C code, a new stack is created by this assembly code.

After moving to 64-bit C code, as Figure 4.2 (a) shows, the boot stage 2 detects all cores available on the system. At this point of its execution, the boot stage 2 program allocates 3 pages for each cpu, and assigns the start address of each of them to the msg_input, msg_output, and msg_ready fields in the cpuinfo data structure. These pages are later
used by VMs, and VMs for IPCs usages. We will explain this in more details later in Section 4.4.

At this point of its execution, the boot stage parses the ELF file of the VMM loaded by the GRUB into the main memory. Then, as Figure 4.2 (b) depicts, the boot stage creates the VMM process instances one per each core. The boot stage stores the information about the VMM process instances such as the virtual and physical start addresses of all process sections, the the status of the process, and any other kind of information about the VMM in \textit{vmm\_info} field of the \textit{cpuinfo} structure (explained in Section 4.1).

In addition, the boot stage parses the ELF files and creates the process instances of all other essential components of the operating system such as PM, FS, INIT; Figure 4.2(c) shows this step. The boot stage stores all information about these VMs in the \textit{vm\_info} field of the \textit{cpuinfo} structure. In addition, The boot stage would map the \textit{msg\_input} page with read-only permission, \textit{msg\_output} and \textit{msg\_ready} pages with read-write access for all VMs, and VMs that it creates.

After all operating system components are loaded and ready in memory, the stage boots up all other cores one by one by sending an INIT IPI to the core. After receiving the IPI, a core starts operating in 16-bit real mode, and must be taken to the 64-bit mode by the operating system. To this end, before sending the IPIs, the boot stage, loads a short assembly code, which is in \textit{boot/stage2/boot\_aps.S}, in a predefined memory address, and forces all the cores to jump to that address and start their execution from there after receiving the IPI. The goal of this code is to boot up the core to the 64-bit mode, and then run the VMM instance created for that core on top of that. To this end, it receives as the parameter a pointer to the \textit{cpuinfo} structure belonging to this core, which contains all the information regarding the VMM including the start address of the page tables, stack, and code section. Thus, before sending each IPI, the boot stage sets the parameters belonging to the target core in the appropriate and predefined addresses, and then sends the IPI to the core. After sending the IPI, the boot stage polls on the flag \textit{ready} in the \textit{cpuinfo} structure, which the core will set as soon as it runs the VMM code. Once this flag is set, the boot stage knows that the core is successfully booted to its corresponding VMM, and continues booting the next core.

When cores are booted to their VMMs, all what they do is to wait for a message from the PM. Figure 4.2 (d) demonstrates that the PM is the very first VM which is launched in Savannah. It runs on the bootstrap processes. In fact, As soon as the boot stage is done with booting up all other cores, it changes its address space to the address space of the VMM instance created for the first core, and then executes a long jump instruction to the start address of its VMM. Unlike VMM on other cores which waits for a message
from the PM, VMM on the bootstrap processor immediately launches the PM virtual machine.

As soon as the PM is launched, it sends a message to all VMMs indicating that the PM is now launched and ready to handle requests. This message has also another meaning for the VMMs running on core number 1, and 2. These cores are supposed to run the FS, and the INIT virtual machines, but even though these processes are already created by the boot stage and are available in memory, VMMs are not allowed to launch these virtual machines before the PM is up. Therefore, this message is a green light for these VMMs to launch their VMs. Therefore, after PM sends the message, the FS, and the INIT virtual machines would be launched on core number 1, and core number 2 respectively. This is what Figure 4.2 (e) shows. Other VMMs on other cores will continue waiting until they receive a message from the PM for launching a VM.

4.3 Virtual Machine Monitor (VMM)

VMM is one of the basic components of Savannah that resembles the kernel in multikernel operating systems. As explained in Section 4.2, one VMM runs on each core and they share no data structures with one another. From this viewpoint, the VMM might looks exactly like the kernel in multikernel design, but the roles and responsibilities defined for the VMM in our design largely differ from those defined for the kernel in the multikernel design. The kernel by its very nature is there to be asked for some services and assistance during the execution of user processes. At any time the kernel may decide to interfere, block, or switch processes for various reasons, but the VMM on the other hand does not interfere during the execution time of the user application at all.

```c
while(1) {
    /* Receive the message from PM */
    m = msg_receive(PM);
    /* Launch the VM inside the environment */
    r = launch_vm(m);
    /* If the VM exits, send the exit value to PM */
    msg_send(PM, r);
}
```

Figure 4.3: Code of a VMMs’ main loop

As Figure 4.3 shows, all what a VMM does in Savannah is to receive the informations of a process from the PM, sets up the VT-x parameters according to the information received, and launches the process within a new VM. Although simple, but this plays a crucial role in our design. This is because not all VMs have the same sets of permissions and authorities, and since there is no kernel to monitor and control the behavior of virtual machines while they are running, everything should be set up so that the security
is insured before the virtual machine is launched. This is what the VMM does by setting up the VT-x parameters according to what PM orders. As Figure 4.4 shows, once the VMM launches a new process inside a VM (a), the VMM will be completely out of the way (b), and the process can run on its virtualized processor within a VM isolation without any interference.

The other important role of the VMM in our design is to handle the VM exits and crashes. That is if a VM exists, or crashes for any reason, the VMM will take the control, and will notify the PM about the termination of the VM.

There are few reasons why the control goes from the VMM to the VM (d). The first one is when the VM itself performs a VMEXIT operation. The second one is when the process does something that violates the restrictions and crosses the boundaries set by the operating system. For example, When an application tries to access an address in memory beyond its own address space, an EPT violation happens and the VMEXIT is automatically performed by the processor. Hence the control is given to the VMM, and a register is set that shows the cause of the VM exit.

As the code snippet in Figure 4.3 shows, every time a VM exists, irrespective of the exit reason, the return value is stored in a VMM local variable. The VMM then reads the content of the exit cause registers, and sends it along with the return value of the process (if any) to the PM. The important point to note is that although the VMM is the operating system component that reads the exit cause register, it does not take any action based on that. Instead, all what it does is to send it to the PM, because in our design, under any circumstances, the PM is responsible to handle all VM exit and crashes.

As the code snippet in the Figure 4.3 shows, after the VMM sends the message to the PM, it waits again for a message from the PM. On the other side, the PM also considers the sending VMM as a free one to launch a VM.
4.3.1 VM Environment

As explained before in Chapter 3, Savannah extensively uses the VT-x extensions [23] available in new Intel processors. A similar extension is also provided in the latest AMD processors called SVM [24], however with a different hardware interface than VT-x. Current version of Savannah does not support the AMD SVM technology.

The VT-x technology provides a supplies a wide range of controls that can be exploited to change the range of accesses and permissions given to the VMs, but here we only focus on the features that we mostly used in our design.

To allow, or prohibit asynchronous events such as interrupt deliveries to the VMs, VT-x technology provides a register called *Pin-Based VM-Execution Controls*. This register contains a 32-bit vector that governs the handling of asynchronous events (like interrupts). In our design, the content of this register is determined according to the nature of the process by the PM. For instance, each device drivers needs to receive a specific external interrupt. So, other external interrupts should not be allowed to be delivered to the VM for device drivers.

To specify to what instructions can be executed and accessed by the VM, VT-x proposes another register named *Processor-Based VM-Execution Controls*. There are two 32-bit registers for this purpose: one which is called *Primary Processor-Based VM-Execution Controls*, and the other which is called *Secondary Processor-Based VM-Execution Controls*. The purpose of these vectors is to govern the handling of synchronous events, mainly caused by the execution of specific instructions. For example these registers can prevent the execution of some instructions. In Savannah, by default, some instructions such as HALT, and MONITOR/MWAIT are allowed, but some other instructions such as IN/INS are only allowed for some particular processes such as device drivers.

During a process creation, based upon the privileges and authorities that the process possesses, the PM defines an environment that the process requires for its execution. Later on, the PM sends the specification of the defined environment within a message to the VMM considered to launch the VM. Then the VMM, sets the VT-x registers so that the required environment is set up for the execution of the VM.

4.4 Inter Process Communication

All operating systems that are comprised of different components require IPC between these components. For monolithic operating systems, IPC is very negligible since all operating system services are accumulated inside the kernel. However, in microkernel,
and multikernel designs, several IPCs are required before a user request can be handled. The overhead of these IPCs can significantly slow down the system. That's the reason.

There have been extensive research done to minimize the negative effect of IPC in the overall performance of the system [12, 25–27]. To this date, most of these studies only mitigate the problem, and none of them can be considered as definitive best single best solution.

Savannah inherits many features of microkernel and multikernel operating systems. It pins processes to core(s), and has servers and drivers running as normal processes. This design entails that to guarantee a good performance, Savannah also needs a very fast IPC method.

In Savannah, processes do IPCs for two reasons: first, when they need a request from a server, few messages are exchanged between the process and servers, and second, when processes need to receive data from device drivers, they receive the data through special type of message passing.

We are going to talk about these two IPC methods in the following sections. First we give implementation details of our normal message passing methods in Section 4.4.1, and later on in Section 4.4.2 we talk about the communication method between processes and device drivers.

4.4.1 Message Passing

As explained in Section 4.4, message passing is used just when a process wants to communicate with a server. In this section we are going to explain what is the architectural design of our message passing. Later in Evaluation Section 5.1 we are going to show the performance results of our design.

4.4.1.1 Architecture

Figure 4.5 shows the overall procedure of how a messages are exchanged between an application running on core 3 to a server running on core 0.

The overall architecture is that every user application has two boxes:

- outbox: Where to send the request
- inbox: Where to receive the reply
and each box is a single memory page in memory, which is by default 4KB in size. This means that the maximum size of the message, by default, in Savannah is restricted to 4KB.

When a process needs to send a request to a server, it writes the message containing the request in its own outbox (1), then notifies the server (2) about the pending message. When the server is available to handle the request, it reads the message from the outbox of the requesting process (3) and starts handling it. Once response is ready, the server writes a message back containing the reply into the inbox of the requesting process (4) and notifies the application (5). At that point the application can read the reply (6).

Since in our design, processes run inside a Virtual Machine and have direct access to some low-level features of the underlying CPU, we have the opportunity to provided various notification methods.

The first (and most popular one also in other operating systems) is polling. To use it, a bitmap is stored in another part of memory where the application can set the bit corresponding to its own cpuid, while the server always polls through all the bits to check which ones are set or not. For the reply, a bit in the inbox is used, so that the application always polls through that bit and when the reply is ready, the server sets it.
To use it, a bitmap is allocated where each application can set a bit belonging to it, while the operating system server is polling over this memory area checking all the bits.

