View-OS: Change your View on Virtualization.

Renzo Davoli
University of Bologna
Dept. of Computer Science
40127 Bologna
Italy
`renzo@cs.unibo.it`

Michael Goldweber
Xavier University
Dept. of Mathematics and Computer Sciences
Cincinnati, OH 45207-4441
USA
`mikeyg@cs.xu.edu`

1 Abstract

The View-OS concept is defined as allowing each process to have its own view of its *execution environment*. For example, each process can have its own view for its file system, networking support, user-id, system-name, etc. Umview and kmview are two proof of concept implementations of the View-OS concept.

Technically, umview and kmview are user-level, system-call based, partial, modular, virtual machine monitors. They virtualize a subset of the kernel requests (system calls) depending upon which umview (or kmview) modules have been loaded and upon which kind of virtualization the user configured.

The effectiveness of this approach is best illustrated by providing some examples of what umview (or kmview) can accomplish.

2 View-OS

A cornerstone of modern operating systems design is the *global view* concept: all processes running on a given system share the same view regarding the services provided by the system.

We define the *view* of a process as its perception of the following entities:

- filesystem namespace, including the related ownership and permission information,
- networking configuration,
- system name,
- current time,
- devices, etc.

Apart from some unique exceptions, all the processes running on the same computer share the same view; e.g. the same pathname refers to the same file, the processes use one shared TCP-IP stack for networking hence the same set of IP addresses and routing policies. We refer to this behavior as the *Global View Assumption*. Alternatively, two processes, each running on a different computer, or virtual machine, have in general independent views. For example, pathnames can/will refer to different file system hierarchies, furthermore, these two processes will use different networking stacks with distinct addresses and policies.

Given the restrictive nature of the *Global View Assumption*, many exceptions, over time, have been developed. Consider *chroot*, *fakeroot*, `/dev/tty` (viewed as a different device depending on the controlling terminal), or `/proc/self`. These facilities (i.e. system calls, virtual files, and programs) allow a process to partially deviate its view from the global view.

Unlike standard operating systems, virtual machines and containers[9] (also known as Zones[8]) can provide individual processes with unique views. A virtual machine emulates the API of a (possibly different) system creating in this way a new *Global View Assumption* domain for all of the processes running on/under it. Containers provide processes with private namespaces, therefore, via containers processes can have different views. Furthermore, using a container, it is possible to implement a light weight virtual private server or a process checkpoint/restart facility [1]. However a container is a kernel feature whose use, therefore is restricted only for system administrators.

View-OS, by design, is not restricted to system administrators, allowing for a user level view reconfiguration. In View-OS, processes change their view by exploiting the standard resources provided by the kernel. For example, a process can *mount* a disk image file or define its own *virtual private network (VPN)* provided the process is allowed to access the disk image file or initiate a TCP connection that can be used for tunneling the network traffic.

3 View-OS tutorial

This section introduces some examples of View-OS usage. Since the set of View-OS application domains goes far beyond these examples, the goal of this section is to provide the reader with an understanding of both the flexibility of and the safety provided by the current View-OS implementations.

There are currently two implementations of View-OS: `umview` and `kmview`. The former (`umview`) runs on standard Linux kernels while the latter is faster, provides a more complete virtualization, but needs some extra kernel support (see section 4).

To run the following tests just download the source code from the View-OS sourceforge repository (the latest svn code is recommended), compile, and install.

The command:

```
$ umview bash
```

starts a View-OS domain and runs a bash shell inside it.

For users running X-Windows who prefer to have their virtual environment in a separate terminal window, enter:

```
$ umview xterm
```

Umview and kmview are already part of many GNU-Linux distributions. (e.g. Debian-sid, Ubuntu in the *universe* repositories.) Since the View-OS project is in rapid development and distributions take time to upgrade their packets, readers should be warned that some of the following experiments may fail or behave differently if using a View-OS version prior to 0.8.

