

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

UMView,
a Userspace Hypervisor Implementation

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Antonio Cardace

Sessione III
Anno Accademico 2015/2016

*No matter what people tell you,
words and ideas can change the world.*

Robin Williams

Introduzione

Le macchine virtuali sono usate in una grande varietà di contesti e applicazioni come Java, Docker, KVM e QEMU. La virtualizzazione è uno dei concetti più interessanti e flessibili del mondo dell'Informatica.

Questo elaborato descrive il lavoro svolto per implementare una nuova versione di UMView. Imparando dai prototipi esistenti creati dal team di Virtual Square una nuova codebase è stata sviluppata con l'obiettivo di essere un punto di riferimento per gli esperimenti futuri sul tema della virtualizzazione parziale.

UMView è una macchina virtuale parziale e un hypervisor in userspace capace di intercettare le system call e di modificare la visione che i processi hanno del mondo esterno. UMView dispone inoltre di un'architettura a plugin in grado di caricare moduli in memoria a runtime risultando in questo modo flessibile e modulare.

Introduction

UMView in particular is the implementation of the View-OS concept which negates the global view assumption which is so radically established in the world of OSes and virtualization.

Virtualization and virtual machines are nowadays used in a variety of applications, from Java to Docker, KVM and QEMU. Virtualization is one of the most interesting, flexible and powerful concepts in computer science.

This dissertation describes the work done to make from scratch a new implementation of UMView. Learning from the existent prototypes created in the past from the Virtual Square team a new modular codebase has been written in order to be the foundation of future experiments on partial virtualization.

The software UMView in particular is the implementation of the View-OS[9] concept which negates the global view assumption which is so radically established in the world of OSes and virtualization.

UMView is a partial virtual machine and userspace hypervisor capable of intercepting system calls and modifying their behavior according to the calling process' view. In order to provide flexibility and modularity UMView supports modules loadable at runtime using a plugin architecture.

Contents

Introduction	iii
1 About Virtualization	1
1.1 Abstraction	1
1.2 Virtualization	3
1.3 Virtual Machines	3
1.3.1 System VMs	3
1.3.2 Process VMs	4
2 Virtualization Today	7
2.1 QEMU	7
2.2 KVM	9
2.3 Linux Kernel Namespaces	11
2.4 Docker	13
2.5 VDE	14
3 A Novel Approach	17
3.1 The Global View Assumption	18
3.2 View-OS	20
3.3 UMView	21
3.3.1 UMView Features	22
3.3.2 Partial Virtualization	23

4	UMView for Users	25
4.1	Starting UMView	25
4.2	Working with Modules	25
4.2.1	Listing Modules	26
4.2.2	Loading Modules	27
4.3	The Unreal Module	28
4.4	A Useless Module	29
4.4.1	Removing Modules	31
5	UMView for Developers	33
5.1	Modules API	33
5.1.1	Declaring a Service	33
5.1.2	Using the Hashtable	35
5.1.3	Creating Virtual File Descriptors	37
5.2	UMView System Calls	38
5.2.1	Redefining System Calls	38
5.2.2	Registering System Calls	39
5.2.3	System Calls Grouping	39
5.3	Writing Modules	41
5.3.1	Hello UMView	43
5.3.2	Module Constructor	43
5.3.3	Module Destructor	44
5.3.4	Building Modules	45
5.3.5	Testing and Debugging	46
6	UMView Internals	49
6.1	The Tracing Unit	49
6.1.1	Ptrace	49
6.1.2	Intercepting System Calls	51
6.2	Virtual System Calls	53
6.2.1	The UMLib	53
6.2.2	Implementation	54

6.3	The Plugin Architecture	54
6.3.1	Modules as Shared Objects	54
6.4	The Guardian Angels Technique	55
6.4.1	Implementation	56
6.5	The Hashtable Unit	57
6.6	Fetch-Execute Unit	60
6.6.1	The Fetch Unit	60
6.6.2	The Execute Unit	63
6.7	Purelibc	69
6.8	The Path Canonicalization Unit	71
6.9	The Epoch Unit	73
6.10	The FS Unit	75
6.10.1	FS Structure Creation	77
6.11	The FD Table Unit	77
6.11.1	Tracking File Descriptors	79
6.11.2	FD Collisions	80
6.11.3	FD Table Creation	81
6.11.4	FD Table API	81
6.12	The UMView System Call Table	82
6.12.1	UMView Configuration File	83
6.12.2	Build Scripts	83
6.13	The Arch System Call Table	85
6.13.1	CFunction and WFunctions Resolution	85
	Conclusions and Future Developments	87
	Bibliography	89
	Acknowledgements	91

List of Figures

1.1	Computer system architecture layers [10]	2
1.2	VMM and virtual machines	4
1.3	OS architecture example	5
2.1	QEMU running different OS on one machine	8
2.2	KVM simplified architecture[8]	10
2.3	Docker architecture	13
3.1	Example of a *nix filesystem structure	19
3.2	System call example	21
4.1	xterm inside UMLView	26
4.2	um_lsmod command	27
4.3	um_insmode command	28
4.4	The unreal module	29
4.5	um_rmmod command	32
5.1	System call grouping	41
6.1	ptrace tracing	50
6.2	Virtualizing a system call	52
6.3	UMLView guardian angels	56
6.4	UMLView Architecture	61
6.5	CFunction and WFunctions resolution	85

List of Tables

5.1	UMView supported system calls	42
-----	---	----

Chapter 1

About Virtualization

1.1 Abstraction

Abstraction is among the foundation ideas of the world of computing as we know it today, thanks to this idea we can design computer systems with well-defined interfaces that separate levels of complexity.

Using well-defined interfaces help developers building complex and modular software architectures with ease, the trick is to exploit abstraction to hide lower-level implementation details thereby simplifying the design process. One straightforward example of how important abstraction is to computer scientists resides in what makes the machines we use every day so flexible and incredibly helpful, the operating system, this utterly essential piece of software abstracts all the complexity of the physical machine and exposes a nice and easy API¹ that application software developers can use to build their application without ever having to take care or worry about paging, scheduling or many other classic problems we face when we design an OS.

¹application programmer interface

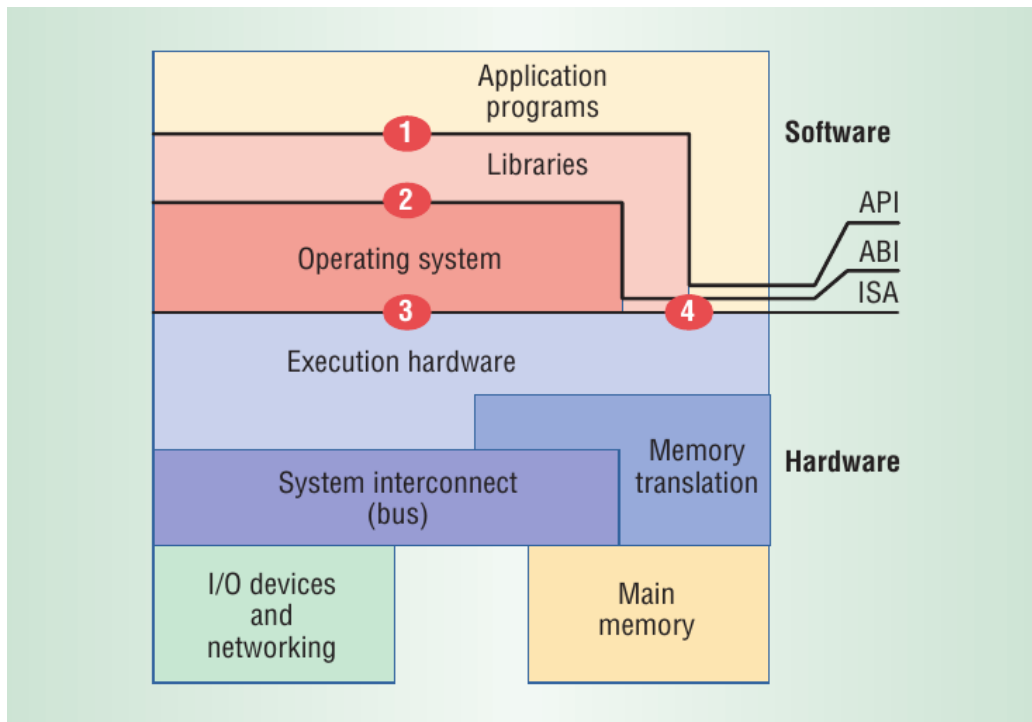


Figure 1.1: Computer system architecture layers [10]

This is just a brief and pragmatic look on the concept of abstraction, nevertheless it undeniably proves the value of this idea and its importance as a paradigm in computing. However every grand idea has its trade-offs and abstraction is no exception, building systems with well-defined interfaces makes components and subsystems designed to specification for one interface unusable with others designed for another.

Let's give another example, applications compiled for a x86_64 architecture will not work on a ARM machine, the same thing applies to applications compiled for different OSes even on the same computer architecture, this is because of dependencies, application binaries depends on a specific ISA² and the operating system interface they have been compiled for.

²instruction set architecture

1.2 Virtualization

Virtualization help us getting around the limitations faced by abstraction, specifically it manages to achieve interoperability between systems or subsystems designed to different specification.

Virtualization can thus be applied to I/O devices, memory, or any other subsystem of a computer architecture, furthermore it can be applied to the entire machine, this last example bring us to the concept of virtual machines.

Virtual machines can be developed adding software layers to the real machine in order to support the target architecture, in such a way we can overcome the limits imposed by well-defined interfaces.

1.3 Virtual Machines

Following the taxonomy made by Smith, J. E. and Nair, R.[10], we can decompose the concept of virtual machines in two macro classes, which are *process VMs*³ and *system VMs*.

1.3.1 System VMs

A system VM provide a consistent and complete environment in which it can support a full operating system, this guest OS is allowed access to virtual hardware resources such as networking, I/O and perhaps even a GPU.

A system VM implies the presence of a VMM⁴ which runs in the most highly privileged mode, while all guest systems run with reduced privileges so that the VMM can intercept and emulate all guest operating system actions that would normally access or manipulate critical hardware resources.

³Virtual machines

⁴Virtual machine monitor

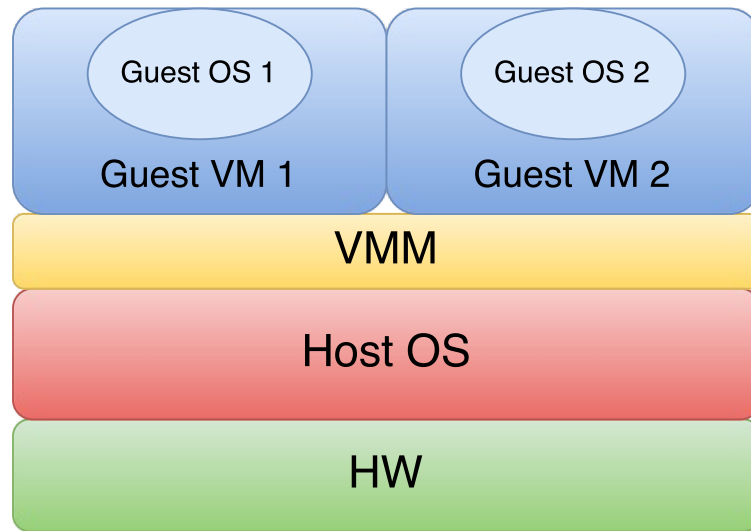


Figure 1.2: VMM and virtual machines

1.3.2 Process VMs

Process VMs provide a virtual environment for user applications. OSes for instance implement this kind of virtualization to offer a simplified view of the machine to user applications (fig. 1.3), virtual memory is such an example, application and compiler developers do not have to worry about physical addressing they can just rely on the OS doing the hard work.

The OS, exploiting virtualization techniques, gives any process the illusion of having the complete machine to itself, as a matter of fact each process has its own space address, its set of open file descriptor and registers. What the OS is really doing to achieve this is sharing the hardware resources by providing a replicated process-level VM to each executing application.

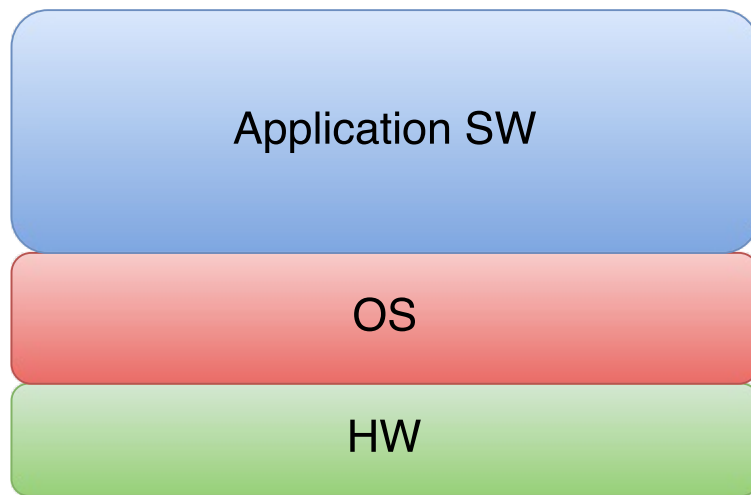


Figure 1.3: OS architecture example

Chapter 2

Virtualization Today

Today virtualization software is among the most flexible and useful for developers and users. In this chapter we are going to do a brief survey about the most famous and used virtualization projects.

2.1 QEMU

QEMU (short for Quick Emulator) is a free and open-source hosted hypervisor which performs hardware-assisted virtualization and emulation[13].

It emulates CPUs through dynamic binary translation and can also be used purely for CPU emulation for user-level processes, allowing applications compiled for one architecture to be run on another. Moreover it can be used together with KVM in order to run virtual machines using hardware-assisted virtualization.

QEMU has multiple running modes:

- **User-mode emulation:** this mode is capable of running Linux programs compiled for a different ISA.
- **System-mode emulation:** this mode emulates a full computer system providing access to different OSes on a single machine.