The second method is using the CPU instructions \textit{MONITOR/MWAIT}. Because in our design applications have direct access to some privileged low level instructions like have access to privilege instructions like \textit{MONITOR} and \textit{MWAIT}. The use of \textit{MONITOR} and \textit{MWAIT} for notification is very similar to polling, but instead of polling through all the bits, a bitmap area can be monitored and the \textit{MWAIT} instruction is woken up when the memory area monitored is changed.

The third, and last method of notification is using Inter Process Interrupts (IPI)s. In our design, using the VT-x technology, interrupts can be delivered directly to the VM that hosts the application. This also can be used as a notification method. This method eliminates the need for a bitmap, because IPIs can be sent directly from applications to servers and vice-verse.

To the best of our knowledge, Savannah is the only operating system that can supports all these message passing methods. We are going to evaluate each of the them in the Evaluation Section 5.1.

The aforementioned methods for message passing can only be used for communication between applications and servers, because servers are the only VMs in Savannah that have access to the while memory, and therefore can read and write into the outbox and inbox of each process. So these methods can not be for communication between two normal processes. To solve this issue, we developed another way of communication called channels, explained in Section 4.4.2 for communication between normal processes, and more specifically, for communication between drivers, and user applications.

\subsection*{4.4.1.2 How to use the API}

To send and receive messages, two functions are implemented in the library that every application can use:

\begin{verbatim}
void msg_send (const int to, const int number, const void *data, const int size);
int msg_receive (int from);
\end{verbatim}

The first one is used to send a message from a normal process to a server, and the parameters passed are the following.

\begin{itemize}
  \item \textit{to}: Cpid of the destination server
\end{itemize}
• number: Number of the service call
• data: Data of the message, usually the parameters of the service call
• size: Size of the data

The second one is used to receive the reply message. It can be called immediately after sending the message or any time later if the reply is not needed immediately. The parameters passed are the following.

• from: Cpuid from who you are expecting a reply, typically the same as the “to” of the msg_send()

On the server-side, two different functions are used:

```c
struct message * msg_check();
void msg_reply ( const int from, const int to, const int number,
                const void *data, const int size);
```

The first function is used to constantly check messages. It can be used both for polling and MONITOR/MWAIT. When a new message arrives, the function returns the message structure. In IPI based IPC, there is no need for this function, but a handler of the IPI needs to be written and installed.

The second function is only used to reply to a message. The parameters are the same as msg_send(), but the implementation is different. Namely, while the msg_send() writes into its own outbox, msg_reply() writes into the inbox of the process specified in the parameter.

### 4.4.2 Channel Communication

As explained earlier in Section 4.4.1, the Message Passing method can only be used for communication between applications and servers. For communication between two normal processes (like an application and a driver) another method needs to be used. For this purpose we designed a method called Channels Communication. In current version of Savannah, Channels are only used for communication between applications and drivers.
4.4.2.1 Architecture

As Figure 4.6 shows, the creation of a channel is not straightforward. When an application running inside a VM opens a special file associated to a device (1), the FS asks the PM (2) to allocate a page in memory and to map it both for the requesting application and the driver corresponding to the device. Once the PM has mapped the memory, it returns the address to FS (3) which the notifies the application and the driver (4). From that point on the channel (the page in memory) is created and it can be used to communicate directly between the application and the driver.

The channel stays in place until the application closes the file descriptor. When an application closes a file, it sends the closure request via a message to the FS. When the request is received by the FS, it sends a message to the PM to free, and unmap the memory allocated for the channel. As soon as the channel removal is confirmed by the PM, the FS notifies the driver that the channel no longer exists, and thus it should not be used anymore.

Since the channel is a shared memory page between the application and the driver, there needs to be a mutual agreement between the two sides on the communication protocol. In our implementation, we attach a small header to the data written into the channel to ease the communication. However, we do not define any standard for this, and different communication protocols can be put into usage as long as both sides have a mutual agreement on the protocol.
4.4.2.2 How to use the API

We use POSIX interfaces to hide the extra complexities added by IPCs. This is also essential to ease programming and porting application from other platforms to Savannah. For instance, we implemented POSIX functions `read()` and `write()`, for receiving and sending data to devices that are currently supported by Savannah.

However, usually in newly design operating systems, POSIX compatible functions are not the most efficient way of handling requests. This is because POSIC usually dictates some extra data copies that can be eliminated in new designs. Therefore, if in an application, performance concerns are greater than portability, developers can chose their own (and probably non-standard) way of requesting for services or communicating with servers or device drivers.

Our user library provides two functions for channel communications:

```c
void cnl_send(struct channel *cnl);
void cnl_receive(virt_addr_t channel);
```

As it can immediately be noticed, the user does not need to pass any data or size to the `cnl_send()` function. This is because this function only notifies the driver, and therefore must only be called when the data is in the channel and ready to be sent to the driver.

The `cnl_receive()` function requires to have the virtual address of the channel as the only argument. This function returns as soon the data is ready, and then the process can read the reply.

In the driver side, two other functions must be used:

```c
struct channel *cnl_receive_any();
void cnl_reply(struct channel *cnl, const int count);
```

The first function has no argument, because the driver can not create or destroy any channel directly. Instead, the FS creates and destroys channels, and the driver just uses the created channels to communicate with user applications.

The `cnl_receive_any()` function loops through all the open channels searching for user requests. Once a request is received, it performs the needed actions and then creates and copies the response into the channel. Next, it calls the `cnl_reply()` function to notify the user application about the pending response.
4.5 Process Manager

The process manager is the first VM launched in Savannah. After it boots up, it sends a message to all VMMs, allowing them to launch their pending VMs, if they have any. The process manager has a wide range of responsibilities. It handles all process related services. In addition, because memory manager is also integrated into the process manager in Savannah, all memory related services are also given by the process manager.

4.5.1 PM vs. MM

Usually the process manager and the memory manager are two separate entities in operating systems, but in Savannah we decided to integrate the memory manager into the process manager. This design decision has its cons, and poses that we are going to discuss in this section.

Having the memory manager as a separate entity boosts the modularity of the system. It reduces the code size of the PM. It will eliminate the need to give the whole PM full access to the memory, which will increase the reliability of the system. On the other hand however, having the memory manager as a separate process on the system will increase the number of messages transmitted dramatically. This is primarily because there are very few services which are solely memory related. For instance in POSIX, \texttt{brk()}, \texttt{sbrk()}, and \texttt{mmap()} are the only memory related services defined. Of course this does not mean that rarely memory operations are taking place in the system, but it does mean that the main customer of memory manager services are not the end user applications, but instead other system components like the process manager and the file server. Any process creation and destruction require multiple memory operations. Disk \texttt{read()/write()s} need memory contents to be copied from the memory address space of the file server into the address space of the end user application. Having the memory manager as a separate entity in the system means that for all such services, the process manager, or the file server, needs to send an extra message to the memory manager so that the memory manager performs the requested operation. These extra messages of course, can cause sever performance penalties.

All this means that a trade off must be made between the modularity, reliability, and the performance of the system. We believe that isolating all system components into virtual machines, and running them as independent processes provides enough reliability and modularity in our design. Therefore, we decided to integrate the memory manager into the process manager to reduce the number of exchanged messages needed to handle process related services in Savannah. This decision is made considering that most of
memory demands of the process manager, need changes in the current memory layout. For example, a `fork()` needs pages to be allocated, data to be copied, page tables to be created and filled, and etc. All these change the layout of the allocated, and free pages, whereas for most of file server requests, this is not the case. For instance, a `read()` from a file, does not need any extra memory to be allocated, or to be freed. All what it needs is to copy data from one buffer to another.

Based on all these considerations, we decided to integrate the memory manager into the process manager which allows the process manager to (de)allocate memory pages directly and without the overhead of sending and receiving extra messages. We do not integrate the file server, and process server, but we do map the whole available memory for the file server. This will reduce the need for message exchanges for every `read()`/`write()` operation, and boosts the response performance to such services. This means that the file server can read/write from/to any memory region, but it will not manipulate the memory layout at any time.

### 4.5.2 Memory Manager

Usually the memory manager in operating system has the following responsibilities: 1. Providing the address space isolation among the residing processes in the memory, 2. Keeping track of allocated and free blocks of memory, 3. Allocate and free memory blocks.

To gain these goals, traditional operating systems usually divide the memory into small blocks known as pages, and (de)allocate these pages for user processes and operating system components. Traditionally, operating systems tend to impose rigorous isolation policies through paging mechanism. This works fine for a large range of applications, however, there are some applications that need more control over their low level paging data structures. Existing commercial operating systems offer some functionalists to respond to these demands, but unfortunately to preserve their rigorous policies, these functionalists are far from ideal solution for demanding applications. Therefore, such applications would significantly benefit from having access to those low level resources. For instance, garbage collectors would have significant speedup if they had access to the paging system [28]. Or as another example, live migration applications would significantly benefit from being able to directly change page protections and handling page faults [29].

Like other operating systems, Savannah also divides the memory into small 4KB chunks, and keeps track of allocated and free memory blocks, nonetheless unlike traditional operating systems, Savannah has a more relaxed memory management policy, meaning that it gives applications some control over their page tables. These protections can
only be made stricter by the application, and not relaxed. Initially, when an application is created, PM applies some default protections on pages given to the process, but applications are give the permission to change these default protections any time they need.

4.5.2.1 Tow Step Page Translation

Since in our design processes are running inside a virtual machine, PM creates two sets of page tables during a process creation: First, local page tables that are used for guest virtual to guest physical address translation. User application has control over these page tables and can be change them based on its needs at runtime. Second, Extended Page Tables (EPT)s which are used for guest-physical to host-physical address translation and applications do not have access to them.

The memory manager applies some default protections for each page at the EPT level and imposes no protection on the application’s local page tables. This design gives processes the chance to easily change the default settings and impose stronger restrictions if needed. As Figure 4.7(a) shows, if any of threads inside the virtual machine violates the protection imposed by the memory manager, this will lead to a VMEXIT operation which moves the control to the VMM. But this relax memory management policy gives the processes...
to have some local control over their memory management, and by manipulating their local page tables, apply their own policies, and thus if any of the threads violates these local policies the application will catch and handle the exception locally and without any intervention by the operating system (Figure 4.7 (b) shows this second case).