3.1 Virtual installation/upgrade of software

In this scenario a user wants to experiment with some software from a GNU-Linux distribution. Either the software is not installed or only an outdated version is available. Furthermore, either the user does not have root access or does not want to install the software permanently/globally on the system.

```
$ um_add_module viewfs
$ mkdir /tmp/newroot
$ viewsu
# mount -t viewfs -o mincow,except=/tmp,vstat /tmp/newroot /
# aptitude install mynewsoftware
```

The first command, `um_add_module`, is used to load View-OS modules. In this case the `viewfs` module is loaded. `Viewfs` is the module that provides for *file system patchworking*, allowing a user to redefine their view of the file system by combining the existing file system in various ways.

`viewsu` is the View-OS counterpart of the `su` command. As with fakeroot[12] the user acquires super user privileges in the virtual environment. `viewsu` does not require a password.

`/tmp/newroot` is a new empty directory which then gets *mounted* as the new root directory using the `mincow` option of `viewfs`. `mincow` stands for minimal copy on write: the /tmp/newroot and / hierarchies get merged. Furthermore, the complete file system is made writable with modifications taking place on the real filesystem when allowed, or the target files are copied and modified on /tmp/newroot when not.

A View-OS file system mount can have exceptions. In this example, `/tmp` has been left "unmounted," so as to provide better performance. Finally, the `vstat` option allows for the creation of files with virtual `stat` information, i.e. assign ownership, permissions, and create device special files.

At this point the user can install or upgrade the software using the standard tools provided by their Linux distribution, such as Debian's *aptitude* facility.

When the installation process terminates the user can exit the viewsu subshell and run/experiment with the software. From inside the View-OS instance, all programs, libraries, manpages, etc. have been installed in their standard directories, exactly as if root had performed the installation globally on the system.

This example shows how View-OS, and more specifically its `viewfs` module implements and extends the features provided by UnionFS and fakeroot.

At another point in time the user may elect to remount `/dev/newroot` as a read-only merged hierarchy:

```
mount -t viewfs -o merge,except=/tmp,vstat /tmp/newroot /
```

Mounting `/dev/newroot` in this way makes it possible to set up packages with pre-installed applications ready to be loaded when needed; just like what one can do using klik[7]

3.2 Virtual multi stack networking

This example shows the versatility of `umnet`, the View-OS network virtualization module.

ViewOS processes can use several networking stacks simultaneously. For example a user can set up multiple TCP-IP stacks, each one having its own interface(s) and address(es). Furthermore, each stack may represent a different implementation or simply implement differing policies.

Each process also has its own default stack for each protocol family, with `/dev/net/default` being the ViewOS domain wide default.

```
$ um_add_service umnet
$ mount -t umnetlwipv6 none /dev/net/default
$ ip link set vd0 up
$ ip addr add 10.1.2.3/24 dev vd0
$ ip addr
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
    inet6 ::1/128 scope host
    inet 127.0.0.1/8 scope host
2: vd0: <BROADCAST,UP> mtu 1500
    link/ether 02:02:5a:44:e2:06 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::2:5aff:fe44:e206/64 scope link
    inet 10.1.2.3/24 scope global
```

```
$ ssh 10.1.2.1
...
```

In the example above, the user *mount*'s a lwipv6 stack on `/dev/net/default`. *lwipv6* is a hybrid IPv6/IPv4 networking stack implemented as a library.[1]

The default configuration of the stack defines a Virtual Distributed Ethernet (VDE)[3, 5] virtual interface connected to the default virtual ethernet switch.