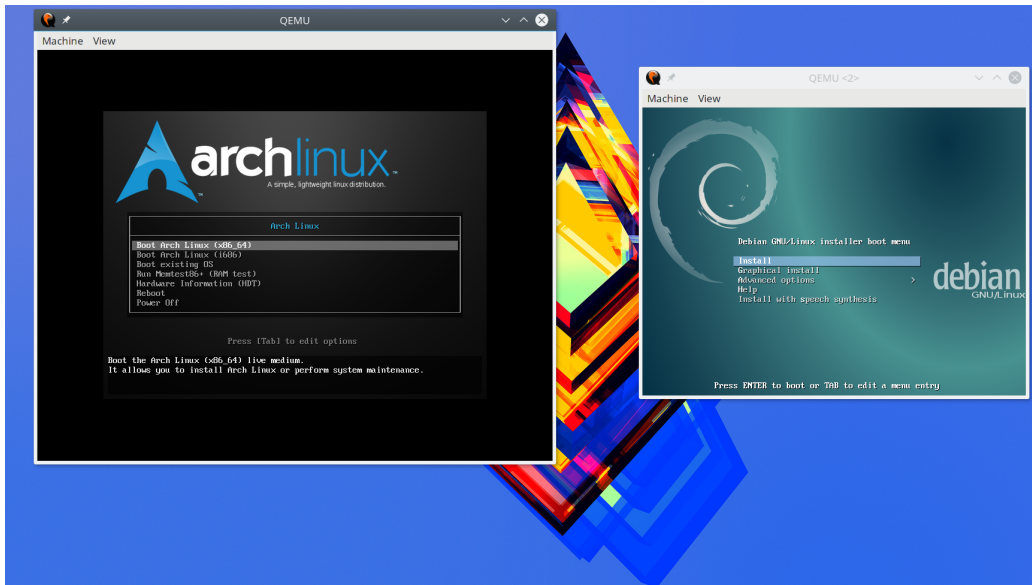


Figure 2.1: QEMU running different OS on one machine

- **KVM-mode:** QEMU can use KVM to run several OSES on a single machine compiled for the same ISA on which QEMU is running, this mode achieves near-native speed.

QEMU's flexibility makes it an essential tool for system and Kernel developers, in this way there's no need to reboot every time a small change to the Kernel is made or when testing new snippets of code, moreover QEMU can be attached to a GDB session allowing developers to precisely observe what the machine is actually doing.

QEMU can also provide a different ISA/ABI for user applications allowing to execute programs compiled for different architectures.

QEMU is not only useful to developers but to users as well, they can now run multiple different OSES on the same machine and since KVM support kicked in speed is not necessary a trade-off in spite of flexibility.

2.2 KVM

Kernel-based Virtual Machine (KVM) is a virtualization infrastructure for the Linux kernel that turns it into a hypervisor. It was merged into the Linux kernel mainline in kernel version 2.6.20[12].

KVM permits to have virtual machines running at near-native speed although it needs hardware support to run on a specific architecture.

Currently KVM runs on x86, S/390, PowerPC, IA-64 and ARM.

KVM also supports paravirtualization for certain devices, paravirtualization is a virtualization technique that presents a software interface to virtual machines that is similar, but not identical to that of the underlying hardware, the intent is to reduce the portion of the guest's execution time spent performing operations which are substantially more difficult to run in a virtual environment compared to a non-virtualized environment.

KVM is essential for building secure and flexible cloud infrastructures, as a matter of fact it is the core of many complex cloud administration softwares (OpenNebula, OpenStack, etc...) since its ability to execute full system virtual machines at near native speed.

How KVM works essentially is through a special device `/dev/kvm` which provides a virtual environment called `virtual-cpu`, using this device application software can run code in a hypothetical CPU ring 1 (between user-land and kernel-land). This `vcpu` will execute code in its assigned memory region as the real CPU would normally do and will eventually block or generate some kind of software interrupt when the `vcpu` is trying to execute any of the reserved instructions¹.

¹This set of instructions is configurable but restricted, based on the HW implementation.

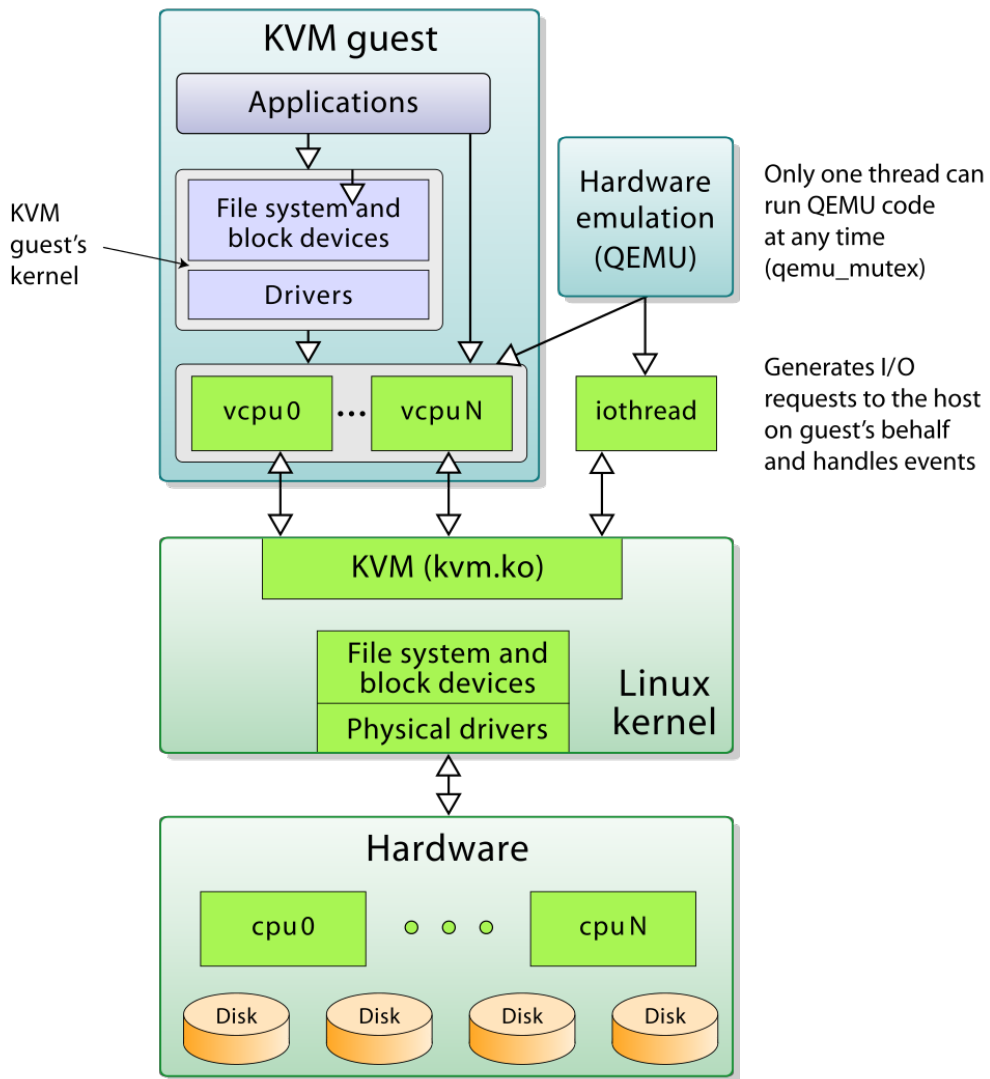


Figure 2.2: KVM simplified architecture[8]

Once KVM reaches a reserved instruction gives control to the userspace application using that vcpu along with a notification of what caused this stop, (this is exactly what happens when QEMU is used in conjunction with KVM) at this point the application software can decide on how to manage this specific stop.

As an example if the application detects a I/O event related stop can emulate the I/O device in question and continue with the virtualization.

There is a special note to be done on QEMU+KVM, the software running using hardware-assisted virtualization starts running in a virtual environment as if a real machine just booted thus this type of software (usually QEMU) will have to supply a BIOS and a bootloader because, for example, on a x86 architecture the vcpu will boot in a 16-bit environment trying to load the BIOS from the ROM.

This last consideration was done to explain that although KVM is a great piece of software its use on the subject of virtualization is very specific, it can solely run full virtual machines, no other options are available.

In some ways this is a pity because having virtual machines (both process and system) running at near-native speed is very appealing but it just cannot be done with this technology, the responsibility is not on KVM obviously but on the hardware virtualization platforms designed by HW engineers.

2.3 Linux Kernel Namespaces

Linux namespaces as one can easily deduce from the name are a set of features specific of the Linux Kernel. A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes[6].

To move a process into a new namespace on Linux we can use the system call *unshare(int flags)*, according to the supplied *flags* argument the process will be put in a new namespace where some of its properties are unshared. Currently the *flags* values supported on Linux are:

- **CLONE_NEWCGROUP**: this flag isolates the cgroup root directory;
- **CLONE_NEWIPC**: this flag isolates system V IPC, POSIX message queues;
- **CLONE_NEWNET**: this flag isolates network devices, stacks, ports, etc.;
- **CLONE_NEWNS**: this flag isolates uount points;
- **CLONE_NEWPID**: this flag isolates process IDs;
- **CLONE_NEWUSER**: this flag isolates user and group IDs;
- **CLONE_NEWUTS**: this flag isolates hostname and NIS domain name;

Namespaces are thus a tool to achieve process-level VMs, processes inside different namespaces will be completely isolated.

In a way is like having a sandbox in which a process can play, as a matter of fact namespaces are mainly used to implement containers and therefore used as a security feature.

Unfortunately creation of new namespaces in most cases requires the **CAP_SYS_ADMIN** capability unless the **CONFIG_USER_NS** options is enabled when compiling the Kernel.

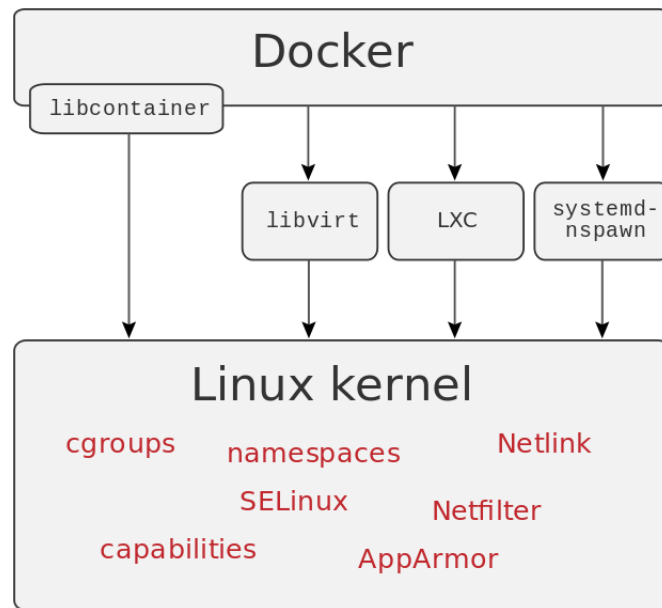


Figure 2.3: Docker architecture

2.4 Docker

Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries and anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment[11].

Docker use Linux namespaces to implement containers and isolate an application's view of the operating environment, including process trees, network, user IDs and mounted file systems.

Docker is widely used for automation, continuous-integration and deployment of applications, it clearly is another example of a process-level VMs creation software.

2.5 VDE

VDE is a sort of Swiss knife of emulated networks. It can be used as a general Virtual Private network as well as a support technology for mobility, a tool for network testing, as a general reconfigurable overlay network, as a layer for implementing privacy preserving technologies and many others[3].

VDE is a virtual infrastructure which gives connectivity to several kinds of software components:

1. emulators;
2. virtual machines;
3. real operating systems;
4. any tool operating on networks;

This software allows to create a completely virtual/emulated networking space by using virtual machines on a set of real computers connected by a real network. VDE virtualizes the main components of a standard Ethernet network we all are accustomed with[2]:

- **vde_switch**: a virtual Ethernet switch;
- **vde_plug**: Like an Ethernet plug, it is designed to connect two vde_switches. Everything that is injected into the plug from standard input is sent into the vde_switch it is connected to;
- **vde_plug2tap**: it is another plug tool that can be connected to vde_switches. Instead of using standard input and standard output for network I/O

everything coming from the `vde_switch` to the `plug` is redirected to a specified tap interface;

- **slirpvde**: it is a slirp interface for VDE networks. It acts like a networking router connected to a `vde_switch` and provides connectivity from the host where it is running to virtual machines inside the virtual network;

VDE with its many tools is a clear example of how virtualization leverages on flexibility, as a matter of fact being a multi-purpose tool for networks there are just no limits for the range of applications. One simple application could use VDE to interconnect various local virtual machines creating a virtual network, but as stated before the possibilities are endless.

Chapter 3

A Novel Approach

When talking about virtualization is common to think about virtual machines and when talking about virtual machines the first applications coming to mind are QEMU, KVM and the Intel VT-x technology.

Associating these softwares and these concepts of virtualization is obviously correct but in a way reduces and simplifies the concept of virtualization to just system-level virtualization because the main applications from the industry world, namely the cloud, mainly leverage on full machine virtualization.

Full virtualization is a useful concept and has brought powerful applications, nevertheless related to the pure idea of virtualization lacks of flexibility, of course you can have different machines running on a single real one, but this is how far away it goes.

These VMs cannot be decomposed, to have a process thinking of belonging to a virtual environment in such a way it is expensive because in order to spawn a VM we need a BIOS, a bootloader, a separate Kernel, separate libraries and separate user applications, therefore we end up needing a full system just for one process.

Is this the power of VMs or can we achieve more? We can bring in process-level VMs tools like the Linux Kernel namespaces, they can actually make a process think to be in some separate virtual environment with its own set of hardware and software resources, no separate Kernel or BIOS or bootloader is required this time, but then again this is just containing the process in some kind of a sandbox (as a matter of fact they are mainly used to implement containers), it is undoubtedly useful but this approach still seems limited.

First to implement this feature we are increasing the Kernel size and by adding more Kernel code we are increasing the possibilities of creating new bugs in the operating system.

It has been estimated that to implement the Linux Kernel namespaces between 7% and 15% of the core Kernel code would have had to be modified[1], this is a path of giant dimensions to be introduced in a modern Kernel.

Another limit of Linux namespaces is the requirement for a process to have the `CAP_SYS_ADMIN` capability to create a namespace (in most cases).

3.1 The Global View Assumption

The concept of a *global view* in computing has always been applied, every process has got the same vision of the resources of the operating systems, if some new resource becomes available or an already existing one changes this shift of state is visible by any running process.