As a practical example of this flexibility, an application may want to statically allocate memory on the heap or the bss segment for its user-level threads' stacks, but at the same time, it may want to have non-executable stack for each thread. In such a case, the application can easily change the attribute of allocated pages to readable and writable, but non-executable.

### 4.5.2.2 Cache Policy

Caching has a great impact on the performance of the running process. Fetching a word from memory can be up to five orders of magnitude slower than fetching the same word from the L1 cache in an X86 processors [30]. Savannah dedicates the cores to processes, and this would cause significant performance gain because the entries in L1 cache are not evicted by other processes sharing the same CPU. Moreover, the entries in cache L2, and L3 are also evicted less frequently because less processes compete for them.

For performance reasons, we try to map as many parts of the system as we can with write-back cache policy, however, some parts of the memory require different cache policies, and even some parts of memory must not be cached at all. For example, the MMIO area must be mapped as uncacheable. Apart from these special memory areas, the memory manager sets the write-back cache policy for all other pages by default. Nevertheless, if a process needs a different cache policy for a page, it can easily change it by changing the corresponding attributes in the page tables.

### 4.5.2.3 Memory Chunk (De)Allocation

As discussed in Section 4.2.3, one of the parameters passed to the process manager is the size and the first free address of the main memory. Figure 4.8 shows the layout of the main memory at this point.

The allocated area in the beginning of memory depicted by Figure 4.8 includes the memory allocated for boot process, servers, and also the INIT process. The memory manager ignores this part of the memory, which means this part of the main memory would never be reused. This is not a serious problem, because some services such as PM, FS, and INIT are so essential that freeing the memory allocated to them is not an option under any circumstances, but as an optimization we might be able to consider the
memory allocated to boot process as free, and add it to the available free memory that MM is managing. We don’t do this however, because this part of memory is relatively small, and the amount of memory saved is so small that might not be worth of the hassle.

The memory manager provides two basic operations which are only available for the process manager: allocate_mem_pages(), and free_mem_pages(). The performance of these two operations can play a very important role in the overall performance of some services like fork(), and exec(), and also channel creation and deletion. Therefore, for the good performance of some process manager services, the overhead of these operations should be minimal.

The memory manager splits the memory into 4KB chunks, and allocates and frees the memory in blocks including one or more chunks and keeps track of all allocated and free memory blocks. To this end, it maintains a linked list of nodes that each of them describes a memory block. The information stored in each node includes a reference counter which indicates to how many processes this memory block is assigned, the start physical address of the memory block. The fact that a memory block can be assigned to more than a process may seem strange, but as we will describe later, in many places, we share pages between more than one process for various reasons. For example, to speed up data flow between processes, or sometimes to optimize operating system services. For instance, in a fork operation, by definition the operating system has to make an identical copy of the requesting process. This operation can be very expensive, since it requires
copying multiple memory pages. As an optimization, we only copy writable pages, and share read-only pages among the parent and the child. A memory block with reference counter zero is considered to be free. The Figure 4.10 shows how this memory allocation works. This algorithm has the complexity of $O(n)$ for the memory allocation.

Initially this linked list has only one node which includes all free memory chunks. Once some pages are allocated, a new node is added to this linked list which includes these allocated pages. To allocate pages, the memory manager should first add a new node to the linked list containing memory blocks. This memory area for these nodes is preallocated when the memory manager begins working. As Figure 4.9 shows, It first reserves a part of the memory as a node vector which is a memory area reserved for nodes describing memory blocks. The memory needed for blocks depends on how fragmented the memory can be. Therefore, the memory manager allocates enough memory for nodes considering the worst case scenario which is when memory is most fragmented. This only happens if each block of memory contains only one page, and these pages can not be fusioned.

The allocation algorithm that the memory manager uses to assign a node to a memory block is to use the physical address of the first memory chunk in the block as the hash value pointing to a specific entry in the node array. This algorithm will guarantee that
once a block is allocated, any future access to that block will take place with the time complexity of $O(1)$.

This allocation mechanism means that after sometime, the memory would be fragmented. To avoid memory fragmentation, we apply a fusion phase after each \textit{free} operation. The fusion takes place if after a free operation, there are two adjacent free memory block nodes in the linked list. In such cases, the memory manager will merge these two blocks into one. Fusion operation is very cheap, because it is not applied on the whole linked list, but rather only on the memory block which is freed and its previous, and next blocks. This guarantees the memory to be totally defragmented, because it is applied after all free operations, so if after freeing a memory block, there are two adjacent free memory blocks, they will be immediately merged into one, and therefore, memory would not be fragmented. Figure 4.11 depicts how this process takes place.
4.5.3 Process services

At the moment, the process manager of Savannah supports fork(), exec(), exit(), waitpid(). The definition of these services are pretty much the same as theirs in other UNIX based systems. The only difference in Savannah is that once processes are created, they run inside a virtual machine on a separated core. Although the behavior of these services are similar to other UNIX based systems, but their implementations are very radically different. In this section, we are going to give detail description of how any of the aforementioned services are implemented in Savannah.

4.5.3.1 fork()

According to POSIX standard, fork() creates a new and identical instance of the running process. Apart from the fact that almost all widely used operating systems are ignoring and deviating from this definition because of performance concerns, the implementation of fork() is relatively easy and straightforward. All what the operating system needs to do is to copy the process image of the parent, to the process image of the child. The process image usually is comprised of all memory pages of the process, plus the values of registers used by that process. Therefore, to handle a fork() request, the operating system has to allocate as much memory as the parent process has already allocated, and then copy all the pages of the parent in the newly allocated pages of the child, and to set up new page tables to allow the child process to access its pages. It also has to the content of the processor registers exactly at the moment it executed the fork() request to the child process image so that when the child begins its execution, it starts right after the point where fork was called. The only exception about the register values in X86 architecture is the value of rax register. In X86 architecture, this register is consider to contain the return value of functions. Since the fork() system call return two different value to child, and the parent, the value that the operating system sets in the process image must be different as well. Thus, the value that the operating system assigns to the rax register in the parent process image is the child’s process ID, and in the child’s process image is zero. These values are the return values mandated by the POSIX specification.

In a traditional design of an operating system, since the fork() service is delivered through the system call interface, the user process traps to the kernel and when it does so, all processor registers are saved on its stack. Hence, its very easy for the operating system to access those values, and copy them to the child’s process image.

In Savannah however, the process does not trap to the kernel (basically because there is no kernel), and because the process itself sends the message to the PM to ask for the
Chapter 4. Implementation

Figure 4.12: Pseudo code of forks’ wrapper function

```assembly
.align 0x10
.globl fork
fork:
    subq $REG_VECTOR_SIZE, %rsp /* Create The Vector */
    /* Save RIP */
    movq REG_VECTOR_SIZE(%rsp), %rax
    movq %rax, CPU_REGS_RIP(%rsp)
    /*
    * Save the value of RSP before fork.
    * Add 8 because of fork return address
    */
    movq %rsp, %rax
    addq $( REG_VECTOR_SIZE + 8), %rax
    movq %rax, CPU_REGS_RSP(%rsp)
    /* ================================== */
    /* ... Save all registers here ... */
    /* ================================== */
    movq %rsp, %rdi /* pass in the vector to fork_internal */
    callq fork_internal
    addq $REG_VECTOR_SIZE, %rsp /* remove the vector */
    /* ================================== */
    retq
.size fork, .-fork
```

Figure 4.13: Routine fork_internal sends the beginning address of register vector to the PM

```c
int fork_internal (virt_addr_t register_array_vaddr)
{
    int result;
    struct fork_ipc fork_args;

    fork_args.cpuinfo_vaddr = (virt_addr_t)&cpuinfo;
    fork_args.register_array_paddr = virt2phys (cpuinfo, register_array_vaddr);

    msg_send (PM, FORK_IPC, &fork_args, sizeof (struct fork_ipc));
    msg_receive (PM);

    memcpy (&result, &cpuinfo->msg_input[PM].data, sizeof (int));
    return result;
}
```

fork service, the user application keeps using the processor registers even after the fork is called. So, in order to save the state of the parent process right after the fork request, what the parent process does is to take a snapshot of the values of all processor registers before sending the request to the PM. To this end, the `fork()` routine is wrapped up into a short assembly code which stores the content of all necessary registers on the parents’ stack. Figure 4.12 shows the pseudo code of this assembly function. Subsequently, as the pseudo code in Figure 4.13 shows, the `fork_internal()` routine sends the beginning address of this vector of registers in its request message to the PM. Figure 4.14 depicts the stack layout of the parent process during the fork process.

When the PM receives the fork request, it first checks to see if there is any free VMM that can host the child or not. If there is no free VMM, the PM immediately send a negative
response to the parent process. In case there exist a free VMM, then theoretically the PM has to allocate as much memory as the parent process possesses, however, as an optimization we only allocate memory for the writable pages. Thus, the parent and child share executable, and read-only pages with each other. This optimization significantly reduces the number of memory movements needed to handle the `fork` request. Following the allocation, the PM needs to copy the content of all writable pages from the parent address space into the childs’ newly allocated pages.

The next essential step is to copy the content of parents’ processor registers into the child process image. As the following code snippet shows, these values are copied into the `vm_info` field of the `cpu_info` structure:

```c
memcpy ((void *)&child_cpu_info->vm_info.vm_regs, (void *)fork_args->register_array_paddr, sizeof (struct regs));
```

If the parent has any open file, all the file descriptors need to be copied in the childs’ process image. In addition, the PM also must create a copy of every channel that the parent process has opened. The PM also copies all the permission and security related fields in the `cpu_info` structure of the parent to the child process to make sure they have the same privilege and permission levels.
All the aforementioned steps are taken to guarantee that both parent and child processes are in exactly the same state when they begin their execution right after the point \textit{fork()} was called.

At the end, the PM sends a message with the process ID of the child to the parent process, and then sends a message to the VMM chosen to launch the child process asking it to launch the new process.