The user can run standard commands to configure, activate lwipv6 interfaces and run networking applications.

lwipv6 also supports `tun` or `tap` interfaces. The following example uses several stacks and different kinds of interfaces.

```
$ um_add_service umnet
$ mount -t umnetlwipv6 -o tn0=tunx none /dev/lwip0
$ mount -t umnetlwipv6 -o tp0=tapx,vd0=/tmp/switch none /dev/lwip1
$ mstack /dev/lwip0 ip addr
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
    inet6 ::1/128 scope host
    inet 127.0.0.1/8 scope host
2: tn0: <> mtu 0
    link/generic
$ mstack /dev/lwip1 ip addr
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
    inet6 ::1/128 scope host
    inet 127.0.0.1/8 scope host
2: vd0: <BROADCAST> mtu 1500
    link/ether 02:02:47:98:ad:06 brd ff:ff:ff:ff:ff:ff
3: tp0: <BROADCAST> mtu 1500
    link/ether 02:02:03:04:05:06 brd ff:ff:ff:ff:ff:ff
$
```

After loading the `umnet` module, two additional network stacks are created, giving processes a total of three network stacks available for simultaneous use. The mount command is used twice to create the two additional stacks. `/dev/lwip0` has a *tun* interface, while the third stack, `/dev/lwip1`, defines two interfaces, a *tap* interface and a *VDE* interface.

Users indicate which stack a given command should run on via the `mstack` facility, or can simply use the default kernel stack. For example:

```
$ mstack /dev/lwip0 ssh remote.host.edu
$ mstack /dev/lwip1 firefox &
$ wget http://download.it/now
```

It is also possible to write programs which access multiple stacks simultaneously: View-OS defines `msocket`, a new (virtual) system call, which supports this feature (see 6.2)

[1]There is just one IPv6 dispatching engine in `lwipv6`, thus it is not a dual stack service. `lwipv6` manages IPv4 packets as a back-compatibility feature.

3.3 Mount a filesystem

View-OS provides the `umfuse` module for mounting filesystems. `umfuse` is an implementation of FUSE[10] as a virtual service, and therefore all fuse modules are source compatible with `umfuse`.

```
$ um_add_service umfuse
$ mount -t umfuseext2 -o ro ext2filesystemimage /mnt
$ mount -t umfusestrangefilesystem strangeimage /mnt2
```

Once `umfuse` has been loaded, it is possible to mount any kind of file system. The VirtualSquare Lab currently provides experimental submodules for ext2/ext3, iso9660 and fat/vfat file systems. When mounting an “unknown” file system, as in the `umfusestrangefilesystem` example above, the user must provide the necessary submodule for the file system implementation.

Several FUSE-compatible file system implementations have been successfully ported to run under `umfuse` (e.g. sshfs and encfs).

3.4 Partition a filesystem image and mount its partition

This example shows how View-OS virtualizations can be nested illustrating how the “view” provided by a loaded module can be used as the basis for the creation of another “view” by a further loaded module. In essence, a virtual view built on top of a virtual view, built on top of the actual system.

```
$ um_add_service umdev
$ viewsu
# dd of=/tmp/diskimage bs=1024 count=0 seek=1024000
# mount -t umdevmbr /tmp/diskimage /dev/hda
```

Here the `umdev` module creates the view of a disk device on an empty file image. The user can use the standard `fdisk` utility to partition the virtual disk

```
# fdisk /dev/hda
Device contains neither a valid DOS partition table, nor Sun, SGI
or OSF disklabel
Building a new DOS disklabel with disk identifier 0xd403417d.
..
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-127, default 1): 1
Last cylinder, +cylinders or +size{K,M,G} (1-127, default 127): 127
Command (m for help): p

Disk /dev/hda: 1048 MB, 1048576000 bytes
255 heads, 63 sectors/track, 127 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0xd403417d

Device Boot      Start         End      Blocks   Id  System
/dev/hda1               1         127     1020096   83  Linux
```

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
```

The user can now create a file system on the virtual partition `/dev/hda1` and then mount it via the `umfuse` module.

```
# mkfs.ext2 /dev/hda1
mke2fs 1.41.8 (20-Jul-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
63872 inodes, 255024 blocks
12751 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=264241152
8 block groups
32768 blocks per group, 32768 fragments per group
7984 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.