A perfect example is the filesystem, every process sees the same underlying structure, if a new file is created every process sees it if and old one is removed no process is able to open it again, in short every change to the tree structure of the filesystem affects every process of the machine.

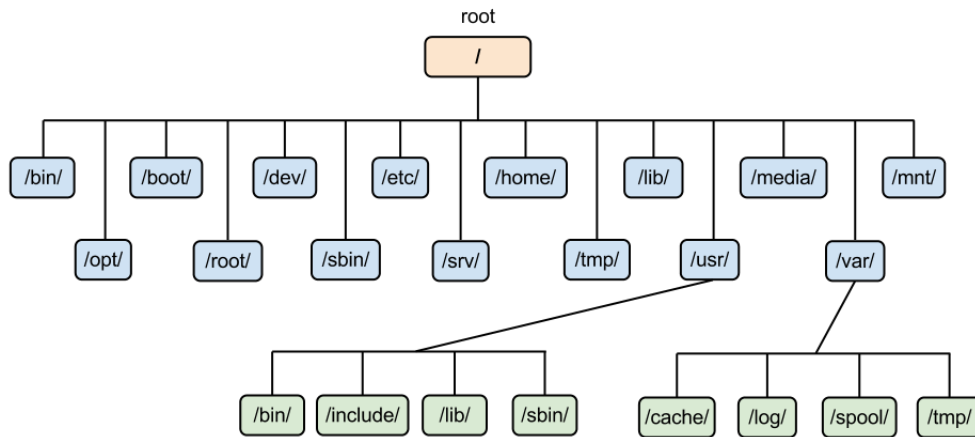


Figure 3.1: Example of a *nix filesystem structure

This is exactly the concept of having a global view assumption, we may not see it at first but operating systems are designed around it, this is why on such operating systems the same identifiers points to the same resources, thus all the processes share the same IP, the same filesystem, the same network stack and so on.

This assumption (always true in POSIX standards) does have some drawbacks, as a matter of fact it draws a tangible and inflexible line between what is allowed to do for users and what is not, most of the times this is understandable but in certain cases it is rather confusing.

Let's consider the *mount* operation, this system call modifies the global filesystem structure therefore it obviously is a restricted operation, if any user could mount a filesystem without any authorization the system would be highly unstable and unsafe. However a user willing to browse, open and edit its own files maybe from a usb-stick cannot do so because mounting a filesystem requires administrator permissions but doing the same action in a VM is possible and it does not require any additional permission, this is a bit

confusing and unjust to be honest, nevertheless is one of the limits imposed by the global view assumption.

3.2 View-OS

View-OS is the idea of having all the processes in the system free from this assumption[9]. The concept is that each process should be allowed to have its own view of the running environment, this does not mean that processes must live in completely different environments, but it can be useful to keep in their view a subset of the of the real system, while part of it is virtual and different for every process.

In particular, a View-OS process can define a new behavior for each system call, in such a way it is possible to run existing executables in multiple scenarios, possibly enhancing their features.

A View-OS process behavior entirely depends on the definition of the system calls it is using, but the same system calls can be re-defined and changed while an executable is running. This makes the environment in which processes run dynamic, flexible and extremely configurable, thus every process will have its own view of the outside world tailored for him.

Finally, since View-OS is part of the Virtual Square Framework, it shares all the V² design guidelines like modularity and user-mode implementation[5]. The concept of View-OS is strictly based on the idea of VM, and should be seen as a configurable, modular and general purpose Process-VM.

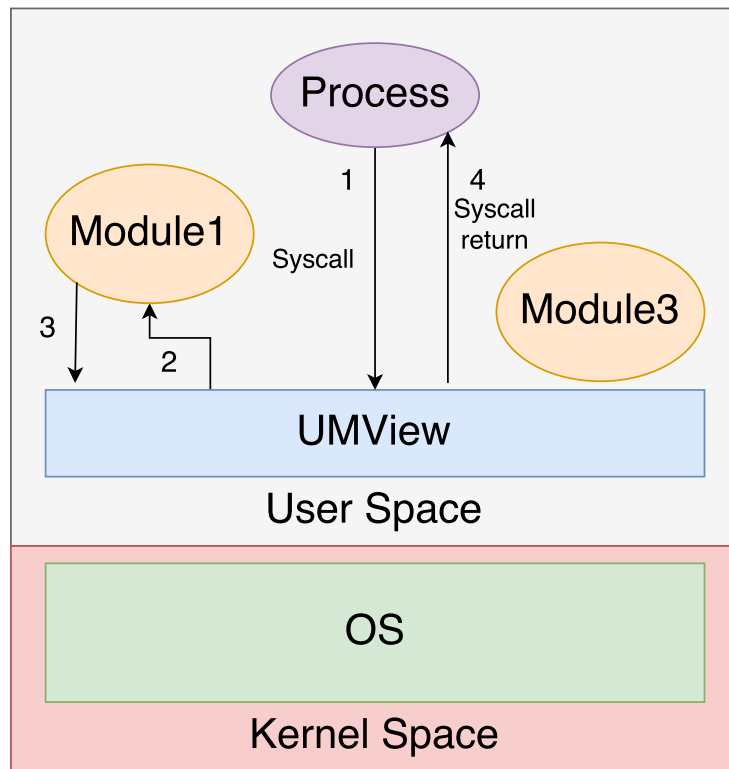


Figure 3.2: System call example

- (1) UMView captures the syscall, (2) the syscall is forwarded to a module,
- (3) The module processes the syscall , (4) UMView injects the syscall return value.

3.3 UMView

UMView lives as a part of the Virtual Square project[4] and specifically is a prototype implementation of the View-OS idea[5].

UMView is a user-level hypervisor capable of intercepting system calls and modifying their behavior according to the calling process view. Moreover UMView is implemented as a system call VM capable of loading modules on-the-fly that can change the view of running processes.

A system call implementation of View-OS is obviously slower than a Kernel one but bring some nice benefits with it:

1. a user-level hypervisor does not need any administration permissions;
2. it does not need any specific feature or support from the underlying Kernel;
3. debugging a userspace application is a lot simpler than debugging a Kernel module;
4. no lines are added to the Kernel code;

3.3.1 UMView Features

UMView brings a new set of interesting features to help developers and users achieve the flexibility they need, the following is a brief description of what UMView is able to do.

- **Syscall¹ rerouting:** UMView can reroute any syscall to a new syscall defined by UMView modules, redefining system calls behavior is an effective way to alter the view a process has of the entire system.
- **User-level implementation:** UMView is a userspace application, no additional permissions are required;
- **Stability:** Since UMView runs only unprivileged code and each process only alter its own view leaving the ones belonging to other processes untouched, stability is not affected.
- **Multi-arch support:** From the beginning UMView has been designed to support various architecture, and porting new ones requires few lines of code.

¹System call

- **Binary compatibility:** There's no need to recompile any application in order to use UMView.
- **Modularity:** UMView relies on modules to redefine system calls behavior, this mean flexibility and plenty of possibilities for developers, anything can implemented, from simple modules implementing a new virtual filesystem to modules aspiring to be a "userspace Kernel".

UMView presents itself as a one-stop solution to every virtualization application, nearly everything can be implemented as a UMView module.

3.3.2 Partial Virtualization

UMView is a View-OS implementation as a system call virtual machine, specifically it is a partial virtual machine (PVM)[5].

A PVM is a system call virtual machine providing the same set of system calls of the hosting kernel.

A PVM allows to:

- combine several PVMs together applying one VM on top of the other;
- forward each system call to the hosting Kernel or to another underlying partial VM;
- load modules inside the VM which redefine system calls under certain conditions;

UMView is developed using the *ptrace()* system call to intercept system calls issued by an application. At the time being this seems to be the most reliable way to perform such a task without requiring additional functionalities from the Kernel or without adding a Kernel module, although *ptrace()* has not been designed for the implementation of partial virtual machines but specifically to support GDB².

²GNU Debugger

Chapter 4

UMView for Users

This actually is a new incarnation of the UMView project, the first working implementation was made by the Virtual Square team led by Prof. Renzo Davoli thus this chapter will only introduce some basic example of UMView modules which have been written for testing and development reasons.

4.1 Starting UMView

Spawning a process wrapped in a UMView session is as simple as launching a shell, as a matter of fact spawning a shell inside the UMView hypervisor is what we are going to do (Figure 4.1):

```
$ umview xterm
```

4.2 Working with Modules

Modules are what make UMView such a flexible software, they can redefine the behavior of any system call supported by UMView. The basic principle is, whenever a process expresses a request to the Kernel through a system call this request usually targets some resource identified

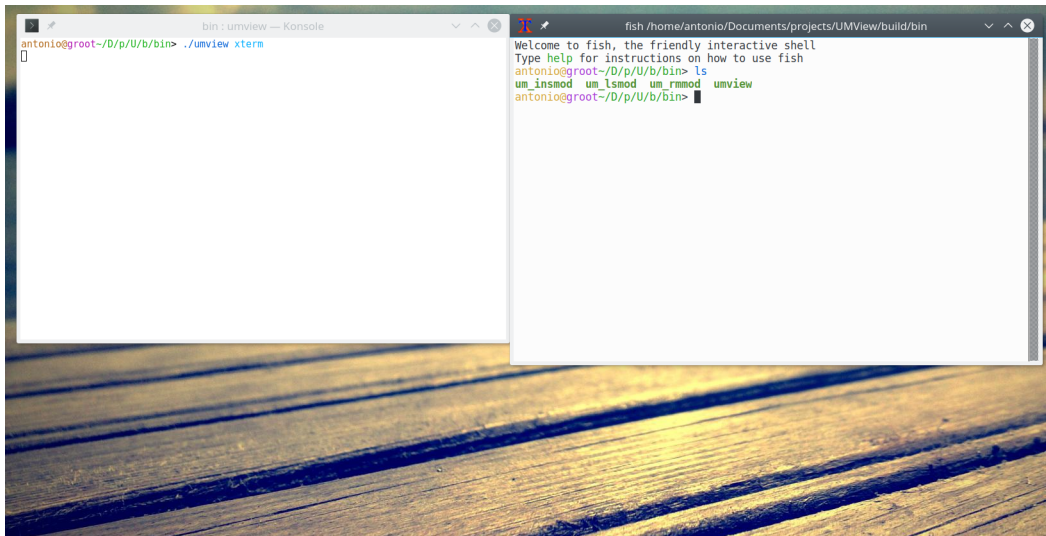


Figure 4.1: xterm inside UMView

by an ID, this can be a file descriptor, a path or a network interface, if a module owning the resource associated to the ID is currently loaded that same module is selected to fulfill the process request.

4.2.1 Listing Modules

UMView has got some special commands to interact with modules, one of these is used to list the currently loaded modules (Figure 4.2):

```
$ um_lsmod
```

The command is nearly identical to the `lsmod` Linux command which shows the currently loaded Kernel modules, this has been done to maintain consistency.

A terminal window titled 'fish /home/antonio/Documents/projects/UMView/build/bin'. The prompt is 'antonio@groot~/D/p/U/b/bin>'. The user enters 'ls', and the output is 'um_insmod um_lsmod um_rmmod umview'. The user then enters 'um_lsmod', and the output is 'No modules are currently loaded'. The prompt returns to 'antonio@groot~/D/p/U/b/bin>'.

```
antonio@groot~/D/p/U/b/bin> ls
um_insmod um_lsmod um_rmmod umview
antonio@groot~/D/p/U/b/bin> um_lsmod
No modules are currently loaded
antonio@groot~/D/p/U/b/bin> █
```

Figure 4.2: `um_lsmod` command

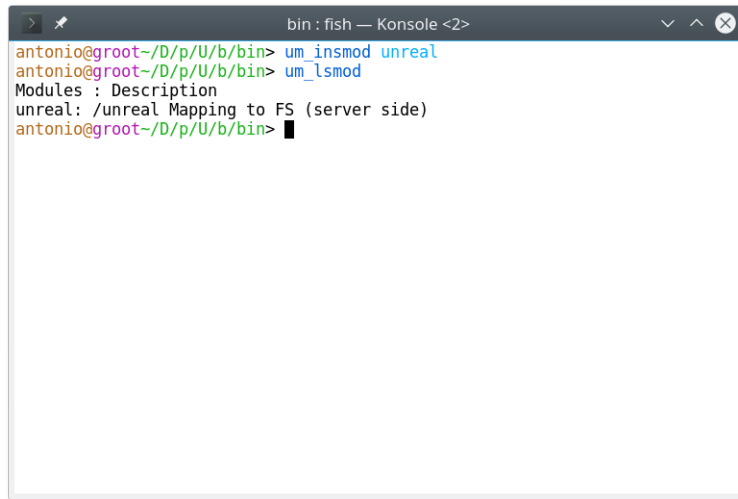
At the time being no modules are loaded but we will get to that shortly.

4.2.2 Loading Modules

Loading a module is supported by launching the command `um_insmod`. The user does not have to type the file extension or the position of the module unless this is located in a directory which is not part of the UMView modules path.

At the time being these are the directories in which UMView expects to find modules ready to be loaded:

1. `~/umview/modules`;
2. `/usr/lib/umview/modules`;
3. `${INTREE_MODULES_PATH}/lib` which is the the directory where output files are put when compiling from source;

A terminal window titled "bin : fish — Konsole <2>" showing the execution of the `um_insmod unreal` command. The prompt is `antonio@groot~/D/p/U/b/bin>`. The command is entered and executed, resulting in the output: `Modules : Description` and `unreal: /unreal Mapping to FS (server side)`. The prompt returns to `antonio@groot~/D/p/U/b/bin>` with a cursor.

```
antonio@groot~/D/p/U/b/bin> um_insmod unreal
antonio@groot~/D/p/U/b/bin> um_lsmod
Modules : Description
unreal: /unreal Mapping to FS (server side)
antonio@groot~/D/p/U/b/bin> █
```

Figure 4.3: `um_insmod` command

For example we can load the `unreal` module launching the following command (Figure 4.3):

```
$ um_insmod unreal
```

Now as it can be seen from figure 4.3 we have loaded the module `unreal`. UMView loads a given module only once for a given tree of processes.