\subsection*{4.5.3.2 exec()}

The \texttt{exec()} family of system calls in POSIX are used to replace the requesting process with another program. In our implementation of \texttt{exec} service, like all other process-related services, the requesting process first sends an explicit message to the PM, and asks for the service. The major difference between the \texttt{exec} and other services is that immediately after sending the message, the requesting process performs a \textit{VMEXIT} operation and exits the VM mode. This step is very crucial, because as we launch the new process on the same CPU as the requesting process was running, if the \texttt{exec} succeeds, the VMM should be awake and ready to launch the new VM. The message sent by the process to the PM in our design conveys the following information:

- The path to the executable file in the file system
- An array of arguments to be passed in to the new process
- A pointer to a vector containing the values of all CPU registers

The PM needs the path of the executable in order to find, parse, and load the ELF file of the program that must replace the requesting process, and the array of arguments is required to be passed to the new process as the \textit{argv} parameter. The path to the executable file, and the list of arguments are de facto parameters passed to almost any of the \texttt{exec} family routines. Additionally, in our design, similar to what explained in the Section 4.5.3.1, the requesting process sends a pointer to a vector on its stack that containing the contents of all CPU registers right at the point before sending the message to the PM. The major difference is that the content of \textit{RIP} register in this vector is engineered such that it points to the end of \texttt{exec} routine. That is because this vector is only used in cases when the \texttt{exec} fails, and to relaunch the requesting process from the point it asked for the \texttt{exec} service. The routine below shows how the \texttt{exec} routine creates the vector and sends the request to the PM. Please note that the interface of our \texttt{exec} implementation slightly deviates from the POSIX standard.
int exec (char *path, char **argv)
{
    struct exec_ipc exec_args;
    size_t len;
    volatile static struct regs registers;
    /* Checks to avoid buffer overflow */
    len = strlen(path) > MAX_PATH ? MAX_PATH : strlen(path);
    strncpy(exec_args.path, path, len);
    exec_args.path[len] = '\0';
    /* Convert virtual addresses to physical. PM needs physical address */
    exec_args.argv = argv != NULL ? virt2phys (cpuinfo, (virt_addr_t)argv)
                            : NULL;
    exec_args.registers = virt2phys (cpuinfo, (virt_addr_t *)&registers);
    /* First, save the content of %rdi into RDI_IDX in cpuinfo */
    /* We are going to use them to save the other registers */
    __asm__ __volatile__ ("movq %%rdi, %c[VM_REGS_RDI_IDX](%0)\n\t"
                        "movq %%rax, %c[VM_REGS_RAX_IDX](%0)\n\t"
                        : "S"(& registers),
                        [VM_REGS_RDI_IDX]"i"(offsetof (struct regs, rdi)),
                        [VM_REGS_RAX_IDX]"i"(offsetof (struct regs, rax)));
    /* Send the message to the PM now */
    msg_send (PM, EXEC_IPC, &exec_args, sizeof (struct exec_ipc));
    /* Exit the VM mode, wake up the VMM */
    __asm__ __volatile__ ("vmcall\\n\t" /* VMEXIT */
                          "exec_resume:\\n\t" /* If exec fails, we resume at this point ... */
                        );
    /* We already failed if we get to this point */
    return -1;
}``

There are few important things in this source code that should be pointed out here. First one is that before sending the arguments of the new process, and the start address of the vector to the PM, the virtual addresses are translated into the physical addresses.
That is because these variables are within the address space of the running process, and hence all values stored in these variables are virtual addresses that are only meaningful within this address space. Therefore, a virtual to physical translation should take place before the PM can really use these values. However, the PM also has enough knowledge to perform the translation, but as an optimization we aim to offload the servers as much as possible. Thus, in our design, the user application will do as much as the job as they can before send the request to the operating system servers.

The second point worth of pointing out is how the RIP value is assigned in the register vector such that if the exec fails, the requesting process resumes its execution as if the exec returned an error message. The value assigned to the RIP register is the address of the label exec_resump in the last inline assembly code snippet. This label is located right after the vmcall CPU instruction. As soon as the vmcall instruction is executed, the processor switches from the VM to the VMM mode, therefore, CPU will not execute the instructions after the vmcall instruction. However, when the VMM takes the control of the CPU, it waits for a message from the PM to launch a VM. If the exec fails, the PM updates the register contents of the vm_info field in the cpuinfo data structure of the CPU belonging to the requesting process with the values in the register vector, and sends a message to the corresponding VMM and orders it to relaunch the requesting process with the new values in the vm_info field. With RIP pointing now to the exec_resume virtual address, the requesting process will resume its execution from that point onward which will leads to the exec returning -1 indicating that the exec failed.

There are few conditions under which the exec fails. The first, and probably the most likely one is if the path to the executable is not valid, which means the executable file is not found in the file system. Its important to note that even though the path is sent to the PM, but the PM requires assistance from the FS to find the executable file. As a result, upon receiving the request, it sends a message with the received path to the FS and asks to open the file. Subsequently, if the FS can not find the requested file, it will send back a message with an error code, or otherwise with a valid file descriptor.

The other situation that may cause exec to fail is memory shortage. If there is not enough free memory left to allocate the pages required to create the new process, the exec will fail.

Nevertheless, this condition can be examined in two different ways. Given that abstractly speaking, the exec entails two major steps: deletion of an existing process, and creation of a new one; the order in which these two steps take place changes the amount of free memory available. Apparently if the deletion takes place first, there would be more available free memory for the creation of new process. Nonetheless, this does not necessarily guarantee that there would be enough available free memory to create the
new process. Considering that if the new process can not be created for any reason, the existing process should resume its execution, deletion before creation can be very dangerous, unless, before deleting the existing process, and creating the new process, a calculation being carried out to guarantee after deleting the requesting process, there is enough free memory to create the new process. That is the approach taken to implement the `exec` is Savannah.

### 4.5.3.3 exit()

The `exit()` service call is relatively simple. Together with `exec()`, they are the only two situations where a VM does a VMEXIT operation in Savannah. As the code snippet in Figure 4.15 shows, in the case of exit, the process sends a message to the PM. This message contains the request for the exit service plus the exiting value of the process. After sending the message to the process manager, the requesting process performs the VMEXIT operation to wake up the VMM. As it was already discussed in Section 4.3 and shown in the code snippet of Figure 4.3 as soon as the VMM is woken up, it again waits for a message from the process manager.

Upon receiving the exit request from the VM, the PM labels the VMM which was running the requesting process as a free VMM, and then deallocates all the memory pages belonging to the requesting process, and if the parent of the requesting process is waiting for its child's termination, the PM will send a message to inform the parent about the termination of the child.

### 4.5.3.4 waitpid()

The implementation of this service in Savannah is very similar to other existing operating systems. Like any other operating system, `waitpid()` allows a parent process to wait for one of its children to terminate. In a time sharing operating system, this service is usually
4.6 File System

Savannah does not support a full-featured file system like EXT, or FAT. Thus, to fulfill our basic needs we created a very simple file system structure that lives in the RAM. This decision entails that our file system must be small so that it can fit into the main memory. Moreover, the fact that the file system is located in the main memory with other user applications and operating system services implies that some special measures must be taken to guarantee the privacy and security.

In this section we explain the details of the file system implementation and how it can be created.

4.6.1 Implementation

For our research purposes, we do not need advanced features of a real file system such as support for directories, and huge files. We just need a very basic file system that can store, and retrieve the binary files of our applications, plus some basic and small configuration files. Therefore, the overall structure of our file system is very simple.

Figure 4.16 shows how different parts of the file system are structured in memory. The first 4 bytes of the file system will store an integer representing the total number of files.
struct header {
  char name[32]; /* File name */
  uint32_t type; /* Type of the file (normal file, char, block ...) */
  uint32_t length; /* File length */
  uint64_t dst; /* Cpuid of the destination (for CHAR or BLOCK fds) */
  uint64_t offset; /* Offset where the file is located starting from 
                  * the beginning of the file */
};

Figure 4.17: Structure of the header

available in the file system. After that all the headers of the files will be stored. The number of all the files available in the file system is useful because otherwise there is no way to know how many headers are stored. Figure 4.17 shows the structure of each header and contains all the informations that the File Server of Savannah needs to know to work properly. After the list of headers, the content of each file is stored, in the same order that headers were stored.

Now we are going now to explain each field of the header structure in more details.

- **name:** File name. The maximum length has been fixed to 32 bytes, but it can be easily increased.
- **type:** The type of the file. If the file is a normal file, it can be read from the File Server, otherwise if the file is a special file that is associated with a device (like console, keyboard or network)
- **length:** The length of the file, so that the File Server can check if an application wants to read more than the length of the file.
- **dst:** If the file is a special file connected to a device, this field is set with the cpuid of the driver that needs to be contacted.
- **offset:** Since every file is stored in memory one after the other, this offset is the number of bytes where the file data begins. This field is counted from the beginning of the file system. This is the only way to understand where the file starts in memory.

This file system is very simple but it has all the features that Savannah needs.

### 4.6.2 Creation

As already explained in Section 4.2.1, the image of the file system is passed as an module to GRUB to be loaded during the booting process. To ease the creation of this image we
have created a simple program available at “initrd/create_initrd.c”. The usage of this program is the following:

```
$ ./create_initrd file1 file2
```

The program automatically creates a file called “initrd.img” containing the file system image with the structure described in previous section. This file is then loaded into the main memory by GRUB, and the start address of that is passed as an argument to the File Server when the process is created by boot stage2 program.

### 4.7 File Server

The File Server as the only authority that has access to the file system handles all file-related requests. As mentioned in Section 4.2.3, FS is created by the boot stage2 program, and always runs on core number 2.

#### 4.7.1 File Server services

In this section we are going to explain which service calls can be used to communicate with the FS. To increase the portability, the programming interface of all FS services (implemented in user Library 4.8 to hide the complexity of IPCs between operating system components) are fully POSIX compatible.

At the moment, the File Server supports all the basic file-related service calls: `open()`, `read()`, `write()`, `close()` and a special service call `load()` that is used just from the process manager.

It’s also important to note that like the PM, FS also has full access to the main memory. This decision is made to reduces the number of messages exchanged to handle a request. Because otherwise, for example for a `read` service, the FS has to send a message to the PM and ask the data to be copied from the file system into the user process buffer, whereas with full access to the memory, the FS itself can copy the data.

In this section, we are going to give detailed description on how each service call is implemented in Savannah.
struct file_descriptor {
    uint32_t type; /* Type of the file (normal file, char, block ...) */
    uint32_t length; /* File length */
    uint64_t dst; /* Cpuid of the destination (for CHAR or BLOCK fds) */
    uint64_t offset; /* Offset where the file is located starting from
        * the beginning of the file */
};

Figure 4.18: Structure of the file descriptor

4.7.1.1 open()

This service call is used to open any file present in the file system. It can be called from any process except the drivers because for security reasons the drivers are not allowed to open files.

If the calling process is not a driver, the open() service call searches for a free file descriptor. By default the maximum number of opened files is 32, but in practice the maximum number of opened files is 30, because the FD 0 is reserved for stdin and the FD 1 is reserved for stdout.