# um_add_service umfuse
# mount -t umfuseext2 -o rw+ /dev/hda1 /mnt
# ls -l /mnt
total 16
drwx------ 2 root root 16384 2009-09-16 11:57 lost+found
```

3.5 User Mode `chroot`

Due to the global view assumption, the `chroot` system call has the potential to be a serious security threat. Therefore, it usage is restricted to root (or to processes owning the `CAP_SYS_CHROOT` capability).

Interestingly, `chroot` is a system call that was created to enhance system security by providing for the creation of *jails*, i.e. filesystem sandboxes. Processes running inside a `chroot` jail can only see a subtree of the file system; thus hiding all the data/files residing outside the "visible" subtree. Unfortunately this can lead to having the contents of sensible files like `/etc/passwd` be redefined, negatively affecting the behavior of setuid programs. The usage of *chroot* system call is therefore restricted because of the *global view assumption*.

In View-OS it is possible for users to create `chroot` "cages," where changes apply only to the process view.

A minimal tree for `chroot` can be created as follows:

```
$ mkdir /tmp/root /tmp/root/bin /tmp/root/lib
$ cp /bin/busybox /tmp/root/bin
$ cp /lib/libm-2.9.so /lib/libc-2.9.so /tmp/root/lib
$ cd /tmp/root/lib
$ ln -s libm-2.9.so libm.so.6
$ ln -s libc-2.9.so libc.so.6
$ cd /
```

The directory `/tmp/root` contains `busybox` along with the minimal set of libraries needed to run it.

The `chroot` cage can be activated using the standard command:

```
$ exec /usr/sbin/chroot /tmp/root /bin/busybox sh

BusyBox v1.13.3 (Debian 1:1.13.3-1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ $
```

Both View-OS implementations support `chroot` as a core feature.

Alternatively, the `viewfs` module can also create a `chroot` cage

```
$ um_add_service viewfs
$ exec busybox sh

BusyBox v1.13.3 (Debian 1:1.13.3-1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ $ mount -t viewfs -o move,permanent /tmp/root /
/ $
```

In both cases the file system view from the new busybox shell is a strictly limited tiny cage.

```
/ $ ls -lR /
/:
drwxr-xr-x  2 1000  1000     4096 Sep 17 13:37 bin
drwxr-xr-x  2 1000  1000     4096 Sep 17 13:37 lib

/bin:
-rwxr-xr-x  1 1000  1000   401216 Sep 17 13:37 busybox

/lib:
-rwxr-xr-x  1 1000  1000  1302732 Sep 17 13:37 libc-2.9.so
lrwxrwxrwx  1 1000  1000       11 Sep 17 13:37 libc.so.6 -> libc-2.9.so
-rw-r--r--  1 1000  1000   149328 Sep 17 13:37 libm-2.9.so
lrwxrwxrwx  1 1000  1000       11 Sep 17 13:37 libm.so.6 -> libm-2.9.so
/ $
```

3.6 Create a Ramdisk and use it

In this scenario a user creates and then makes use of a ramdisk.

This file system is stored in an array allocated by the `umdevramdisk` submodule and is therefore only in the "view" of those processes running in the particular View-OS instance. A direct implication of this is that other processes cannot access the ramdisk file system, not because it is invisible to them, but because, for them, it simply does not exist. Even the system administrator of the host machine has no direct way to inspect the file system contents; the ramdisk file system is just a chunk of private memory allocated to an individual process. A persistent system administrator would need to dump the memory of the View-OS monitor program and search this dump for the file system data - no where easy a task as simply entering a forbidden directory.

The following example uses a FAT file system but the same procedure applies to other file system types.