4.3 The Unreal Module

The `unreal` module when loaded creates a virtual filesystem mounted at the root directory which mirrors the content of the filesystem structure. It basically redirects any system call to the mirror counterpart, figure 4.4 shows the idea behind this module.

After loading the module any operation done on a real path on which we prepend `/unreal` will be forwarded to the real path giving the illusion of a real filesystem mounted at `/unreal`.

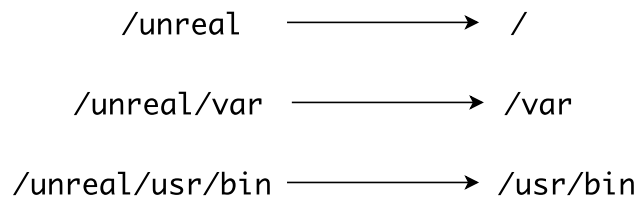


Figure 4.4: The unreal module

We can try some common unix commands to test the module.

```
$ ls /unreal
bin dev home lib64 opt root sbin sys usr
boot etc lib mnt proc run srv tmp var
$ echo "unreal" > /unreal/tmp/file
$ cat /unreal/tmp/file
unreal
```

The Linux Kernel knows nothing about a virtual filesystem called *unreal* and if a process tries to do the same operation outside of the UMView hypervisor it will result in a "No such file or directory".

The reason is at this point obvious, UMView only alters the view of the wrapped process which has loaded the `unreal` module.

4.4 A Useless Module

The second module we are going to take under consideration is the `useless` module.

This module mimics the behavior of a simple Kernel device driver implementing a char device which stores into an internal buffer some content.

First we have to load the module:

```
$ um_insmod useless
```

Now that the module is loaded we can put a store a string into its internal buffer:

```
$ echo "Hello UMLView!" > /dev/useless
```

Let's use a minimal C++ program to print out the content of /useless:

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
    string buf;
    ifstream file("/dev/useless");
    getline(file, buf);
    cout << buf << endl;
    return 0;
}
```

Once executed it will correctly print out the content we have just put into the *useless* module:

```
$ ./simple_read Hello UMLView!
```

Again we've seen a simple virtualization associated to a path resource in the filesystem structure only visible from a process wrapped inside a `UMView` session.

This is exactly the same as having a Kernel module implementing a char device exposing a `/dev/useless` resource which is one of the first examples many try when writing their first Kernel module, but the just shown module being a `UMView` one runs in userspace and needs no additional permission to be loaded.

4.4.1 Removing Modules

Removing a module is supported by launching the command `um_rmmod`. The only argument of this command is the module which has to be removed, in this case since we previously loaded two module we are going to remove them both (Figure 4.5).

```
$ um_rmmod unreal
$ um_rmmod useless
```



```
bin : fish — Konsole <2>
antonio@groot~/D/p/U/b/bin> um_lsmod
Modules : Description
unreal: /unreal Mapping to FS (server side)
useless: This module does nothing
antonio@groot~/D/p/U/b/bin> um_rmmod unreal
antonio@groot~/D/p/U/b/bin> um_rmmod useless
antonio@groot~/D/p/U/b/bin> um_lsmod
No modules are currently loaded
antonio@groot~/D/p/U/b/bin> █
```

Figure 4.5: um_rmmod command

Chapter 5

UMView for Developers

This chapter will cover all the material needed to start developing modules for UMView.

A substantial part of this dissertation has been dedicated to design a comfortable environment for developers willing to develop UMView services and to constructing an easy and intuitive set of APIs.

5.1 Modules API

In order to start developing a module one must import the definitions of the facilities UMView offers, in this case there is just a single header file called `umview/header.h`.

5.1.1 Declaring a Service

Every UMView module is described by a data structure called `umview_service_t`.

```
typedef struct umview_service_t {
    char *name;
    char *description;
    void *dlhandle;
    void (*destructor)();
    sysfun *module_syscalls;
} umview_service_t;
```

The following is a description of the fields:

- **name**: the name of the module;
- **description**: a brief description of what the module does;
- **dlhandle**: used internally;
- **destructor**: a handle to the destructor function, used internally;
- **module_syscalls**: the system calls table containing the redefinitions of the module. This table is used internally by UMView and is dynamically allocated by `umview_add_service` and freed by `umview_del_service`;

To actually create a module two simple steps are required, the first one is to declare a `umview_service_t` static variable.

The variable can have any name as long as the `UMVIEW_SERVICE()` macro is used to state that the variable will indeed contain a UMView module, without it the module will have no visibility when trying to loading it into a UMView session and will therefore fail to load.

```
#include <umview/module.h>
// Declaring a UMView service
static umview_service_t service;
UMVIEW_SERVICE(service);
```

To finally register our module we will have to call the `umview_add_service()` function inside the module's constructor.

```
umview_add_service(service, "Hello-module", "My first UMView module");
```

The registration function must be called before any other UMView API, otherwise the module's data structure used to contain the system calls redefinitions will not be allocated and ready to be used.

5.1.2 Using the Hashtable

UMView internally utilizes a global hashtable used for storing any kind of object, this data structure can also be used by modules.

It mainly serves one purpose from a module perspective, it can be used to create a resource owned by a module. Whenever an application requests to do some processing using that resource the module can take responsibility for the system call to be executed.

```
hashtable_obj_t*  
ht_tab_pathadd(uint8_t type, const char *source,  
               const char *path, const char  
               *fstype, unsigned long mountflags,  
               const char *mountopts, umview_service_t  
               *service, unsigned char trailingnumbers,  
               confirmfun_t confirmfun, void* private_data);
```

The following is a brief description of the function's parameters:

- **type**: this must always be `CHECKPATH`¹ and indicates we are adding a virtual path to the filesystem structure;
- **source**: where this new path will be mounted at;
- **fstype**: name of the filesystem type;
- **mountflags**: mount options expressed as a bitmask, the same used by the Linux Kernel;
- **mountopts**: mount options expressed as a comma separated list of values, the same used by the Linux Kernel;
- **service**: the UMView module associated with this resource;
- **trailingnumbers**: if 1 then the path is allowed to have trailing numbers, otherwise set to 0;
- **confirmfun**: now unused;
- **private_data**: an address the module can store to associate it with the path resource being added, it can be anything;

This function returns the newly created hashtable element.

For example we can use UMView's hashtable to add a virtual device *hellodev* using the following call:

```
//registering a path and storing the new hashtable element
hashatable_object_t hte;
hte = ht_tab_pathadd(CHECKPATH, "/", "/dev/hello", "hellodev", 0,
                    "", &service, 0, NULL, NULL);
```

¹In future releases of UMView more options will be available

From now on if some application does a system call X on `/dev/hello` and the module described by the variable `service` redefines X with a new implementation X_1 then every system call X will be forwarded to the module which in turn will execute X_1 .

Obviously what has just been described applies only for the processes living inside the UMView hypervisor that loaded the `hellodev` module.

Once a module is unloaded it should free any previously allocated resource, in this case the `hellodev` virtual device should remove the hashtable element created in the constructor.

The place to do this is of course the module's destructor.

```
//in the destructor  
ht_tab_del(hte);
```

5.1.3 Creating Virtual File Descriptors

UMView modules gives the possibility of creating and managing a set of virtual file descriptors, this can be useful in many situation mostly to associate an ID to some resource the module is offering.

Every virtual file descriptor is just like a real one, specifically it is just an integer representing some resource, in addition to this UMView offers the possibility of associating some data (through a pointer) with every VFD².

The creation of VFDs is completely controlled by the hypervisor, this is done to avoid collisions with real file descriptors, therefore a function call has been created for modules to request a new VFD.

²Virtual file descriptor

```
int umview_create_fd(int close_on_exec, void *private);
```

- **close_on_exec**: if 1 then this VFD will be closed if the process executes an `execve()` call;
- **private**: a pointer used to associate data to the VFD.

Of course the return value is the value of the newly created VFD.

UMView also offers a pair of getter/setter functions to manage the VFD's associated data, their usage is straightforward.

```
void *umview_fd_set_private(int fd, void *private);  
void *umview_fd_get_private(int fd);
```

They both return the value of the `private` variable associated to the VFD.

5.2 UMView System Calls

This section is going to describe how modules can redefine system calls and the facilities offered by UMView about the topic.

5.2.1 Redefining System Calls

For a module to redefine a system call the first step is to give a new implementation of it, this is fairly simple. It is sufficient to copy the signature of the system call and develop a new implementation.

The following example redefines the behavior of the `write` system call, instead of actually writing the data contained in the buffer passed to the system call it will just store it in a module's buffer.

```
static char module_buf[4096];

static ssize_t new_write(int fd, const void *buf, size_t count)
{
    memcpy(module_buf, buf, count);
    return 0;
}
```

5.2.2 Registering System Calls

Now that the new system call is ready to be used UMLView must be informed about its existence otherwise it will not be able to perform the syscall redirection towards the module.

To register a new system call the following macro is available:

```
module_syscall(service, syscall_name, new_syscall);
```

This macro must be called in the module's constructor.

Following our example the call to register the new `write` system call will be:

```
//inside the constructor
module_syscall(service, write, new_write);
```

This is enough to register the new system call.

5.2.3 System Calls Grouping

The Linux system calls for the x86_64 architecture are more than 300 and a relevant number achieve the same results with minor differences in their signatures.

To avoid overwhelming developers with many system call implementations

basically doing the same thing UMView internally classifies system calls in several sets based on the syscalls behavior.

In such a way developers only need to implement the UMView system call for that set and the whole set will be covered. For example Linux has got different calls to retrieve information about a file: `stat`, `fstat`, `lstat` and `newfstatat`.

They all have the same behavior but accept different arguments and in some cases different option flags. This group of system calls is the *stat* set and the UMView system call representing this set is the `lstat` call which makes it the only system call a module needs to implement to support the whole set.

Since there could be different forms of IDs pointing to the same resource (path/file descriptors) when UMView receives a system call according to the provided arguments it internally and uniquely identifies the resource the system call is referring to.

Once this resolution process is completed UMView rearranges the system call parameters in order to always pass the module an absolute path as the resource identifier or a file descriptor when the same resource is non representable by a path.

In the *stat* system call family case all the system calls are translated into a `lstat` call with the path provided as an absolute one (Figure 5.1). The same mechanism applies for any option flags.

This is utterly important because it simplifies the module development process removing duplicate code which could bloat modules/textquoteright source code.

Some system calls can even be ignored by module developers, if the be-

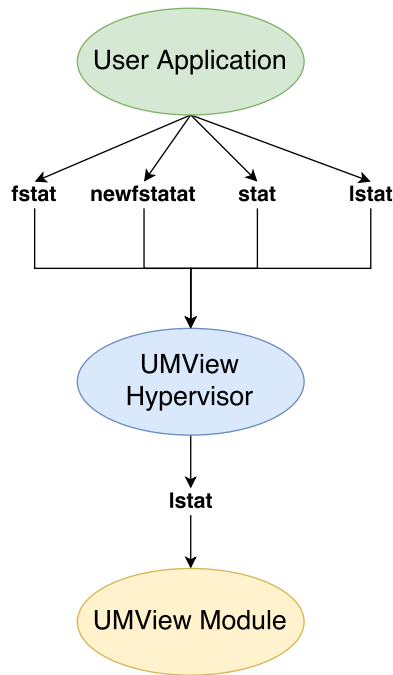


Figure 5.1: System call grouping

havior is the one expected UMView will deal with them without requiring the module intervention.

For example if a module redefines the `open` system call but not the `close` one UMView will automatically close the file descriptor created by the first system call, these are called UMView optional system calls and they can be ignored if they are to behave like the real system call (this works even with VFDs). At the time being the only optional system calls are `close` and `dup3`.

The table 5.1 shows the currently supported system calls and how they are grouped in different sets.

5.3 Writing Modules

After a quick introduction to the UMView modules API it's helpful to show a minimal module in order to wrap up the concepts covered so far.

Table 5.1: UMView supported system calls

System Calls	UMView Sytem Call	Resource ID
open, creat, openat	open	path
write, writev	write	file descriptor
read, readv	read	file descriptor
pread64	pread64	file descriptor
pwrite64	pwrite64	file descriptor
close	close	file descriptor
chdir, fchdir	not forwarded to modules	-
lstat, stat, newfstatat	lstat	path
access, faccessat	access	path
readlink, readlinkat	readlink	path
lseek	lseek	fd
utimensat, futimesat, utimes, utime	utimensat	path
dup3, dup2, dup	dup3	file descriptor
symlink, symlinkat	symlink	path
link, linkat	link	path
unlink, unlinkat	unlink	path
mkdir, mkdirat	mkdir	path
rmdir	rmdir	path
getcwd	not forwarded to modules	-
getdents64, getdents	getdents64	file descriptor
chmod, fchmod, fchmodat	chmod	path
lchown, fchown, chown, fchownat	lchown	path

5.3.1 Hello UMView

First we have to include the header file containing all the definitions needed to develop a UMView module.

Then we are going to implement a read system call in order to create a VFS³ that when any of its node is read the string "Hello UMView!" is returned.

```
#include <string.h>
#include <umview/module.h>

static ssize_t hello_read(int fd, void *buf, size_t count)
{
    strncpy(buf, "Hello UMView!", count);
}
```

5.3.2 Module Constructor

Now that we have our read system call the module's constructor must be written. The constructor is the first function that gets called as soon as the module is loaded and is therefore where all the initialization routines and operation should be put.

The constructor has a special signature, as matter of fact it is a function having the special macro **UM_MOD_INIT** preceding its name like shown in the example below.

Inside the constructor we have to:

1. add a hashtable path element to create the virtual path `/hello`;
2. register the *hello* module;
3. register the new `read` system call;

³Virtual filesystem

```
static umview_service_t hello_service;
static hashtable_obj_t *hte;

UM_MOD_INIT hello_init()
{
    //register the "hello" module
    umview_add_service(hello_service, "hello", "The greeting module");
    //create a hashtable element for the "/hello" path
    hte = ht_tab_pathadd(CHECKPATH, "/", "/hello", "hellofs", 0,
                        "", &hello_service, 0, NULL, NULL);
    //register the new read system call
    module_syscall(hello_service, read, useless_read);
}
```

5.3.3 Module Destructor

The only thing missing to have a complete minimal UMView module is the module's destructor.

The destructor is where all the allocated memory should be freed and all the hashtable elements should be deleted.

For the newly created *hello* module writing the destructor is quite a trivial task, like the constructor the destructor is a special function having the macro **UM_MOD_FINI** preceding its name.

The destructor is obviously the last function that is going to be called before unloading the module.