In Savannah all the informations about the FDs are stored in the cpuinfo structure. So, to search for free FDs the File Server has to look into the cpuinfo structure of the process. It is important to note that servers have full access to memory, while processes do not, so the File Server can change the cpuinfo structure and set the file descriptor as busy, while the process can just read it.

If a file descriptor is available, the next check is to check whether the requested file exists on the file system or not. If one file descriptor is available and the file exists, the function can proceed to opening the file. The procedure is simple; the FS simply copies the file information from its header into the file descriptor assigned to the process. The only information which exist in the header, but is not copied in the file descriptor is the file name. This is because file name is used only to find the file on the file system, and open it. But once the file is found, and its start address in the memory is acquired, there is no real usage for the file name.

There are also some exceptional cases for opening special files representing char or block devices. In such cases, as shown in Section 4.4.2 and in Figure 4.6, the file server contacts the process manager asking to allocate a memory page and map it both for the process and the device driver. In this case, the type field in the file descriptor structure 4.18 is set to either “block” or “char” depending on what driver it is, the dst field is set to the cpuid of the device driver, and the offset field is set with the start address of the memory page mapped for the channel.
Chapter 4. Implementation

After this process is completed, the file server sends a reply message back to the process with the file descriptor number that is used. If any error occurs, -1 is returned. If the file opened is a special file, the file server also adds the newly opened channel to the list of available channels in `cpuinfo` structure of the device driver, so that the `channel_receive_any()` automatically starts checking also the newly created channel.

### 4.7.1.2 read()

This service call is used to read from a file on the file system. Since it uses the POSIX interface, the prototype of this function is the following:

```c
int read(int fd, void *buf, int count);
```

The file server does few sanity checks, such as checking if the file descriptor is valid and opened, if the buffer is a valid pointer and if the count is not greater than the maximum size of the file.

After these sanity checks, it simply calls a `memcpy()` function to copy `count` bytes from the file system to the buffer and returns the number of bytes copied.

User applications use file-related functions in user library to communicate with the FS. These functions are mostly intended to hide the tedious details of IPCs required to handle the requests. The `open()` function for example, sends the message to the file server and waits for the reply. For the `read()` however, the implementation is a bit different, because the `read()` can be called to read either from a normal file or a special file. In case of a normal files, a message is sent to the file server and then the library function waits for the return value. But in the case of reading from a special file, instead of sending a message to the file server, the function reads from the channel. Thus, all read operations from devices bypass the FS, and hence offload the server. As a result, the FS can respond to the requests that are really about reading or writing into file more efficiently.

### 4.7.1.3 write()

This service call implementation in library is similar to the `read()` service call. The difference here is that the file system structure, explained in Section 4.6, does not support any writing into the file system, so when a `write()` function is called, if the file descriptor is a regular file, the function immediately returns -1. Otherwise, if the file descriptor is associated to a special char/block device, it copies the buffer data into the channel.
4.7.1.4 close()

This service call is used to close an already opened file descriptor. Upon receiving the closure request, the file server checks to make sure the requested file is really open, and then if true the file server resets all the fields in the file descriptor structure 4.18 (type, length, dst and offset) to zero, so that it can be reused if another file is opened.

4.7.1.5 load()

This is a special service call that is not coming from any standard. It can be called just from the Process Manager and is used when a process sends a message to the Process Manager asking to exec() a new process. One of the parameters of the exec() is the path to the executable file. The process manager needs to know where the ELF binary file is stored in the file system in the memory For this reason the process manager calls this special service call with the following interface

\[
\text{phys_addr_t load(char* path);}
\]

So it takes the path of the ELF binary file. Then the FS returns to the PM the start physical address of the file in the memory if the file was found and NULL otherwise.

4.8 User Library

In most operating systems, kernel usually hides hardware complexities and provide a high level of abstraction for application developers. This eases the development and increases the portability of software. Savannah has no kernel; therefore, to hide hardware complexity, it provides a rich and extensive library that provides the same level of portability and ease of software development. Similar to what exokernel was doing [20].

This library can provide as much portability and compatibility as desired. Our library provides some POSIX compatible routines as well as some which are Savannah specific. POSIX compatibility is essential because it allows available UNIX applications to be ported to Savannah. In addition, our special design gives us the opportunity to implement some nonstandard functions that if used in application development, that can lead to significant performance improvements. For example, in the POSIX standard, the read() operation is blocking, but our library gives developers the opportunity to use aread(), which is a non-blocking version of the POSIX read().
4.8.1 Driver in User Library

Giving user application direct access to some hardware devices entails that they need to program the device before they can use it. This can immensely increase the programming complexity. To ease the programming, we provide configurable drivers as parts of the library. For instance, a driver for LAPIC timer is available in the library that facilitates working with local timer. This timer driver is also used to implemented some other parts of the library, like our special threading system that explained in Section 4.8.3 that heavily depends on this library driver.

4.8.2 Network Stack

Typically, the network stack is part of the operating system and, in monolithic designs such as Linux, Windows, BSD, and others, it is part of the kernel. Integration of the network stack into the operating system can increase the network latency and also would decrease the predictability of networking behavior. Therefore, the ideal case would be when each user process is attached to a network interface and a network stack, however, without proper hardware support this is not possible.

With virtualization penetrating into the network interfaces [7], now its possible to give each application its own network stack. To this end, we ported LwIP [31], a fully functional light weight user level network stack to Savannah, as a part of the user level library. Thus, when an application needs to receive data from network, it calls our implementation of \texttt{read()} in the user library which sends a request to the network interface driver through the channel. When a packet belonging to this user application shows up on the wire, the driver writes the raw packet to the channel. Upon receiving the packet from the channel, \texttt{read()} passes the packet to LwIP to deal with the packet, and deliver the payload to the user application in case there is any data in the packet. Figure 4.19 shows this design.

What mentioned so far about the network stack is only feasible if the network interface supports virtualization, or in some special occasions when an application is pinned to a particular network interface. Nevertheless, virtualization in network interfaces is not yet a very usual, and we have put this into consideration in our design. Therefore, if the network interface does not support virtualization, we propose two alternative designs. Either run the LwIP as a separate server, or link it with the network interface driver. Each of these designs has it own pros and cons. Running the LwIP as a separate server increases the overhead of IPC, and thus the network latency, whereas linking the LwIP with the network interface driver requires less IPC, but increases the workload in driver side which decreases the responsive of the interface. Figure 4.20 shows these designs.
Regardless of the implementation method, as long as programmers are using our user level library, this whole process of sending and receiving network packets is transparent to the user applications, and the only interface that needs to be used are the standards `read()` and `write()` functions like in other operating systems.

4.8.3 Special Thread Library

Threads are either managed by the operating system, or within the user process. Each of these methods have its own advantages and disadvantages. For instance, managing threads in user level has the advantage that switching threads can be cheaper, and the
process can have more control over its threads. However, the downside is that if any of the threads performs a blocking call, the whole process would be blocked. Moreover, scheduling threads in user level requires a quick and cheap access to a timing resource, and since user applications do not have direct access to timer hardware, their scheduling system is not very efficient. On the other hand, some operating systems such as Linux, treat threads as normal processes and schedule them with other processes within the kernel. This way of looking at threads has the advantage that if any of the threads within a user process performs a blocking call, the whole process won’t be blocked, but the downside is that from the operating system point of view, the thread is just an opaque entity and the operating system does not know anything about what is inside the thread. Therefore, all decisions that the operating system makes is based on the observations it gathers from the behavior of the running threads, and based on these observations, it will make its further decisions.

Our special design provides a unique opportunity to take the best from both worlds. Being able to program the local APIC to receive timer interrupts in customized intervals empowers us to implement a lightweight threading library to give each application the opportunity to have multiple threads running simultaneously. This special threading system, allows each application to perform customized scheduling algorithms to let threads run with maximum performance; Applications can collect statistics of the current execution performance, and change the scheduling algorithm on the fly to gain better performance and efficiency. This unprecedented flexibility is possible thanks to being able to program low level devices such as timer, directly by the applications.

Another feature that the architecture of our system provides is ring protection among different threads. Each thread (by default) runs in ring 0 inside the VM. However, using a function provided in the library, each thread can switch between protection rings. All such features that we provide give more fine-grain control to applications that needs to handle multiple threads in application-specific ways.

4.9 Drivers

Device drivers form the biggest part of any operating system and they have always been considered as the source of serious issues for the operating system. These problems stem from the fact that on one hand device drivers need enough privilege to access critical parts of the system, and on the other hand studies have shown that error rates in device drivers are three to seven times higher than the rest of the operating system [32]. Monolithic operating systems are most vulnerable to errors in device drivers, primarily because all device drivers are compiled and linked with other parts of the operating
system in one large kernel, and this means that even a very small bug in one of the
device drivers can crash the whole operating system. For example, studies show that
63-83% of all crashes in Windows XP are caused by device drivers [33, 34]. That is the
root of all reliability issues monolithic operating systems have long been tackling with.

Extensive studies have been done to isolate device drivers from other parts of the operating
system. Microkernels propose to isolate each device drivers into a separate process. Thus
if anything goes wrong with the device driver, other parts of the operating system will
not be affected and system can continue its execution and relaunch the crashed device
driver hoping it will not crash again.

Although this design can significantly boost the operating system reliability, it poses
some serious issues as well. The most important one is that a device driver by its nature
needs to access hardware device and some low level instructions, but most of the time
these privileged features are reserved for the kernel. For instance, virtually all the device
drivers need to have access to hardware interrupts. Isolating a device driver into a user
process may increase the reliability, but deprives the driver from accessing such privileged
features.

The solution which microkernel operating system adopt to address this problem is to
let the kernel and device drivers cooperate with one another to get the job done. For
instance, in MINIX3 [17] operating system, interrupts are caught by the kernel, and then
the kernel is responsible to send a message to the device driver informing it about the
external event. As expected these extra messages transmitted between the kernel and
the device drivers increases the overhead of responding to external events, and decreases
responsiveness of the operating system.