```
$ um_add_service umdev
$ um_add_service umfuse
$ um_add_service umproc
$ mount -t umdevramdisk -o size=100M none /dev/hdx
$ /sbin/mkfs.vfat /dev/hdx
mkfs.vfat 3.0.3 (18 May 2009)
$ mount -t umfusefat -o rw+ /dev/hdx /mnt
$ mount
rootfs on / type rootfs (rw)
/dev/root on / type ext3 (rw,errors=remount-ro,data=ordered)
tmpfs on /lib/init/rw type tmpfs (rw,nosuid,mode=755)
... ...
none on /proc/mounts type proc (ro)
none on /dev/hdx type umdevramdisk (size=100M)
/dev/hdx on /mnt type umfusefat (rw+)
$
```

The example also shows the virtualization of `/proc/mounts` by the `umproc` View-OS module. The mount command in this example is that provided by `busybox`.

3.7 Virtualization of running processes

In this scenario a user wants to provide a virtualization to a running process without having to restart the process from within a View-OS session.[2] A standard process (i.e. one running without any virtualization) can also access a virtual file or a virtual network using this same technique.

For this example the user needs two shells. In the first shell –a standard linux shell– the user creates an empty directory.

```
sh1 $ mkdir /tmp/mnt
sh1 $ ls /tmp/mnt
sh1 $
```

In a second shell –one controlled by a View-OS monitor– the user mounts a filesystem on `/tmp/mnt`.

```
sh2 $ um_add_service umfuse
sh2 $ mount -t ext2 /tmp/linux.img /tmp
sh2 $ ls /tmp/mnt
bin  boot  dev  etc  lib  lost+found  mnt  proc  sbin  tmp  usr
sh2 $ um_attach 18672
```

[2]Currently this example only works with certain limitations (mmap is not yet supported) and only with `umview`. Running process virtualization support has not yet been implemented in `kmview`.

where 18762 is the pid of sh1. After performing the `um_attach` command, the user can now access the virtually mounted filesystem from both `sh1` and `sh2`.

```
sh1 $ ls /tmp/mnt
bin  boot  dev  etc  lib  lost+found  mnt  proc  sbin  tmp  usr
```

With `um_attach` it is possible to *on the fly* change the view of already running processes in any way supported by View-OS: add or update available applications, remove or change the contents of system files (e.g. /etc/hosts, /etc/passwd etc), force the process to work on a virtual network etc.

3.8 Process *proper time*

In this final scenario, View-OS can be used to modify the *proper time* of a process making its time run slower or faster than the time measured by the other processes.

From a standard shell, launch an `xclock` with a seconds hand to be used as a reference time.

```
$ xclock -update 1 &
```

In a View-OS shell run the same command.

```
$ xclock -update 1 &
$ um_add_service ummisc
$ mount -t ummisctime none /tmp/mnt
```

The two clocks should show the same time.

The file `/tmp/mnt/frequency` stores the frequency in Hz of a virtual second; initially set to 1.0. By editing `/tmp/mnt/frequency` it is possible to change the pace of the process' *proper time*.