```
UM_MOD_FINI hello_fini()
{
    ht_tab_del(hte);
    umview_del_service(hello_service);
}
```

The last line of every UMView module's destructor should be the macro `umview_del_service` having the `umview_service_t` variable describing the module as the only argument.

The purpose of this call is to free the memory used for all the internal data structures related to this module and is therefore very important otherwise memory leaks may occur.

5.3.4 Building Modules

Building the module shown in the previous sections is fairly simple thanks to the CMake build system.

After cloning the git repository the module source code should be placed in the `src/modules` directory because CMake has been instructed to detect and compile every module placed in that directory.

A clean and easy way to build projects with CMake is to create a separate directory in which all output files will be put and this is exactly what we are going to do.

The shell commands shown below create a new directory and tell CMake to build the project there, after that the command to compile the hello module can be launched.

```
$ mkdir build
$ cd build
$ cmake ..
$ make hello
```

The module has been compiled and put into the `build/lib` directory. If compiling UMView from source the UMView binary will be put in `build/bin` and it will automatically find modules in the `build/lib` directory.

5.3.5 Testing and Debugging

Modules can be debugged with GDB or any other debugger the user may prefer, being UMView a userspace application modules and UMView can be debugged together, this is also useful for educational reasons as to better understand the inner workings of UMView.

The suggested way to test and debug a module is to set a breakpoint to the module's constructor or any other interesting module's function or system call.

UMView also offers a set of macros to facilitate logging to the system log:

```
um_syslog(prio, ...);
um_log(...);
um_log_notice(...);
um_log_err(...);
um_log_critical(...);
um_log_debug(...);
um_log_warning(...);
```

To have them available the header file `umview/syslog.h` must be included.

To wrap everything up the components a minimal UMView module should have are:

- a constructor;
- the module's syscall redefinitions;
- a destructor;

Chapter 6

UMView Internals

Throughout this chapter all the inner workings of UMView will be explored and discussed in order to give a detailed picture of all the software developed during this dissertation including its architecture.

6.1 The Tracing Unit

UMView being an implementation of View-OS and in particular a partial VM lays its foundation on having a mechanism to intercept and manipulate the system calls issued by an application.

The mechanism used for UMView, which is developed on and for Linux, is the only one available on Linux which is the `ptrace` system call.

6.1.1 Ptrace

The `ptrace` system call is described as follows by the *Linux Programmer's Manual*:

The `ptrace()` system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's

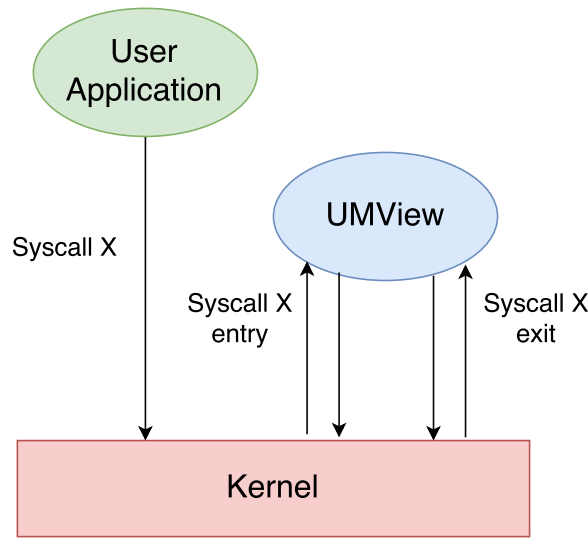


Figure 6.1: ptrace tracing

memory and registers. It is primarily used to implement breakpoint debugging and system call tracing[7].

The Manual is quite clear about the intended usage for this system call, as a matter of fact it has been designed for applications such as `gdb` and `strace` and not for system call manipulation/injection or even virtualization. Nevertheless both the original UMView implementation and this porting use this system call as a backend to intercept and manipulate system calls hoping that some specific and more performant technique will be available in the future.

Despite its use in this project the `ptrace` system call has some performance drawbacks and at times is quite cumbersome to use. The performance penalty comes from the fact that the only available option to track a process's system call is the `PTRACE_SYSCALL` which stops the tracee at the next entry to or exit from a system call.

This means that for every system call two `ptrace` calls are needed even when there's no interest in inspecting the next entry/exit transition or the entire next system call.

For example from the UMView perspective if the system call X is intercepted and then virtualized there's no interest in stopping again at the exit transition from this system call because the call has already been processed, but a second `ptrace` call (with the `PTRACE_SYSCALL` option) is needed because there is no other way to tell the Kernel we are interested in the next system call without stopping at every entry/exit transition.

Moreover UMView needs to stop the tracee at every system call to be able to virtualize the ones it is interested to because the Linux Kernel does not offer any facility to choose only a set of system call to trace.

Nevertheless the system call interception mechanism in UMView is implemented by the tracing unit using `ptrace`.

6.1.2 Intercepting System Calls

The tracing unit implementation resides in the source file `src/umview/tracer.c` and its API is composed of the sole `umview_trace` function.

```
void umview_trace(int tracee_tid);
```

Its only argument is the `tid`¹ of the thread the calling process is going to wrap inside a UMView session.

The `umview_trace` function has two main tasks, the first one is of course to trace the system calls issued by a thread/process and the second one is to create new tracer-threads each time a tracee executes a `clone` system call.

The tracing unit basically is an event-driven routine that according to the type of event performs a set of actions. As stated before the tracing unit uses the `ptrace` backend and obviously receives the events detected by the `ptrace` support. The main event in which UMView is interested in is of course the reception of system calls.

¹Linux thread ID

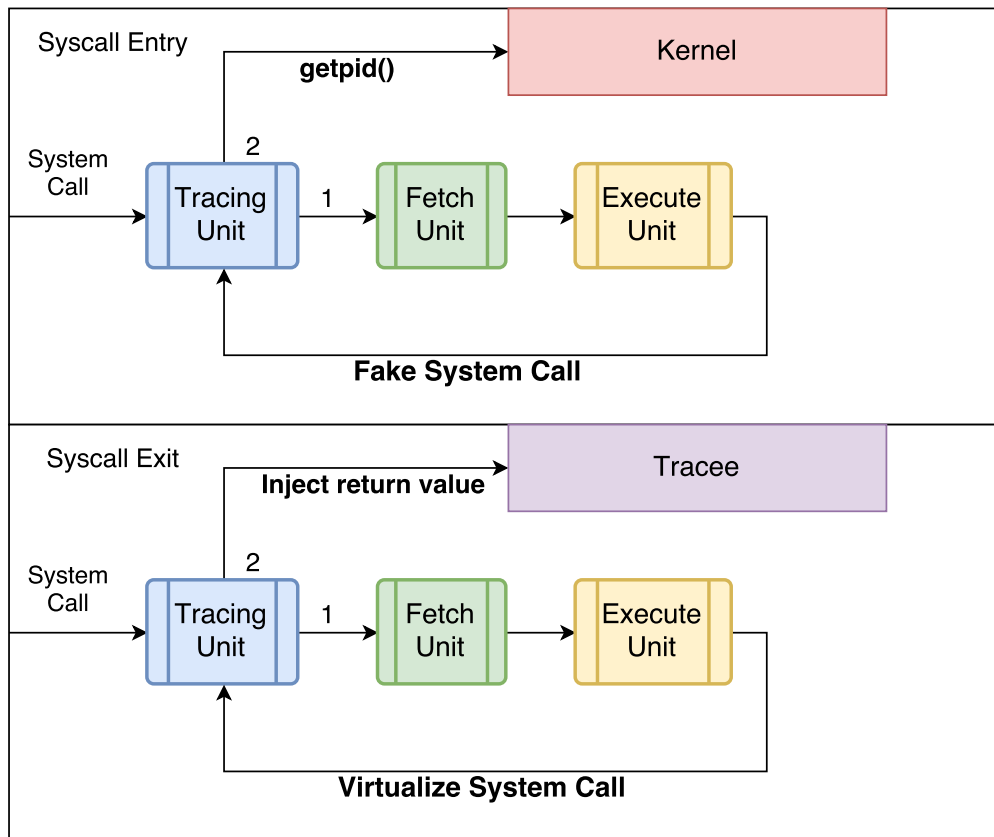


Figure 6.2: Virtualizing a system call

After the tracing unit intercepts a system call the values of the machine registers are saved and passed on to the fetch unit and consequently to the execute unit. Once the execute unit has completed its processing, the intercepted system call will be either faked or virtualized according to the output of the two previous units.

When a system call is faked what actually happens is a manipulation of the system call in order to replace it with a `getpid()`, in such a way the Kernel will never receive the original intercepted system call. This usually happens during a system call *entry* to transition to make the Kernel skip the call.

When a system call is virtualized a return value is injected into the tracee registers to make it look like the real return value of the just issued system call. This usually happens during a system call *exit from* transition.

6.2 Virtual System Calls

UMView enlarges the set of Linux system calls creating some virtual ones in order to be able to communicate with UMView-aware applications.

6.2.1 The UMLib

At the time being the new virtual system calls are just three:

1. `int umview_add_service(const char *path, int permanent);`
2. `int umview_del_service(const char *name);`
3. `int umview_list_service(char *buf, int len);`

The first one can be used to load a module, the second one to unload it and the third one returns a buffer filled with a description of all the currently loaded modules, if `buf` is `NULL` then the call returns the size of the buffer that should be allocated to store the description string.

These functions are part of a library called **UMLib** which is compiled as a shared object, thus the applications using this functions should be linked against the UMLib library adding `-lumview` to the GCC command.

The commands `um_insmod`, `um_rmmod` and `um_lsmod` use this library to load, unload and remove modules.

Obviously these virtual system calls are only available inside a UMView session.

6.2.2 Implementation

Regarding the implementation of the UMView virtual system calls the technique

used is quite simple, since Linux system calls numbering starts from 0 the UMView's one start from -1 instead, in such a way no collision is possible between real or virtual system calls.

- `umview_add_service` → System call -1;
- `umview_del_service` → System call -2;
- `umview_list_service` → System call -3;

6.3 The Plugin Architecture

In order for UMView to be able to load modules a plugin architecture was needed. A simple and straightforward mechanism existed in the original implementation of UMView thus it has been ported with minor modifications introduced to improve code clarity and modularity.

6.3.1 Modules as Shared Objects

UMView modules are compiled as dynamic shared objects and therefore have a `.so` extension.

The functions used to load/unload modules reside in the `modutils.c` source file, their signature are the following:

```
1 umview_service_t *open_module(const char *modname); void
2 close_module(umview_service_t *module);
```

The first one loads a module and the second one unloads it. Both functions internally use the `dl` library in order to load dynamic shared

objects and obtain addresses of symbols contained into libraries.

The `open_module` function in particular tries to load the target module from different locations, once found it loads it into memory and adds a reference to it in the global hashtable.

The module's constructor is automatically called when the module is loaded thanks to the GCC constructor attribute `__attribute__((constructor))` contained in the `UM_MOD_INIT` macro.

The `close_module` function instead unloads the module from memory, removes its reference from the global hashtable and call the module's destructor.

6.4 The Guardian Angels Technique

UMView is designed to take maximum advantage of the parallelism and multi-threading characteristics of the machine on which in running on.

Differently from the previous implementation of UMView which was single-threaded the new one follows a different tracing model, as a matter of fact UMView has got a 1-1 relationship between tracers and tracees, this means that for every process or thread living under the supervision of UMView there is a tracer instance guarding its own tracee. This is what gave the name to the parallelization technique called *The Guardian Angels*. The first proof-of-concept of this technique has been made by Giacomo Bergami, Gianluca Iselli and Matteo Martelli.

This technique basically consists in associating a new tracer to a tracee every time the tracee spawns a new child via a `clone` system call. The result is both performant and simpler to manage because each tracer has to worry about one and only one tracee at a time.

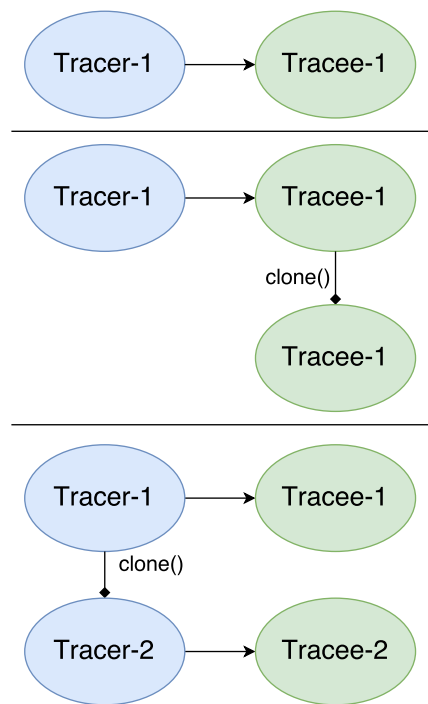


Figure 6.3: UMView guardian angels

6.4.1 Implementation

The implementation is based on the system call tracing backend which is `ptrace`.

When a tracee performs a `clone` as soon as the new child does its first system call UMView executes the `transfer_tracee` routine, this function blocks the newly created thread/process by changing the just issued system call to a `poll(NULL, 0, -1)` which obviously stops the calling process indefinitely.

While the child tracee is blocked the tracer copies or share the parent tracee filesystem information and file descriptor table according to the `CLONE_FS` and `CLONE_FILES` flags presence in the `clone` system call. The tracer now creates a new thread which is going to become the child tracee's tracer.

Once this process is complete the new tracer unblocks the child tracee and starts tracing its system calls. The child tracee's first system call is recovered before the unblock.

6.5 The Hashtable Unit

UMView inherited from its previous implementation the global hashtable. The source code implementing this data structure has been left mostly untouched and for this reason is not going to be described in depth being a non-original part of this dissertation.

The global hashtable is shared between the UMView tracer-threads and is designed to store any kind of object.

Specifically each element of the hashtable is structured as follows:

```
typedef struct hashtable_obj_t {
    void *obj;
    char *mtabline;
    unsigned long mountflags;
    epoch_t timestamp;
    uint8_t type;
    uint8_t trailingnumbers;
    uint8_t invalid;
    umview_service_t *service;
    struct hashtable_obj_t *service_hte;
    void *private_data;
    int objlen;
    long hashsum;
    int count;
    /* confirmfun_t */
}
```



```
int (*confirmfun)(int type, void *arg, int arglen,
                  struct hashtable_obj_t *ht);
struct hashtable_obj_t *prev, *next, **pprevhash, *nexthash;
} hashtable_obj_t;
```

Here is a description of the various fields:

- **obj**: the address of the object stored in this hashtable element;
- **mtabline**: the mount tab line;
- **mountflags**: mount options;
- **timestamp**: element timestamp (discussed in the *Epoch* section)
- **type**: the hashtable element type;
- **trailingnumbers**: if this is a path element and is allowed to have trailing numbers;
- **invalid**: indicate an invalid element;
- **service**: address of the related UMView module;
- **service_hte**: address of the UMView module's hashtable element;
- **private_data**: data associated to this hashtable element;
- **objlen**: length of the object stored;
- **hashsum**: the hash value;
- **count**: reference counter;
- **confirmfun**: not used;
- **prev**, **next**, **pprevhash**, **nexthash**: elements for hashtable navigation and chaining;

The used hash function is the djb2 which is adequately simple and fast.

The hashtable, as pointed out before, is capable of storing any kind of object but despite the presence of the *type* field at the time being the only types of object actually stored are the CHECKPATH type (used for paths) and the CHECKMODULE type (used for storing loadable modules).

UMView can interact with the hashtable with an extensive set of APIs, even though the most used and important are actually four.