As Figure 4.21 (a) shows in an imaginary scenario of handling a key stroke, handling
an external events such as a keyboard interrupt in a pure microkernel operating system
like MINIX3 [17] is not that straightforward. In a microkernel architecture, since device
drivers are running in user land, they have no access to hardware interrupts. Therefore,
in this scenario first the kernel catches the keyboard interrupt, and sends a message to
the device driver to inform it about the receipt of the hardware interrupt. After being
informed, the keyboard driver needs to read the scan code of the pressed (or released) key.
Again this action requires privileges that are not available in user level. Consequently,
the keyboard driver asks the kernel (the only authority with enough privilege) to read
the scan code of the key from the keyboard internal buffer. Subsequently, the kernel
reads the scan code, and sends another message to the keyboard driver containing the
read value. At the end, the keyboard driver can look up its key map table to realize
what key has been pressed or released.
As this very simple example shows, to handle a key stroke, the keyboard driver needs multiple switches to the kernel. This might not seem to be a very serious issue in case of keyboard, because the time interval between key strokes are usually large, but for other devices like a network interfaces these extra message passing and context switches can considerably degrade the performance and responsiveness of the operating system.

What Savannah proposes has the best of both worlds, as device drivers are isolated into a separated virtual machines with direct access to required low level features. For instance, they have direct access to external hardware interrupts, as well as all privileged instructions they need to communicate and program the device they are working with. Therefore, all what a driver needs from the other parts of the operating system is to set up the execution environment before it actually begins serving the requests. For example, drivers of devices with registers mapped in MMIO area, need the PM to map that part of the MMIO into their address space before they can directly communicate with the device. But as soon as the execution environment is set up, device drivers can begin serving requests without any assistance from other parts of the operating system. This design guarantees both responsiveness and reliability for the operating system.
The example of Figure 4.21 (a) also depicts another issue in traditional designs of operating systems. That is, in such designs, unnecessarily the VFS is the middle man for almost all \texttt{read()/write()} requests from/to devices. In monolithic operating systems, this overhead is negligible (because device drivers and the virtual file server are all compiled and linked in the same address space and therefore no more communication or context switch is required) and might be compensated for the extra abstraction that this design can bring into the system, but for microkernel and multikernel designs, this design entails more message passing and context switches among different operating system components. For instance, back to our previous keyboard example, when finally the driver detects the pressed/released key, it sends another message to the VFS informing it about the new event, and only then the waiting process can receive the data from the VFS.

In the design of Savannah, we tried to put this into consideration as well. Thank to the channels explained in Section 4.4.2 user applications can directly receive data from device drivers without going through the File Server. This design has two important advantages: first, it reduces the number messages transmitted as well as the number times data must be copied before the end user application receives it, and second, it offloads the File Server and allows it to respond to requests which are genuinely file system related more quickly.

All in all, as Figure 4.21 shows, handling external interrupts in Savannah require much less messages to be transmitted and thus more performance, efficiency and responsiveness, while still offering the same level of reliability and modularity as a pure microkernel operating system.

### 4.9.1 Console Driver

![Figure 4.22: Communication between applications and console driver](image)

We have developed a very basic CGA driver for Savannah. Since there is no support for virtualization in Console devices, this driver is shared among all running processes, and
application send their text to the driver, and the driver prints the data on the screen in FIFO manner. Like all other drivers, application send their data to console driver through channels explained in Section 4.4.2. Figure 4.22 shows how this whole process works.

For debugging purposes, we also added few functions in library that writes directly in the part of memory associated to the CGA, so that there is no need of communication between the application and the console driver for debugging purposes.

4.9.2 Keyboard Driver

Device drivers in Savannah resemble to microkernels’ device drivers with two important differences: first, drivers in Savannah can directly communicate with their corresponding device without any support from other components in the operating system, and second, the communication between device driver and user applications is not mediated by the File Server.

```c
int main () {
    while (TRUE) {
        /* Get a request message. */
        r = driver_receive (ANY, &tty_mess, &ipc_status);
        if (r != 0)
            panic("driver_receive failed with: %d", r);

        switch ( _ENDPOINT_P (tty_mess.m_source)) {
            case HARDWARE:
                /* hardware interrupt notification */
                /* fetch chars from keyboard */
                if (tty_mess.NTIFY_ARG & kbd_irq_set)
                    kbd_interrupt(&tty_mess);
                break;
            default:
                /* do nothing */
                break;
        }
    }
    return 0;
}
```

Figure 4.23: Pseudo Code of a TTY driver in MINIX3

Figure 4.23 depicts a pseudo code of a TTY device driver in MINIX3 [17] operating system. Almost all device drivers and operating system servers in a pure microkernel operating system follow similar pattern which includes an infinite loop that continuously waits to receive a message or request followed by some steps and actions to deliver the appropriate response. The important point to notice is that in a microkernel operating system such as MINIX3, device drivers and operating system servers are running on top of the kernel and therefore are scheduled like other ordinary user processes. As a result, when they are waiting for an external event such as a hardware interrupt, they are blocked by the kernel.
This is one of the major differences between device drivers in Savannah and other microkernels or multikernel operating systems. There is neither a kernel pre-se nor a global scheduler in Savannah and after the operating system sets up the execution environment for device drivers, they have to manage everything by themselves.

Figure 4.25 depicts the pseudo code of the keyboard device driver in Savannah. This pseudo code clearly shows the very low level nature of device drivers in Savannah. In the very beginning, the driver sets up the GDT, interrupt vectors and then installs its own handler in IDT for the keyboard interrupts. Although this pseudo code might resemble to device drivers in microkernel, it is fundamentally different for various reasons. First, in microkernels, the device drivers usually waits for messages to come from kernel whereas in Savannah device drivers wait to receive their messages directly from user applications. Second, in microkernel operating systems, device drivers send the replies to the VFS whereas in Savannah they send the response directly to the user application.

This code needs to be blocked at two points: First, while waiting for messages to come from user application, and second when the driver waits for a key to be pressed. We already discussed about sending and receiving messages through channels in Section 4.4.2, so now we focus on how the driver blocks itself without needing any support from other operating system component.

The keyboard driver receives requests from user application which contains how many characters the user application wants to read from the keyboard. After receiving the request, the keyboard driver waits for the user to press a key. This is done in function 

\texttt{wait\_for\_completed\_request()}. Figure 4.24 shows the source code of this function. This function first enables interrupts, then puts the CPU in \texttt{HALT} mode. When the CPU is in \texttt{HALT} mode, it means that no more instructions will be executed, unless an external event like a hardware interrupt happens. Upon receiving the keyboard interrupt, the CPU exist the \texttt{HALT} mode and executes the interrupt routine which, in our case, is the function \texttt{kbd\_proc\_data()}. This is they way keyboard driver used to block itself. This is possible thank to special privileges available to drivers in our design.

In the keyboard hardware interrupt handle function \texttt{kbd\_proc\_data()}, the driver executes some \texttt{inb} instruction. This instruction is not allowed to all virtual machines by default, because each driver needs to work with some particular ports and not others. VMX \cite{35} technology provides some controls over what can be (un)set by the process manager to allow some virtual machines to execute such instructions on all or some specific ports directly and without existing the virtual machine, whereas if a virtual machine without such permission attempts to execute such instruction, it will cause a \texttt{VMEXIT}. More information about this is available in
void kbd_proc_data (void) {
  static bool shift_pressed = false;
  static bool ctrl_pressed = false;
  static bool alt_pressed = false;
  int scan_code;
  int ascii_code;

  if ((inb (KBD_STAT_PORT) & KBD_STAT_DATA_IN_BUF) == 0) {
    panic ("Data can’t be read from keyboard buffer\n");
  }
  scan_code = inb (KBD_DATA_PORT);

  /* ... Some sanity checks here */
  switch (scan_code) {
    case 0x1D: /* If CTRL is pressed */
      ctrl_pressed = true;
      goto eoi;
    case 0x38: /* if ALT is pressed */
      alt_pressed = true;
      goto eoi;
    case 0x2A: /* If any of SHIFTs is pressed */
      shift_pressed = true;
      goto eoi;
    case 0x1D | 0x80: /* If CTRL is released */
      ctrl_pressed = false;
      goto eoi;
    case 0x38 | 0x80: /* If ALT is released */
      alt_pressed = false;
      goto eoi;
    case 0x2A | 0x80: /* If either of SHIFTs is released */
      shift_pressed = false;
      goto eoi;
    default:
      break;
  }
  if (scan_code & 0x80) /* If the key is released */
    goto eoi;

  /* Find the key in keymap table */
  ascii_code = keymap[scan_code][((ctrl_pressed<<1) | (shift_pressed) |
               (alt_pressed<<2));
  buffer[bufpos++] = ascii_code;

  eoi:
  /* Set the EOI for local APIC */
  lapic_eoi();
  return;
}

/* ==-------------------------------------------------------------------- */
void wait_for_completed_request(char *channel, int count) {
  buffer = channel;
  while (1) {
    __asm__ __volatile__ ("sti; hlt
	"):
    if (bufpos >= count) {
      bufpos = 0;
      break;
    }
  }
}

Figure 4.24: Snippet of code from keyboard driver in Savannah
int main (int argc, char **argv)
{
    /* interrupts are disable at this point */
    struct read_char_rq req;
    struct channel *from;
    /* Setting up GDT and IDT */
    create_default_gdt();
    interrupt_init();
    add_irq(KBD_IRQ, &kbd_proc_data);
    /* Interrupts are still disable */
    while (1)
    {
        /* Wait for the request */
        from = cnl_receive_any();
        /* Make a local copy of the request */
        memcpy(&req, from->data, sizeof(struct read_char_rq));
        /* This will enable the interrupts */
        wait_for_completed_request((char*)from->data, req.count);
        /* Disable interrupts while we are copying the current
         * reply into the channel.
         */
        cli();
        cnl_reply(from, req.count);
    }
}

Figure 4.25: Main function from keyboard driver in Savannah

When enough keys are pressed by the user to finish the current request, the function
\textit{wait\_for\_completed\_request()} exits. At this point, to be in the safe margin, driver disables
interrupts and writes the reply of the user application into the channel. After writing
the data into the channel, current request is finished, therefore, the keyboard driver goes
to receive and handle the next request.

### 4.9.3 PCI Driver

The PCI driver is essential for probing, detecting, and getting information about devices
attached to the PCI bus. Savannah has a basic driver for the PCI, but the interesting
thing and difference between other drivers in Savannah (like the already mentioned
console and keyboard driver) is that the PCI driver is not running as an independent
process, but rather its implemented as a library that can be included and linked to other
device drivers that need it. The PCI driver is not the only driver implemented so, as
we already mentioned in the Section 4.8.1, the local timer driver is also implemented as
a function library so that every user process be able to include and link the library in
order to use the local timer.