```
$ echo 2 > /tmp/mnt/frequency
```

Virtual time now flows/ticks twice as fast as real time. Similarly, a value less than 1 causes the virtual time to flow/tick slower than the standard time. It is even possible to set the frequency to a value less than or equal to zero. While `xclock` has not been designed to animate the hands counterclockwise, by issuing several `date` commands one can see that the virtual time is flowing backwards.

At first glance, it may not be clear what practical application modifying the *proper time* of a process may have, beyond *nix geek appeal. It turns out that the potential speedup one may achieve by running a parallel algorithm on N processors can be crudely estimated under View-OS by modifying the process's proper time as a function of the number of non-idle processors available at various points in the computation: Simply update the time-frequency to represent the number of non-idle processors.

In a broader sense, this example illustrates the power and flexibility of a non-dedicated general-purpose modular virtualization engine. Instead of developing a single virtualization engine that can handle all the scenarios we could conceive of, our goal was to create a flexible virtualization environment that can be configured/extended to not only handle the scenarios we did conjecture, but to be able to handle the new scenarios that we and other View-OS users have yet to conceive.

4 Behind the scenes

`Umview` and `kmview` are two reference implementations of View-OS created by the VirtualSquare Lab. Umview uses `ptrace` to capture and modify process system calls, and therefore runs on standard Linux kernels. Unfortunately, the inherent limitations of ptrace's current implementation negatively

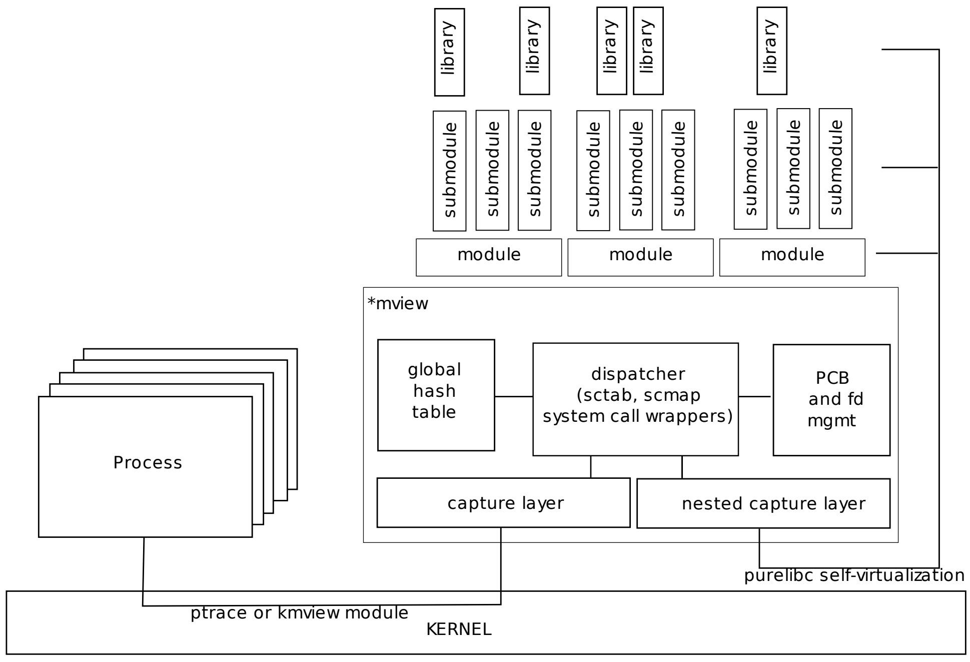

Figure 1: A simplified view of ∗mview structure

affect umview performance. (See section 5.) To address this, we propose some general purpose, back-compatible, safe optimizations for ptrace. (See section 6.) Alternatively, the other View-OS reference implementation, kmview, requires a kernel module based on Roland McGrath's utrace support[6].

This section is meant to provide the reader with a basic understanding of the internal structure of both View-OS implementations. (See Figure 1.) Interested readers are directed to the VirtualSquare Lab's wiki [4] or to the internal comments in the code itself for a more detailed description of View-OS internals

Though umview and kmview are based on different system call capture technologies, they nonetheless share the major part of their code. The only only difference between their code bases is their system call capture layers. For the remainder of this paper ⋆mview is used to refer to either implementation, i.e. when the differences between umview and kmview are not relevant.

⋆mview is structured around modules, with each module providing one type of virtualization. Several modules use submodules to support specific virtualization "flavors" or extensions. A list of the currently supported modules and submodules is presented in Table 1.

⋆mview uses a global hash table to keep track of active virtualizations. Each module inserts objects into the hash table to activate their particular virtualization. The hash table manages different kinds of objects:

- Pathname objects - When a system call involves a pathname (e.g. `open`, `stat`) the request is matched against the registered pathnames present in the hash table. When several prefixes of the pathname match entries in the hash table, the most recent wins. A pathname object is added to the hash table as a side effect of a successful mount operation.