```
hashtable_obj_t*
ht_tab_add(uint8_t type, void *obj, int objlen,
           umview_service_t *service, confirmfun_t confirmfun,
           void *private_data);
```

The `ht_tab_add` is used, as the signature explicitly suggests, to add generic object to the hashtable.

```
hashtable_obj_t*
ht_tab_pathadd(uint8_t type, const char *source,
               const char *path, const char *fstype,
               unsigned long mountflags, const char *mountopts,
               umview_service_t *service,
               unsigned char trailingnumbers,
               confirmfun_t confirmfun, void *private_data);
```

The `ht_tab_pathadd` function as explained in chapter 5 is used for storing path elements in the hashtable.

```
int ht_tab_del(hashtable_obj_t *ht);
```

Another important function is the `ht_tab_del` one, used for removing an element from the global hashtable.

```
hashtable_obj_t*  
ht_check(int type, void *arg, umview_stat_t *st, int setepoch);
```

The last most used function of the hashtable set of APIs is the `ht_check` one, this routine searches the hashtable looking for the `arg` object of `type` type. If `setepoch` is 1 and an element is found the virtual epoch will be set to the hashtable's element one (epochs will be discussed in the relevant chapter).

The hashtable unit is of vital importance for UMView since its extensive usage as a way of communication and sharing memory between tracer-threads and modules.

6.6 Fetch-Execute Unit

UMView internal architecture has been inspired by the classic *fetch, decode & execute* model used by CPUs.

Even though the naming used throughout UMView's code essentially reflects the fetch, decode and execute operations only two of them actually exist as modules in the source code structure, the fetch and the execute module. The execute one absorbed the functionalities of the decode one (due to its simplicity).

6.6.1 The Fetch Unit

UMView's fetch unit is the one coming to play after a system call is intercepted. Its task is quite simple, retrieving the system call arguments from the machine's registers.

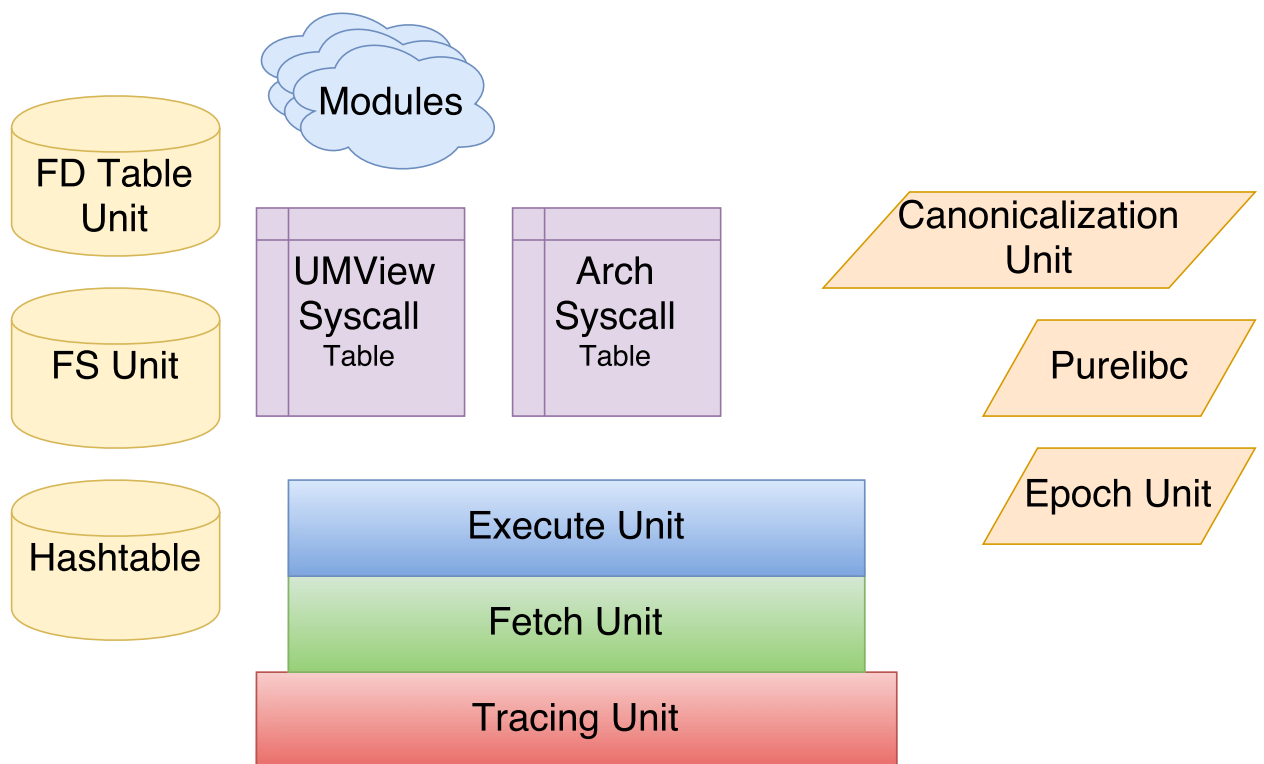


Figure 6.4: UMView Architecture

This is one of the few architecture-dependent components in UMView since its implementation depends on the representation of the machine's registers, thus to port UMView to a new architecture the `umview_syscall_fetch` routine must be implemented.

The function's signature is as follows:

```
void umview_syscall_fetch(struct user_regs_struct *regs,
                          syscall_descriptor_t *syscall_desc,
                          syscall_state_t sys_state)
```

A brief description of its arguments:

1. **regs**: a data structure describing the content of the machine's registers (input);
2. **syscall_desc**: the UMView data structure used to describe a system call (output);
3. **sys_state**: variable denoting if UMView is in a entry to or exit from system call state (input);

The `syscall_descriptor_t` struct is structured as follows:

```
typedef struct syscall_descriptor_t {
    int syscall_number;
    int fd_close_on_exec;
    syscall_arg_t syscall_args[SYSCALL_ARG_NR];
    syscall_arg_t ret_value;
} syscall_descriptor_t;
```

To port this function one must simply fill-in the `syscall_desc` fields using the information contained in the variable `regs`.

Since the tracer and tracee threads live in separate address spaces the fetch unit also implements a set of function allowing to transfer memory between the two address spaces, this is possible thanks to the special capability a Linux process has when p-tracing another.

```
char *umview_peek_string(char *addr);
```

This first function is used to retrieve a NULL-terminated string from the tracee memory at address `addr`.

```
void umview_peek_data(void *addr, void *buf, size_t datalen);  
void umview_poke_data(void *addr, void *buf, size_t datalen);
```

The `umview_poke_data` function writes *datalen* bytes of data from the buffer *buf* into the tracee memory starting at address *addr*.

The `umview_peek_data` function reads *datalen* bytes of data from the tracee memory starting at address *addr* storing them into the buffer *buf*.

All the fetch unit routines used to transfer memory employ the `process_vm_writev` and `process_vm_readv` system calls, which allow to do these operation in a single system call.

6.6.2 The Execute Unit

The execute unit takes control immediately after the fetch one, its task is to inspect the system call (decode unit) and to take appropriate actions.

This unit is implemented by the `umview_syscall_execute` function which after inspecting the number of the intercepted system call has two distinguished execution paths according to the *entry to/exit from* transition of the ptrace backend.

When the transition is an *entry to* UMView checks if the system call is a real one or a UMView's virtual one, in case of a virtual system call its implementation is called and the return value is stored. If the system call is indeed a real one UMView, according to the system call number, retrieves the addresses of two functions: the **wrapper** and **choice** function.

Choice functions

The choice function is used by UMView to check if the intercepted system call context is owned by a module, which means that the file descriptor or path the system call is referring to is actually owned by a currently loaded module. This happens when a system call is trying to fulfill any kind of operation on either a VFD, a real file descriptor or a virtual path added in the global hashtable by a module.

All the choice functions return a hashtable element, if this element is not *NULL* then indeed there is a loaded module waiting to process the intercepted system call, otherwise the call will be simply passed to the Kernel.

Choice functions are distinguished by their signature, which has to be as follows:

```
hashtable_obj_t>(*choice_function)(syscall_descriptor_t *);
```

Different system calls can share the same choice function but since these functions/textquoteright behavior mostly depend on the system call signature, specifically on the position of the arguments, the usual case is that system calls belonging to the same set (table 5.1) share the same choice function.

To better understand how a choice functions is implemented is useful to actually inspect one. The following CFunction² selects a hashtable element according to a path passed as the first argument of the system call.

²Choice function

```
hashtable_obj_t *choice_path_follow(syscall_descriptor_t *sd)
{
    char *path = umview_peek_string(SYSARG(char *, sd, 0));
    hashtable_obj_t *ht;

    ht = get_hte_from_path(&path, FOLLOW_LINK, 1);
    SET_SYSARG(sd, 0, path);
    return ht;
}
```

The `choice_path_follow` implementation is quite straightforward, after retrieving the path from the tracee memory searches the hashtable for a corresponding element, if found it returns that element.

CFunctions usually perform a fixed set of operation in order to simplify further processing of the system call.

As shown in the `choice_path_follow` example these operations usually are:

1. retrieving syscall arguments from the tracee memory and making them available for further processing;
2. relative path resolution in order to obtain an absolute path;
3. at-call (`newfstatat`, `linkat`, `unlinkat`, etc...) path resolution into an absolute path;

Wrapper functions

Wrapper functions are used by `UMView` to forward intercepted system calls to modules. They are called immediately after the system call choice function.

Like the CFunctions many system calls can use the same WFunction but

the usual case is that system calls belonging to the same set share the same WFunction.

WFunctions³ actually classify in two distinct categories, the WINFunctions⁴ and the WOUTFunctions⁵.

The WINFunctions are called in the case of a *entry to* system call transition and the WOUTFunctions when there is a *exit from* system call transition.

WINFunctions

WINFunctions are distinguished by their type, which has to be as follows:

```
void (*wrapin_func)(hashtable_obj_t *, syscall_descriptor_t *,
                   syscall_behaviour_t *);
```

The input arguments are a hashtable element (returned by the choice function), which tells UMView if this system call has to be processed by a module, a system call descriptor and a pointer to a variable of type `syscall_behaviour_t`.

```
typedef struct syscall_behaviour_t {
    long ret_value;
    syscall_action_t action;
} syscall_behaviour_t;
```

The `syscall_behaviour_t` structure contains the return value of a given system call and the action to be taken. The *action* variable is an enumerated type having as possible values VIRTUAL, STD or FAKE.

³Wrapper functions

⁴IN wrapper functions

⁵OUT wrapper functions

The WINFunction's task is to either forward the system call to a module or to the Kernel and if the former case applies to store the return value of the processed system call.

The following code shows the implementation of the *chmod* system call WIN-Function.

```
1 void wi_mkdir(hashtable_obj_t *ht, syscall_descriptor_t *sd,
2             syscall_behaviour_t *sysb)
3 {
4     char *path = NULL;
5     int mode = 0;
6     sysfun fun;
7
8     if (IS_SYS(sd, mkdir)) {
9         path = SYSARG(char *, sd, 0);
10        mode = SYSARG(int, sd, 1);
11    } else if (IS_SYS(sd, mkdirat)) {
12        path = SYSARG(char *, sd, 1);
13        mode = SYSARG(int, sd, 2);
14    }
15    sysb->action = STD;
16    if (ht && (fun = get_module_syscall(ht, SYSNO(mkdir)))) {
17        sysb->action = SKIP;
18        sysb->ret_value = fun(path, mode);
19    }
20    xfree(path);
21 }
```

WINFunctions essentially carry out the same operations for the many system calls supported by UMLView. First, according to the intercepted call, the system call's arguments are retrieved and then if the *ht* variable is not *NULL* and the module pointed by it has an implementation for the just received

system call that same implementation is called in order to process the system call.

In the `wi_mkdir` function at line 16 we can notice the translation of an eventual `mkdirat` system call into a `mkdir`. Another thing to be noticed is how the `sysb→action` variable is used, if the call has to be simply passed to the Kernel the value `STD` is stored into it, which denotes the normal execution flow for this system call, otherwise it takes the value `SKIP` which tells to the tracing unit to skip the system call.

`get_module_syscall` is a macro which actually retrieves the address of the module's implementation of the system call and at line 18 (`wi_mkdir`) we can notice the actual call and the return value being stored.

`IS_SYS(sd, syscall)` is a helper macro returning 1 if the intercepted system call described by the `sd` variable is a *syscall* system call;

WOUTFunctions

Once the *entry to* transition is concluded, which means that the WIN-Function has been executed and the tracing unit has taken appropriate actions, at the next *exit from* transition the system call's associated WOUT-Function is called.

WOUTFunctions/textquoteright type has to be as follows:

```
void (*wrapout_func)(syscall_descriptor_t *, syscall_behaviour_t *);
```

Their task is usually very trivial, it consists in inspecting the action previously taken by the tracing unit which can be `SKIP` if the system call has been forwarded to a module or `STD` if no virtualization took place.

When a system call has been taken care by a module the WOUTFunction simply stores the value `VIRTUAL` into the `sysb→action` variable in order to make the tracing unit inject a custom value as the system call return one.

If the call has not been virtualized the WOUTFunction does not take any action.

What has just been described is what happens in the majority of cases, therefore one WOUTFunction is shared between most of the system calls, this function is `wo_virtual`. When some custom action has to be taken a different WOUTFunction can be implemented and used.

6.7 Purelibc

UMView also supports virtualization of its own system calls and the ones generated by modules, this is done thanks to a library called **purelibc**. Purelibc is a C library developed by the Virtual Square team to provide a pure access to the system call interfacing functions of the C language.

UMView initializes the purelibc library before starting to trace the first process.