Just like other drivers in Savannah, current version of our PCI driver is very basic and
only provides primitive functionalities that our other drivers essentially need to work.
These functionalities include routines for probing and detecting devices attached to the
PCI bus, getting some basic information about them such as the MMIO addresses, the
IRQ line(s), and so on and so forth. Moreover, to facilitates developing new device
drivers, PCI library includes some routines that automates the procedure of requesting the PM to redirect an IRQ to the device driver core, or asking the PM to map the MMIO area of the device into the address space of the device driver.

4.9.4 E1000 Driver

To provide a networking driver in Savannah we decided to implement a driver for the E1000 class of Intel networking cards. E1000 network interfaces provides many features that can be leveraged to improve networking performance of the system. We did not implemented a fully featured e1000 driver as described in specification [36] because it requires plenty of time and energy, rather we implemented just the basic functionalities to receive and send packets.

Unlike keyboard, console, and other simple devices, network interfaces typically include extensive internal features developed during the years for better networking performance, latency, etc. These extra features makes driver development further more difficult.

Because we are not aiming at developing a full featured network driver but rather doing a research in operating system design, we only implemented those features that are directly needed for our purposes. One of the decisions we had to make before beginning development of the driver was to choose whether we wanted our driver to work using hardware interrupts or in polling mode. Both are possible for network interfaces, but the results can be radically different.

Existing drivers for Linux support both modes and are intelligently switching among them to get the maximum performance with minimum CPU overhead and power consumption. This decision is important because when the network traffic is too high, storing and restoring CPU registers for handling hardware interrupts can cause trashing, therefore in such cases its better to leave the driver in polling mode. Since we wanted to test our operating system under very high loads, we made the decision to use polling mode in our network interface driver.

The driver we implemented uses the channels to communicate between the application and the driver. This means that when the application opens the netif special file, the file server automatically creates a channel between the application and the network driver. After that, every time it calls a read() instruction, the application blocks until a new packet arrives and the driver will copy the raw content into the channel. When something is written from the application in the channel using the write() function, the driver will copy the packet into the network card transmitter ring buffer. It is important to notice that the messages exchanged through the channel are RAW packets, since, as explained before, the driver just copies what is in the channel into the device. This means that the
packet should include not only the payload but also all the headers. It is the role of the application to use the LwIP library provided in the library to extract each header and discard or accept the packet. The other option is to compile the network driver with the LwIP library and to parse the headers before sending the payload to the channel. We decided to leave this decision to the application side.

Figure 4.26 shows a simple version of the code we implemented for our e1000 driver and now I am going quickly through it. Every time a new request comes in the channel, an integer is copied and the driver can check which kind of request it is. It can be either a NETIF_READ or a NETIF_WRITE request, which means that there is something to read or to write from/to the network card.

In the case of a NETIF_READ request, a loop is started to read the E1000_REG_TPR which saves the number of the total packet received, so that if two packet comes, this registers saves the number 2. With this loop, if one (or more) packet is missed, that number is stored in the missed_packets local variable and it exists from the loop. The register is automatically reset to zero from the device once it is read. After the loop is completed, the packet is read using the e1000_read() function and the length of the packet is returned and saved in the len local variable and the buffer buf is filled. When the packet is saved, the number of missed_packets is decreased and the buf is copied into the channel, and it is returned to the application with the number of bytes of the packet.}

In the case of a NETIF_WRITE request, the data is directly written to the network card with the help of the e1000_write() function, and the number of bytes written to the network card is returned in the channel.
Chapter 4. Implementation

Figure 4.27: Source code structure

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Blank</th>
<th>Comment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>129</td>
<td>5976</td>
<td>13098</td>
<td>40666</td>
</tr>
<tr>
<td>C Header</td>
<td>150</td>
<td>2446</td>
<td>6098</td>
<td>9747</td>
</tr>
<tr>
<td>make</td>
<td>22</td>
<td>186</td>
<td>151</td>
<td>595</td>
</tr>
<tr>
<td>Assembly</td>
<td>16</td>
<td>58</td>
<td>210</td>
<td>557</td>
</tr>
<tr>
<td>Bourne Shell</td>
<td>4</td>
<td>21</td>
<td>19</td>
<td>227</td>
</tr>
<tr>
<td>SUM</td>
<td>321</td>
<td>8687</td>
<td>19568</td>
<td>51792</td>
</tr>
</tbody>
</table>

Table 4.1: Source code statistics

4.10 Source Code

Figure 4.27 shows how the source code of Savannah is organized. Table 4.1 shows statistics about the overall source code, and as you can see most of the source code is written in C programming language.

The directory `boot/` contains the source code for booting up Savannah. Because of its special design, Savannah has a complex and multi-stage boot process. In Section 4.2 we have given a detailed description of how Savannah is booted up.

The source code of our library can be found under `lib/` directory. The library includes the source code for Savannah IPC routines explained in Section 4.4, some POSIX compatible string functions like `strcpy()`, or `strlen()`, plus some process-related services such as `fork()`, and `exec()`. Also filesystem-related functions like `read()`, and `write()` are implemented in the library. In addition our library includes a very special user level thread library which has been explained in detail in Section 4.8.3. The library also includes some routines to ease programming low-level devises such as local timer and handling hardware, and software interrupts. Finally it also includes a ported version of LwIP to easily parse raw packets from the network.

The `vms` directory contains the implementation of our essential VMs including `PM 4.5`, `FS 4.7`, and `INIT`. Under `vmm` directory, the source code for the `VMM 4.3` can be found.
The directory include/ contains the header files that can be used by all parts of the system. The bin/ contains the source code of some user applications that we used for our benchmarks.

Current version of Savannah, does not have hard disk support; instead, it uses a ramdisk as explained in Section 4.6. The source code of the program to create this ramdisk image is under the initrd/ directory.

The source code also includes a tools/ directory which contains some files, and programs that are not directly part of the source code, but they are either used during the boot process or they are used to automatically generate some header files.

Finally, implementation of device drivers are under drivers directory. As explained in Section 4.9, we have developed four basic drivers: Console 4.9.1, Keyboard 4.9.2, e1000 4.9.4 and the PCI 4.9.3. All of those are stored in sub folders of the drivers/ folder.

4.11 Compilation

We used both GCC 1, and CLANG 2 for the compilation of Savannah. Moreover, we compiled and and tested Savannah under all optimization levels both with GCC, and CLANG. GCC and CLANG suggest four general optimization level (-O0, -O1, -O2, -O3 3) that respectively increase the optimizations applied on the code.

Despite the significant performance improvements that code optimization can bring in, it can also create serious bugs in the final output binary code, particularly when its applied on very low level parts of the operating system such as memory management, or boot process. To avoid this, the programmer should be aware of the optimizations and write the code very carefully to guide the compiler about the parts that should not touched or the parts that the compiler should not make any assumption about the content of a memory address. During the development of Savannah, we faced many times bugs that were created because of the code optimization.

Code shown in Figure 4.28 is an example where high optimization level caused a bug in our code. In this code snippet, local variable gdtr is defined as volatile. This is a hint to the compiler not to make any assumption about the value, and the usability of this variable. If not defined as volatile, the compiler may decide to remove this variable altogether assuming its not used. This is likely because the only usage of this variable is as a pointer in the inlined assembly code snippet, and the compiler has no way of

---

1version 4.6.4
2version 3.0
3There are also -O which is equal to -O1, and -Os which optimized for code size
void create_default_gdt (void)
{
  size_t sizeof_gdt;
  static struct system_descriptor gdt[NGDT] __aligned (0x10);
  volatile struct descriptor_register gdtr __aligned (0x10);

  /* ... Some stuff here ... */
  gdtr.dr_limit = sizeof_gdt - 1;
  gdtr.dr_base = (virt_addr_t)gdt;

  /* Reloading GDT */
  __asm__ __volatile__ (;
    "cli\n"
    "lgdtq (%0)\n"
    "movl %1, %ds\n"
    "movl %1, %es\n"
    "movl %1, %fs\n"
    "movl %1, %gs\n"
    "movl %1, %ss\n"
    "pushq %%fs\n"
    "pushq %%gs\n"
    "pushq %%ds\n"
    "pushq $1f\n"
    "lretq\n"
    ::"r"((virt_addr_t)&gdtr), "r"(DSEL), "I"(CSEL) );
}

Figure 4.28: Potentially vulnerable code to optimization

understanding the semantic of inlined assembly codes. Nonetheless, it is important to note that the behavior of compilers differ from version to version in terms of optimization and code generation.

For our benchmarks however, we chose GCC, and applied -O3 optimization level. The reason behind preferring GCC over CLANG is that GCC has better support for vectorization optimizations than CLANG does. Asking the compiler to vectorize string operations and memory movements wherever it can, usually leads to significant performance gains. Traditional operating systems are usually reluctant to use vectorization and such CPU extensions due to the high cost of saving and restoring the long registers of these CPU extensions in every context switches and interrupt handling. Because in the design of Savannah, we pin processes to cores, and this eliminates the cost of context switches altogether, hence, there is no reason not to use them. Therefore, we enable all these extensions in Savannah, and both in our coding and in compilation phases, we give hints to compiler about the parts we wanted to be vectorized.

The result is fascinating. The vectorized version of memcpy() function outperforms its non-vectorized version almost an order of magnitude. Similar behavior was also observed in some other memory movement and string operations. We were particularly interested in improving the performance of memory movement functions, as they are one of the major sources of latency in operating systems. Therefore, we could gain great
performance improvement without changing the code, and only letting the compiler optimize the code during compilation.
Chapter 5

Evaluation

In this section, we evaluate multiple parts of our system with some microbenchmarks showing how important parts of the system perform and the benefits this can have for a real world application that needs low latency.

All tests were performed on a single socket machine using a quad-core Xeon E3-1270 with hyperthreading. All the cores run at 3.4 GHz.

The system has 32 GB of memory, and three levels of cache. Hyperthreads on each core share L1 and L2 cache while the L3 cache of 8 MB is shared between all the cores on the die.

5.1 Inter Process Communication

As explained in Section 4.4, the only way process can communicate in our design is by using IPC. We have two basic ways of message passing and three ways of notifying a VM that a message is ready. The notification mechanism has a direct impact on how fast the processes can exchange messages. In this section we evaluate all three approaches and discuss their pros and cons. We also measure the difference of messaging between different cores or just between different hyperthreads in the same core.