```
void init_purelibc()
{
    native_syscall = _pure_start(puresyscall_landing, NULL, 0);
}
```

The `_pure_start` function takes as its first input parameter the address of a function that will be called every time UMView or a loaded module issues a system call, the return value of this function is a hook to the `syscall` function allowing to directly call a real system call.

The virtualization offered by the purelibc library does not incur any performance penalty because the actual system calls are redirected to the `puresyscall_landing` function without any context switch.

```

static long int puresyscall_landing(long int sysno, ...)
{
    syscall_descriptor_t sd;
    syscall_tab_entry *tab_entry = NULL;
    hashtable_obj_t *ht;
    va_list arg_list;

    sd.syscall_number = sysno;
    va_start(arg_list, sysno);
    for (int i = 0; i < SYSCALL_ARG_NR; i++)
        sd.syscall_args[i] = va_arg(arg_list, syscall_arg_t);
    va_end(arg_list);

    /* check if some module wants to run this syscall */
    tab_entry = arch_table[sysno];
    if (tab_entry && tab_entry->purelibc_choice) {
        ht = tab_entry->purelibc_choice(&sd);
        if (tab_entry->purelibc_wrap)
            return tab_entry->purelibc_wrap(ht, &sd);
    }
    /* forward the syscall to the Kernel */
    return native_syscall(sysno, sd.syscall_args[0], sd.syscall_args[1],
                        sd.syscall_args[2], sd.syscall_args[3],
                        sd.syscall_args[4], sd.syscall_args[5]);
}

```

The role of `puresyscall_landing` is to gather the system call arguments and call in sequence the `Pchoice`⁶ function and the `Pwrap`⁷ which perform the same task as the `CFunctions` and the `WINFunctions`, one thing to notice

⁶Purelibc choice

⁷Purelibc wrapper

is that now arguments live in the same process address space of the tracer and need no fetching.

Throughout UMView’s code one can find system call prefixed by “r_”, these are real system calls (directly forwarded to the Kernel), otherwise they will be virtualized by purelibc.

Purelibc’s Pchoice and Pwrap functions have the following types:

```

/* purelibc choice function */
hashtable_obj_t (*purelibc_choice)(syscall_descriptor_t *);
/* purelibc wrap function */
long int (*purelibc_wrap)(hashtable_obj_t *, syscall_descriptor_t *);

```

The only difference with their “real” counterpart (CFunctions and WFunctions) is that the Pwrap function has no *entry to/exit from* transition to take into account but simply is a single function call.

6.8 The Path Canonicalization Unit

UMView has a set of routines to deal with paths in order to always resolve relative paths in absolute ones.

The actual function used to resolve a relative path into an absolute one has been inherited from the former implementation of UMView and is called `umview_realpath`.

```

char *umview_realpath(const char *const name, char *resolved,
                      realpath_follow_t follow);

```

The first two input arguments of the function are self-explaining, the `follow` argument is an enumerated type allowing the values `FOLLOW_LINK` and `NO_`

`FOLLOW_LINK`, if `name` is a symbolic link then the first value tells the function to follow it otherwise the symbolic link will not be followed during the resolution process.

In the original implementation of UMView this was the only available function to deal with paths, during this dissertation, building on the previously done work, the canonicalization unit has been expanded to handle more cases.

The `umview_realpath` function is never actually called throughout UMView's code which uses more high-level functions built on it.

The new set of canonicalization API is the following:

```
char *umview_absolute_path(char *path, realpath_follow_t follow);
char *path_from_fd(int dirfd, char *path);
hashtable_obj_t *get_hte_from_path(char **path, realpath_follow_t follow,
                                   int set_epoch);
```

These functions use `umview_realpath` always storing an absolute path in the `path` variable while trying to consume as less memory as possible.

The first one, `umview_absolute_path`, simply resolves the path contained in the `path` argument returning the address of a string storing the absolute path.

The second one, `path_from_fd`, is used to obtain an absolute path when the intercepted system call is one of the at-type, such as `linkat`, `unlinkat`, `newfstatat` and so on. The variable's address storing the absolute path is the return value of the function.

The last one, `get_hte_from_path`, returns the address of the hashtable element representing the module in charge of dealing with that path. The `path` argument is a double pointer because if it is a relative path it will be resolved and stored in the `path` variable. If `setepoch` is 1 and an element is found

the virtual epoch will be set to the hashtable element one (epochs will be discussed in the relevant chapter).

6.9 The Epoch Unit

The epoch unit is yet one of the other features that UMView inherited from its former implementation with minor changes due to the new multi-thread environment.

UMView uses an extensive timestamping system to differentiate between epochs. Epochs in UMView work like a version control system for processes/textquoteright views, as a matter of fact every change in the view of a process results in a change of the process/textquoteright current epoch. Actually two type of epochs are used to timestamp events.

- `virtual_epoch`: this is the tracer-thread specific epoch, which means that different tracer-threads can have different epochs at the same time. This epoch can be both set back or updated;
- `epoch_now`: this is the unique global epoch shared by all tracer-threads. This epoch is a monotonic counter, it cannot be set back;

An epoch is just a counter which is incremented every time a change in the process/textquoteright view occurs and UMView obviously offers a set of API to deal with epochs.

```
void update_vepoch();
epoch_t update_epoch();
epoch_t set_vepoch(epoch_t e);
epoch_t get_vepoch();
epoch_t get_epoch();
epoch_t matching_epoch(epoch_t service_epoch);
```

- `update_vepoch()`: updates the current virtual epoch to the value of the global epoch;
- `update_epoch()`: increments the value of the global epoch by one and returns its previous value;
- `set_vepoch(epoch_te)`: sets the value of the virtual epoch to the one contained by `e` and returns its previous value;
- `get_vepoch()`: returns the value of the virtual epoch;
- `get_epoch()`: returns the value of the global epoch;
- `matching_epoch(epoch_tservice_epoch)`: if the virtual epoch is consistent with the service epoch it returns the matching epoch;

UMView using the Epoch unit allows the virtualization nesting for modules. The idea is that each operation on a process/textquoteright view has an associated timestamp. When a tracee executes a system call more than one module could be eligible for processing the call, what happens is that the module having the most recent timestamp is chosen.

In this way if several filesystems have been mounted at the same location the latest mount operation is the one seen by the tracee.

Once a module is chosen to process the intercepted system call the virtual epoch is moved back to the M epoch (which is the one of the module), as a matter of fact every time a module is loaded, the process/textquoteright view changes and the event is timestamped. In such a way all the system calls generated by the module will be captured by purelibc and executed in the environment at the time of M , this means that only the view changes older than the M epoch will be considered and not the more recent ones[5].

As an example we might consider the following situation: both the virtual and global epochs are 8; The user mounts a FS FS_1 on `/mnt` by the module

M_1 and both the epochs now advance to 9. `/mnt/image` (inside FS_1) is the image of the file system FS_2 , the user mounts it again on `/mnt` by the module M_2 , both the epochs are now 10.

When a user process reads `/mnt/somefile`, UMView finds out that that this file is inside the mountpoint subtree of both M_1 and M_2 .

The mount of FS_2 is the most recent and the module M_2 is thus selected. Module M_2 's implementation of read is called and the virtual epoch is moved back to 9. M_2 needs to read `/mnt/image` which is again in the subtree of both mountpoints. This time the second mount cannot be considered because is too recent for the current virtual epoch, thus M_1 's read is called.

6.10 The FS Unit

The **FS** unit is used by UMView to keep track of the filesystem information of every tracee, which include the root of the filesystem and the current working directory.

Each tracer-thread use a structure of type `umview_fs_t` which is visible only inside the FS unit but accessible through a set of APIs.

```
typedef struct umview_fs_t {
    pthread_rwlock_t lock;
    char *cwd;
    char *rootdir;
    size_t cwd_len;
    size_t rootdir_len;
    size_t count;
    int path_rewrite;
} umview_fs_t;
```

- `lock`: mutex used for sharing the `umview_fs_t` structure;

- `cwd`: the current working directory. All functions operating on this field are capable of autoresizing `cwd` on-demand;
- `rootdir`: the root of the filesystem. All functions operating on this field are capable of autoresizing `rootdir` on-demand;;
- `cwd_len`: the length of the string allocated to contain `cwd`;
- `rootdir_len`: the length of the string allocated to contain `rootdir`;
- `count`: used for reference counting;
- `path_rewrite`: not yet used (it will be used to implement `chroot` without root permissions);

The following is the available set of APIs used by UMView to interact with the FS unit:

```
umview_fs_t *umview_fs_create_fs(umview_fs_copy_method_t copy_method);
char *umview_fs_get_cwd();
void umview_fs_set_cwd(char *wd);
char *umview_fs_get_rootdir();
void umview_fs_set_rootdir(char *dir);
umview_fs_t *umview_fs_get_fs();
void umview_fs_set_fs(umview_fs_t *fs);
void umview_fs_destroy_fs();
void umview_fs_readlock();
void umview_fs_unlock();
```

There is a pair of getter/setter functions to interact with the `cwd` and `rootdir` fields which are capable of auto-resizing the fields if needed.

There is also a pair of getter/setter functions to set or retrieve the current `umview_fs_t` structure used in a tracer-thread.

The `umview_fs_readlock` and `umview_fs_lock` functions are used to obtain

the mutex of the FS structure and have therefore to be used whenever there is a necessity to read a specific field of the structure.

6.10.1 FS Structure Creation

The most interesting function in the FS unit is `umview_fs_create_fs`.

```
umview_fs_t *umview_fs_create_fs(umview_fs_copy_method_t copy_method);
```

This routine takes as its only input argument a `umview_fs_copy_method_t` parameter which is an enumerated type allowing the values `FS_NEW`, `FS_COPY` and `FS_REFERENCE`; according to the supplied value `umview_fs_create_fs` will either create a new FS structure, make a full copy of the one belonging to the calling process or create a reference of the calling process/textquoteright one. The newly created structure will be returned as the function's return value. FS structures can thus be shared between tracer-threads if the tracee child has been created with a `clone` system call having the flag `CLONE_FS`.

The only place where `umview_fs_create_fs` is used at the moment is when `UMView` needs to create a new tracer-thread in order to trace a child the tracee has just spawned.

6.11 The FD Table Unit

The FD Table unit is used to keep track of the file descriptors used by the tracee and to implement virtual file descriptors.

Each tracer-thread owns a structure of type `umview_fd_table_t` which is visible only inside the FD table unit but accessible through a set of APIs.

```
typedef struct umview_fd_table_t {  
    pthread_rwlock_t lock;
```

```

    size_t count;          /* this table can be shared by threads */
    size_t first_free_fd; /* first usable fd */
    size_t size;
    int to_close_on_exec;
    umview_fd_entry_t *table;
} umview_fd_table_t;

```

- `lock`: mutex used for sharing the `umview_fd_table_t` structure;
- `count`: used for reference counting;
- `last_free_fd`: number of the first usable file descriptor;
- `size`: allocated size of the `table` variable;
- `to_close_on_exec`: 1 if this table contains some file descriptors which are to be closed on the execution of an `execve` system call;
- `table`: the actual table containing informations about the open file descriptors. All functions operating on this field are capable of autore-sizing `table` on-demand;

The following is the structure of each entry of the file descriptor table:

```

typedef struct umview_fd_entry_t {
    hashtable_obj_t *ht;
    short int tracked;
    short int close_on_exec;
    umview_fd_type_t type;
    void *private;
    char *path;
} umview_fd_entry_t;

```

- `ht`: a hashtable element associated with this file descriptor entry;

- `tracked`: 1 if this file descriptor is kept track of by UMView;
- `close_on_exec`: 1 if this file descriptor has been created with the `O_CLOEXEC` flag set;
- `type`: the file descriptor entry type. It is an enumerated type allowing values `FD_REAL`, `FD_VIRTUAL` and `FD_KERNEL`.
A file descriptor of type `FD_KERNEL` is opened and used only by the tracee and not virtualized by UMView or its loaded modules.
A FD of type `FD_REAL` is opened by either UMView directly or one of the loaded modules.
A FD of type `FD_VIRTUAL` is a VFD.
- `private`: pointer to data associated with a `FD_REAL` or `FD_VIRTUAL` file descriptor;
- `path`: string variable containing the path associated to the file descriptor. The associated FD's path is stored only if the FD is either `FD_REAL` or `FD_VIRTUAL`;

6.11.1 Tracking File Descriptors

The role of the FD Table unit is to keep track of all the FDs used by the tracee, therefore every FD opened either directly by the tracee or by a UMView's module gets its place in the UMView file descriptor table.

The function used to keep track of a single file descriptor is `umview_fd_open`.

```
int umview_fd_open(char *path, int fd, hashtable_obj_t *hte,
                  umview_fd_type_t fd_type, int close_on_exec);
```

When calling `umview_fd_open` if `fd_type` is either `FD_VIRTUAL` or `FD_REAL` the `path` variable must point to a dynamic allocated string containing the

path associated to the FD and `hte` must point to a hashtable element representing a module. If instead `fd_type` is `FD_KERNEL` both `path` and `hte` must be `NULL`.

6.11.2 FD Collisions

Since the FD unit tries to keep track of any kind of open FD collisions between `FD_KERNEL` or `FD_REAL` FDs and `FD_VIRTUAL` ones can occur.

To better understand how collisions can happen it is useful to discuss an example; a module creates the VFD FD_1 with value 3, consequently the same module opens a real FD FD_2 and the Kernel returns the value 3. At this point we have a collision because the position 3 of the file descriptor table is already occupied by FD_1 and FD_2 cannot be stored in the same position.

To solve FD collisions the FS unit uses the following routine.

```
static int fd_solve_collision_nolock(int fd)
{
    int newfd = umview_fd_get_free_fd_nolock();
    newfd = r_dup2(fd, newfd);
    if (newfd == -1) {
        perror("assertion failed in fd_solve_collision_nolock()");
        pthread_exit(NULL);
    }
    r_close(fd);
    return newfd;
}
```

The mechanism is quite simple, every time a collision occurs the FS unit uses the `umview_fd_get_free_fd_nolock` to find the first usable file descriptor

position and performs a `dup2` system call to duplicate the just created file descriptor into the new position.

Going back to the example, `UMView` would call `fd_solve_collision_nolock` on FD_2 which would get the new value 4, in such a way both FD_1 and FD_2 can be stored causing no collision.

6.11.3 FD Table Creation

The FD Table unit's function to create a FD table works similarly to the one used for creating FS structures in the FS unit.

```
umview_fd_table_t*
umview_fd_create_table(umview_fd_copy_method_t copy_method);
```

This function takes as its only input argument a `umview_fd_copy_method_t` parameter which is an enumerated type allowing the values `FD_TABLE_NEW`, `FD_TABLE_COPY` and `FD_TABLE_REFERENCE`; according to the supplied value `umview_fd_create_table` will either create a new FD table structure, make a full copy of the one belonging to the calling process or a create a reference of the calling process' one. The newly created structure will be returned as the functions return value. FD tables, just like the real ones, can thus be shared between tracer-threads if the tracee child has been created with a clone system call having the flag `CLONE_FILES`. The only place where `umview_fd_create_table` is used at the moment is when `UMView` needs to create a new tracer-thread in order to trace a child the tracee has just spawned.

6.11.4 FD Table API

The following is the complete set of API offered by the FD Table unit, including some of the functions which have just been described in depth.

```

umview_fd_table_t *umview_fd_get_table();
int umview_fd_get_free_fd();
void umview_fd_close_on_exec();
int umview_fd_is_tracked(int fd);
void umview_fd_set_table(umview_fd_table_t *newtable);
umview_fd_table_t*
umview_fd_create_table(umview_fd_copy_method_t copy_method);
void umview_fd_destroy_table();
umview_fd_entry_t *umview_fd_get_hte(int fd);
int umview_fd_is_virtual(int fd);
int umview_fd_open(char *path, int fd, hashtable_obj_t *hte,
                  umview_fd_type_t fd_type, int close_on_exec);
void umview_fd_close(int fd);
char *umview_fd_get_path(int fd);
int umview_fd_get_type(int fd);
int umview_create_fd(int close_on_exec, void *private);
void *umview_fd_set_private(int fd, void *private);
void *umview_fd_get_private(int fd);

```

The majority of the API functions have already been covered, the remaining ones have self-explanatory names.

6.12 The UMView System Call Table

UMView has its unique system call list, this decision comes from the necessity of having a stable and UMView's specific reference. System calls are added to the list as soon as they are implemented in UMView's core, at the time being the supported system calls are the one shown in table 5.1.

6.12.1 UMView Configuration File

UMView employs a single configuration file to manage the supported system calls. This file has the following structure:

```
syscalls... :CFunction, WINFunction, WOUTFunction, Pchoice, Pwrap
```

Each line describes a single or a set of system calls, as matter of fact the first element delimited by the semicolon is a comma-separated list of system, the first of this list is considered as the UMView system call of the set. After the semicolon there must be a comma-separated list of functions name which indicate in order: the CFunction for the set of system calls, the WINFunction, the WOUTFunction, the Pchoice function and the Pwrap function.

The following is a snippet from the current UMView's configuration file.

```
lstat, stat, newfstatat, fstat: lstat, lstat, virtual, NULL, NULL
access, faccessat: access, access, virtual, NULL, NULL
readlink, readlinkat: readlink, readlink, virtual, NULL, NULL
access, faccessat: access, access, virtual, NULL, NULL
readlink, readlinkat: readlink, readlink, virtual, NULL, NULL
```

6.12.2 Build Scripts

CFunction and WFunction Generation

During CMake execution a python script residing in `scripts/wrappers_gen.py` is called, this script is in charge of creating the header files `include/wrappers.h` and `include/choices.h` which export CFunctions, WFunctions, Pchoice and Pwrap functions.

The script simply parses UMView's configuration file to find out each system call associated CFunction, WFunctions, Pchoice and Pwrap function.

Table Generation

CMake also executes another python script `scripts/syscall_table_gen.py` which parses UMView's configuration file and creates the source file containing UMView's system call table `src/umview/syscall_table.c` on-the-fly.

The UMView system call table is just an array of `syscall_tab_entry` elements.

```
typedef struct syscall_tab_entry {
    /* syscall number */
    int sysno;
    /* the choice function: this function tells the service which have to
     * manage the system call */
    hashtable_obj_t *(*choice_function)(syscall_descriptor_t *);
    /* wrapin function: this function is called in the IN phase of the
     * syscall */
    void (*wrapin_func)(hashtable_obj_t *ht, syscall_descriptor_t *,
                        syscall_behaviour_t *);
    /* wrapout function: this function is called in the OUT phase of the
     * syscall */
    void (*wrapout_func)(syscall_descriptor_t *, syscall_behaviour_t *);
    /* purelibc choice function */
    hashtable_obj_t *(*purelibc_choice)(syscall_descriptor_t *);
    /* purelibc wrap function */
    long int (*purelibc_wrap)(hashtable_obj_t *, syscall_descriptor_t *);
} syscall_tab_entry;
```

Each entry in the system call table tells how each UMView's system call have to be processed, in terms of which CFunction, WFunctions, Pchoice and Pwrap functions have to be called by the Execute unit.

6.13 The Arch System Call Table

The Arch system call table is specific for each architecture to which UMView has been ported and is created by the `scripts/archtable_gen.py` script during the execution of the CMake build system.

The arch table source file is created in `src/umview/arch_table.c` and differently from the UMView's system call one this table is an array of pointers of `syscall_tab_entry` elements.

6.13.1 CFunction and WFunctions Resolution

Each position i of the arch system call table represents the i_{th} system call of the target architecture and the `syscall_tab_entry` pointer contained in that position points to an element of the UMView's system call table.

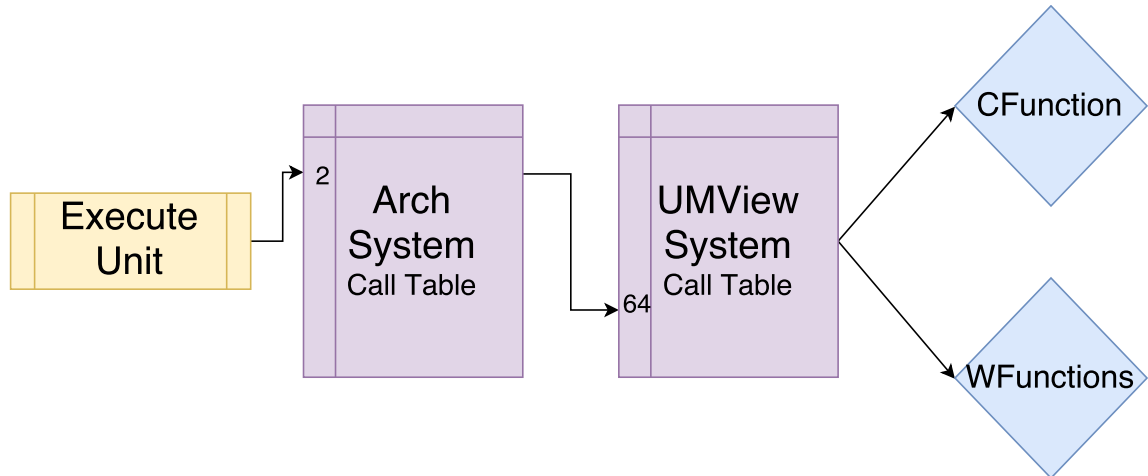


Figure 6.5: CFunction and WFunctions resolution

This is exactly how the Execute unit chooses the CFunction and WFunctions to use upon interception of a system call by the tracing unit, it simply retrieves the address of both functions from the element of the UMView's system call table pointed by the arch table position of the intercepted syscall.

In this way there are no delay penalties for the Execute unit and there's also no management effort for both the UMView and arch table since they are automatically created by python scripts according to the UMView's configuration file.

Conclusions and Future Developments

UMView other than being a partial virtualization software brings to new life the concepts introduced by View-OS which offers a new and broad vision on virtualization and specifically on the virtualization we use every day on our machines.

The aim of this dissertation was to design and develop a new modular and flexible implementation of UMView which will hopefully be used to share knowledge and ideas about partial virtualization.

UMView is a powerful piece of software whose possibilities and application can only be bounded by the ideas of developers and users.

Nevertheless there is still a long path ahead, as a matter of fact there is much room for improvement in the work done during this dissertation.

The tracing unit could be vastly improved by adding support for new (yet to come) tracing backends. Unfortunately, ptrace has not been designed to implement partial virtualization and as a matter of fact UMView's tracing unit is more complicated than how it should be because it has to deal with ptrace's weirdnesses (from a partial virtual machine perspective). An example is the code to manage *entry to/exit from* system call transition, this could be eliminated if another tracing backend would be available because UMView is not really interested in such transitions.

In terms of coverage a lot of system calls are yet to be virtualized and supported by UMView, including some harder system calls to deal with like `poll` or `select`.

New WOUTFunctions should be implemented for each system call capable of creating new file descriptors, otherwise the FD Table unit will not be able to correctly perform its task.

In terms of what's the future like for UMView there is a lot going on in Virtual Square's laboratory and UMView's codebase is far to be frozen.

The following is a list of interesting ideas for UMView's future:

- implement the possibility of executing the `chroot` command without having root permissions;
- develop a Go library to allow writing UMView's module in Go;
- develop a Python library to allow writing UMView's module in Python;
- develop a C++ library to allow writing UMView's module in C++;
- integrate **fuse** in the codebase as a way to manage system calls operating on file descriptors more efficiently;
- add support for the ARM architecture;
- add the possibility for module to add new virtual system calls;
- implement an inter-module communication mechanism in order to let modules know when a certain event happens;

Bibliography

- [1] Eric W Biederman and Linux Networx. “Multiple instances of the global linux namespaces”. In: *Proceedings of the Linux Symposium*. Vol. 1. Citeseer. 2006, pp. 101–112.
- [2] Renzo Davoli. *VDE components*. 2012. URL: http://wiki.v2.cs.unibo.it/wiki/index.php?title=VDE#VDE_components (visited on 02/12/2017).
- [3] Renzo Davoli. “VDE: Virtual Distributed Ethernet”. In: *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*. TRIDENTCOM ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 213–220. ISBN: 0-7695-2219-X. DOI: 10.1109/TRIDNT.2005.38. URL: <http://dx.doi.org/10.1109/TRIDNT.2005.38>.
- [4] Renzo Davoli. *Virtual Square Wiki*. 2014. URL: http://wiki.v2.cs.unibo.it/wiki/index.php?title=Main_Page (visited on 02/12/2017).
- [5] Renzo Davoli and Michael Goldweber. *Virtual Square: Users, Programmers & Developers Guide*. 2011.
- [6] Michael Kerrisk. *Linux Programmer’s Manual – Linux Namespaces*. 2016. URL: <http://man7.org/linux/man-pages/man7/namespaces.7.html> (visited on 02/11/2017).

- [7] Michael Kerrisk. *Linux Programmer's Manual – PTRACE*. 2016. URL: <http://man7.org/linux/man-pages/man2/ptrace.2.html> (visited on 02/17/2017).
- [8] Wikipedia User V4711 License: CC-BY-SA-4.0. *A high-level overview of the KVM/QEMU virtualization environment*. 2015. URL: https://commons.wikimedia.org/wiki/File:Kernel-based_Virtual_Machine.svg (visited on 02/07/2017).
- [9] Renzo Davoli Ludovico Gardenghi and Michael Goldweber. “View-OS: A New Unifying Approach Against the Global View Assumption”. In: *Computational Science – ICCS 2008. 8th International Conference, Krakow, Poland, June 23-25, 2008, Proceedings, Part I*. Springer Berlin Heidelberg, 2008, pp. 287–296.
- [10] J. E. Smith and R. Nair. “The architecture of virtual machines”. In: *Computer* 38.5 (May 2005). Ed. by IEEE, pp. 32–38. ISSN: 0018-9162.
- [11] Wikipedia. *Docker*. 2017. URL: [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software)) (visited on 02/11/2017).
- [12] Wikipedia. *KVM*. 2017. URL: https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine (visited on 02/10/2017).
- [13] Wikipedia. *QEMU*. 2017. URL: <https://en.wikipedia.org/wiki/QEMU> (visited on 02/07/2017).

Acknowledgements

I would like to thank Prof. Renzo Davoli for always being ready to share his knowledge and experience to curious and passionate students and for making the University feel like a big crazy laboratory where anyone can experiment his ideas and have fun.

I would also like to thank all the people that for one reason or another joined the Virtual Square team and contributed to the Virtual Square laboratory.

Ringraziamenti

Grazie alla mia famiglia, Salvatore, Sara e Maria Grazia, per aver sempre messo in discussione ogni mia scelta, per la fiducia che ripongono in me ogni giorno e per concedermi sempre la possibilità di realizzare i miei sogni.

Grazie a Jessica, la cui gioia contagiosa rende ogni giorno speciale, per esserci sempre. Grazie soprattutto perchè credi sempre in me nonostante non sia ancora capace di aprire una porta o di camminare senza scontrarmi contro il mondo.

Grazie a Federico, Fabio, Gianluca, Giovanni, Giovanni, Matteo, Marco, Simone, Stefano e Umberto per aver reso indimenticabile un corso di Laurea in Informatica Magistrale e per avermi mostrato come l'Università non sia fatta solo di esami e progetti ma soprattutto di esperienze, divertimento e amici.