Our benchmark is a so-called request-response. We send a message from one process to another and we wait for a reply in the process that initiates the communication. The request-response communication is depicted in Figure 4.5. The first process writes a message in its outbox, notifies the receiver, and then waits for notification from the receiver that the reply is ready. We perform the request-response 10,000 times and, using the timestamp counter (TSC), we measure the total time of the test.
Chapter 5. Evaluation

The same core (cycles) | Cross cores (cycles)
---|---
Polling | 194 | 320
Polling MWAIT | 2700 | 2310
IPI | 1641 | 1546

Table 5.1: Comparison of IPC notification in Savannah

Table 5.1 shows the results of our benchmarks. Polling is the method with the least latency, but uses the most energy. When achieving low latency is the goal, no matter how much energy is used, polling wins. When energy usage matters, we can use MWAIT, which halts the CPU. Waking up the CPU has some latency, which shows up as a longer round trip time. MWAIT can also put the CPU in a lower-power state but the wakeup latency then increases. The last method is sending IPI. To our surprise, using IPI in this request-response benchmark proved to be a little faster than using MWAIT. Since each of these methods has its advantages and disadvantages, there is no clear winner and there must be an agreement between the communicating parties on which method to use.

Many modern processors, including the one we used for evaluation, employ hyperthreading. This means that there are several hardware execution threads on a single core. Intel products feature two. The hyperthreads look like full cores, however they are not since they share one execution pipeline. They also share the L1 and L2 cache, which has impact on the latency of our messaging.

In polling mode, it is faster to notify another process on the same core. We attribute it to the fact that communication channel between hyperthreads on the same core is in the cache they share while cross-core messaging goes through L3 cache.

On the other hand, we observed the opposite effect in the case of using MWAIT or IPI. We guess that since both hyperthreads share the same execution pipeline, probably the internal scheduler of the core is responsible for the delay.

5.2 Interrupt latency

Real-time applications need to run on a system with two important properties: low interrupt latency and good predictability. In our system, the predictability is implicit because the application can have a dedicated core with no competing threads. In this section, we show that by eliminating the operating system and having hardware interrupts delivered directly to the VM, we can greatly reduce jitter compared to other systems.
We carried out three tests to determine the interrupt latency jitter in Savannah and compare it to vanilla Linux (version 3.8.13) as well as to the same version of Linux including the Real Time patch [3] (Linux-RT). We ran the benchmark application with the highest priority on a dedicated core with FIFO scheduling algorithm both on Linux and on Linux-RT.

POSIX offers several way of using timers and we picked two. We used `nanosleep()` since it is the single-shot timer with the highest precision. We also evaluated `time_create()`, which creates a periodic (interval) timer. To have a fair comparison with the other operating systems, we implemented both in a POSIX compatibility library which we linked to our test application in Savannah. We used the local APIC timer either in single-shot or in periodic mode.

We used `nanosleep()` to wait 1 millisecond in a loop. We read the TSC before and after the `nanosleep()` and we saved the sample in a large array. We executed the loop 10,000 times, after which we calculated the jitter as the standard deviation of all the results in the array. As Table 5.2 shows, the results in both Linux and Linux-RT are in the order of microseconds. Table 5.2 shows the same results in Linux and Linux-RT when when the idle task in Linux is a busy loop. This does not halt the CPU and thus avoid the latency of waking up the CPU, which has significant impact on the jitter. The default Linux has jitter in the range of several microseconds while using the busy loop as the idle task pushes it down to tenth of a microsecond. In contrast, the jitter in Savannah is just a few nanoseconds, two orders of magnitude better than the best Linux can do. For applications that require very short latency, this is clearly a big win.

`Nanosleep()` is not the best choice if an application needs to set a timer for a precise interval. Because there is some delay in setting up a single-shot timer and some delay delivering it, `nanosleep()` can cause some drift. Instead, `settimer()` and `timer_create()` are the preferred choice. Unlike `nanosleep()` which “just” returns from the call, the interval timers use signal which the kernel delivers to the process whenever the timer expires.

<table>
<thead>
<tr>
<th>Function</th>
<th>Linux</th>
<th>Linux-RT</th>
<th>Linux</th>
<th>Linux-RT</th>
<th>Savannah</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nanosleep</code></td>
<td>6.91</td>
<td>6.76</td>
<td>0.84</td>
<td>0.076</td>
<td>0.0044</td>
</tr>
<tr>
<td><code>settimer</code></td>
<td>1728.46</td>
<td>0.27</td>
<td>1728.46</td>
<td>0.18</td>
<td>-</td>
</tr>
<tr>
<td><code>timer_create</code></td>
<td>7.61</td>
<td>2.94</td>
<td>0.57</td>
<td>0.12</td>
<td>0.0022</td>
</tr>
<tr>
<td><code>timerfd_create</code></td>
<td>10.24</td>
<td>12.21</td>
<td>9.89</td>
<td>9.92</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison of jitter of timer functions in Linux, Linux-RT and Savannah (in µs)
Figure 5.1: Jitter of Savannah, Linux and Linux-RT
This eliminates the drift problem, however, it does not remove delayed delivery. Another Linux-specific function that has been designed for this purpose is timerfd_create(). This function uses file descriptors instead of signals. In Savannah we provide a library function with the same interface as timer_create(). We program the local APIC timer in periodic mode which gives us an interrupt every 1 millisecond. Every time we receive the timer interrupt, we can immediately execute the equivalent of the POSIX signal handler without the need to wait for the operating system to translate it for us.

We carried out the previous tests on an idle system, but we also performed a test when the system was under pressure by a ping flood. Figure 5.1 shows the results. In Savannah the result is basically the same when system is idle since the ping-flood is steered to the core which hosts the network driver and stack. Thus, it does not affect the real time application at all. However, it is a different story in Linux. It is interesting to notice that in Linux when the system is under pressure the jitter decreases significantly. The reason is that when the network traffic is high, the Linux network driver stays in polling mode. Hence, CPU does not need to wake up again when the timer interrupt shows up. In addition, staying in polling mode causes the scheduler to run periodically and actively checks whether a process became runnable. Consequently, little time is wasted and the chance that when the timer expires, CPU is granted to the real time application increases.

### 5.3 High Frequency Trading

High Frequency Trading (HFT) is a way of trading securities very quickly, many times per second, using computers. In such a business, a millisecond is almost eternity. The trading system must have access to the actual market information to decide to buy or sell. This information is pushed to traders by the markets via network. A trading algorithm makes a decision and sends it to the market as quickly as possible. To avoid that the decision is stale by the time it reaches the market, the traders require extremely low latency within the network. They reduce it by placing their trading machines close to the market’s data centers and they try their best to reduce the latency within the devices they use as well as inside their the software stack [37, 38]. In this section, we are going to show that our operating system design can significantly reduce the network latency.

To prove our claim, we run a simple HFT application and feed it through network with market updates and compute the average latency in every step of the receipt and transmission for 10,000 packets. All incoming and outgoing packets were small TCP packets of less than 1KB in size.
It is important to note that our focus in this test was merely on latency in software, not hardware, therefore the latency we compute is not wire-to-wire, but rather NIC-to-NIC. According to [39], as long as NICs are connected to the processor via PCI bus, the communication between the CPU and the NIC, causes significant latency. That statement matches with our observations in our benchmarks. We found that reading the content of only one register of E1000 NIC can take almost 1µs. However, we believe with virtualization penetrating into network interfaces [7], our design can benefit more from the latest improvements in the hardware than other operating systems, because this means Savannahs’ processes can receive data directly and without any mediation.

![Diagram of latency in HFT benchmark]

**Figure 5.2:** Breakdown of latency in HFT benchmark

We begin measuring the latency from the moment that the driver starts writing the incoming packet into the channel. This implies a `memcpy()` operation. As you can see
in the Figure 5.2 this happens very fast. This can be explained by the fact the market update packets are usually fairly small and Savannah uses a highly optimized memcpy() using vectorization. After writing the packet into the channel, the driver notifies the application about the pending packet. Because latency is the major concern here, we configured the system to use polling method for IPC. The numbers shown in Figure 5.2 resemble the result shown in the Table 5.1. Our HFT application is linked with an unchanged LwIP network stack, so after being notified by the driver, HFT gives the packet to LwIP. Our results also show that LwIP network stack introduces very small latency.

Excluding the latency introduced by the HFT application itself, we have a latency of 790ns NIC-to-NIC. HFT applications use FAST protocol that has a well known and well studied latency [38, 40]. The latency of current implementations of FAST plus HFT applications usually have a latency in the range of 1 µs. We do not include the latency of our HFT application here, because we don’t have access to any real-world HFT application to measure and compare the latency.

Rumble et al. [39] argues that on average, the latency caused by the operating systems in networking is almost 10µs. Of course, these numbers might differ from one operating system to another, but they are generally in the same range.

The latency we showed in this section is not the absolute minimum latency achievable in Savannah, because we wanted to show a latency less than 1µs is achievable even for normal applications. But in justified cases where low latency has the highest priority, its possible to link driver, network stack, and HFT application altogether on a dedicated core. This eliminates the latency of channels, and also and few memory copies. Our measurements show that in such a case latency decreases to 500 ns NIC-to-NIC.
Chapter 6

Conclusions

User applications show a broad variety of demands, and expect the operating system to be able to respond to all their demands with maximum performance and efficiency. Although existing operating systems can respond to a large number of user demands, an increasing set of applications are not satisfied with the services they get from the existing operating systems.

In this master thesis, we began with reviewing some of the most successful operating system designs to point out that they are not suitable for some latency sensitive, and real-time applications. We argued that this problem stems mainly from the role defined for the operating system in these designs, which is the provider of isolation among processes, and the resource multiplexer. These roles entail that the operating system would need to run underneath the user applications, meaning that the operating system has to make some decisions on behalf of the user application. This does not give the user applications the possibility to get the best out of the hardware they are running on because the operating system does not know what is best for each application, and therefore, its has to make compromises.

We presented and described our novel system architecture which abandons the traditional ring protection mechanism, and adopts virtualization as its main building block to allow the operating system to run side by side with the user applications and allow them to take better advantage of the hardware they run on. We discussed how real-time and latency sensitive applications as well as legacy applications, can enjoy running without any unwanted operating system effects and interferences in their execution. In addition we showed how our novel design can allow user applications to manage their threads more flexibly by utilizing their ability of having fast access to local timer interrupts.
At the end, we presented that our benchmarks show by running real-time applications in a bare virtual machine, the jitter of timers is reduced by two orders of magnitude, even when compared to a Linux system tuned for real-time. We also demonstrated that it is possible to achieve very low latency networking due to the fact that the application has direct access to the hardware and does not have to wait for the packets to go through the operating system.
Bibliography