- Filesystem type objects - These objects are used by the `mount` system call. When a module is loaded, a filesystem type object is usually added to the hash table.
- Protocol family objects - To support `socket` system calls.
- Definition of major/minor numbers or ranges - For supporting device special files.
- System call numbers.

module	description	submodule	submodule description
umproc	`/proc/mounts` virtualization		
umfuse	user-mode fuse	umfuseext2 (fuse-ext2 for FUSE)	ext2 implementation
		umfuseiso9660 (fuseiso9660)	iso9660
		umfusefat (fusefat)	fat and vfat
		umfusessh (sshfs)	remote file system via ssh
		umfuseencfs (encfs)	encrypted file system
umnet	network multi stack virtualization	umnetnull	null stack
		umnetlwipv6	IPv4/v6 hybrid stack
		umnetlink	move/merge stacks
		umnetnative	bypass for kernel provided native stack
umdev	device virtualization	umdevmbr	DOS master boot record support
		umdevnulli	null device
		umdevramdisk	virtual ram disk
		umdevtap	virtual tuntap
ummisc	system call based virtualization	ummisctime	time virtualization
		ummiscuname	uname id virtualization
viewfs	filesystem patchworking		
umbinfmt	virtualization of interpreter selection for execve		

Table 1: View-OS modules and submodules

The system call capture layer notifies the dispatching layer of all the system calls requested by each controlled process, i.e. each process running inside the partial virtual machine. The dispatching layer searches the hash table to see if there is a virtualization object that matches the system call request.

All system calls involving file descriptors (e.g. `read`, `close`, `lseek`, `fchown`) get assigned to whichever module handled the system call that returned the file descriptor in the first place. (e.g. `open`, `creat`, `socket` or `msocket`)

Table 2 is an example that shows the interactions between some commands and system calls run by user processes and the corresponding actions in the virtual machine monitor.

`um_add_service umfuse`	add `umfuse` to the global hash table as a new filesystemtype
`mount -t umfuseext2 diskimage /mnt`	`umfuseext2` matches `umfuse`, that module loads the submodule `umfuseext2` and registers the pathname `/mnt`
`fd=open("/mnt/f",O_RDONLY);`	`/mnt/f` matches `/mnt`, umfuse module handles the request.
`n=read(fd,buf,BUFSIZE);`	If fd is the descriptor returned above the request is dispatched to the umfuse module,

Table 2: Example illustrating interaction between system calls and View-OS monitor

Sometimes one needs to nest virtualizations, as in example 3.4, or when a filesystem image contains a file which is itself another filesystem image. In this latter case the user can mount the outer filesystem as in the example above. The problem arises when the user tries to mount the inner filesystem image. The filesystem image does not exist in the real file system, thus the approach illustrated above would fail. In this case, to support the inner mount, the system call to virtualize is not generated by the user process but by the virtual machine monitor itself.

⋆mview uses `purelibc`, a self virtualization library. (Also developed at the VirtualSquare Lab.) The implication of this is that all system calls generated by the virtual machine monitor can themselves refer to virtual entities. When this occurs, the system call is handled again by the virtual machine monitor itself. The view for a nested virtualization system call is the view of the process just before the creation of the virtualization object currently managed. This means that commands like:

```
$ mount -t umfuseext2 diskimage /mnt
$ mount -t umfuseext2 /mnt/2nd_image /mnt
```

are legal in ⋆mview.

When a user opens a file in `mnt`, say `/mnt/f`, the request matches both objects in the hash table, but the most recent is the second. The umfuse file system implementation will read/write the `/mnt/2nd_image` file. The most recent match, *prior to the second mount* is the object of the first mount. For the sake of completeness, the self virtualization actually operates a third time: for diskimage. If no matching objects exist in the hash table *older than the first mount*, the request is sent to the Linux kernel.

`purelibc` is a libc library that does not directly generate system calls. When using a C library, e.g. `glibc`, `eglibc` or `dietlibc`, both the library functions and corresponding system calls are in the same library. Hence when a function needs a system call, it invokes it directly. For example, `printf` invokes `write`. In purelibc, library functions invoke a system call interfacing function that can be overwritten. In this way ⋆mview and its modules can use both standard code and libraries, with purelibc providing self virtualization when needed. For example umfuseext2 uses the standard `libext2fs` library from the e2fsprogs project [11].

5 Some performance figures

In this section we present our results from some View-OS performance experiments. The figures in this section convey the degree of usability of the current View-OS implementations as well as indicate the direction of investigation to further improve the project.

The first set of experiments try to evaluate the cost of a "potential virtualization," i.e. the cost to run processes inside an *empty* View-OS monitor. (No specific virtualizations loaded beyond the base View-OS ability to support user requested virtualizations.)

TIME	kernel	umview	umview-n	kmview	UML-hostfs	umview-mov	umview-n mov	kmview mov
Average	27.51	66.11	88.63	63.82	122.12	88.28	124.97	86.68
Std. Dev.	3.481	0.950	2.138	2.273	2.528	1.062	1.867	0.520

Table 3: The "make vde-2" experiment

Table 3 compares the execution times of the command *make* to compile *vde-2*, a software package for providing virtual network support. The execution time of this command was evaluated in several different scenarios/configurations on a Intel Core2Duo machine @ 1Ghz, 2GB RAM, Linux 2.6.29, Debian SID.

1. Kernel - The command ran from a standard shell, no virtualization.

2. umview - The command ran from a shell running on a umview virtual machine. No virtualization modules are loaded. The host kernel contains support for ptrace-vm and ptrace-multi.

3. umview-n - The command ran from a shell running on a umview virtual machine. No virtualization modules are loaded. The host kernel is "vanilla," and therefore does not contain support for ptrace-vm or ptrace-multi.

4. kmview - The command ran from a shell running on a kmview virtual machine. No virtualization modules are loaded. The host kernel contains support for both the utrace and the kmview modules.

5. UML-hostfs - The command ran from a shell running on a User-Mode linux kernel. The source code was on a mounted *hostfs* partition.

6. umview-mov - Like umview above, only the source code to be compiled resided on a partition mounted by the viewfs module.

7. umview-n mov - Like umview-n above, only the source code to be compiled resided on a partition mounted by the viewfs module.

8. kmview mov - Like kmview above, only the source code to be compiled resided on a partition mounted by the viewfs module.

Not surprisingly, these results show that there is a computational cost related to supporting virtualization. The *make* experiment took over twice the time to run in View-OS (umview configuration) than on the bare kernel.

Except for one configuration (umview-mov), all $\star$mview configurations were faster than User-Mode Linux. When comparing View-OS performance to User-Mode Linux, one should only examine the $\star$mview "mov" configurations. Since User-Mode Linux is a standard virtual machine providing its own global view assumption domain, the simplest way to access the file system of the hosting machine is by using the `hostfs` feature. Examining $\star$mview performance to that of User-mode Linux with no virtualization modules loaded makes for an unfair comparison since under $\star$mview (none of the "mov" configurations) system calls to access data are directly evaluated by the kernel.

A more consistent comparison must have umview or kmview involved in accessing the data, i.e. the "mov" configurations where a filesystem hierarchy is mounted by `viewfs`

```
mount -t viewfs -o move / /mnt
```

Umview, when run on a kernel with no ptrace support exhibits approximately the same performance as User-Mode Linux. A 25-28% speedup was achieved by both umview and kmview when executing on a kernel with specific View-OS support.[3] The extra cost to support virtualizations on a vanilla kernel is due to the context switch needed for each memory word exchanged between the virtual machine monitor and the process.

TIME	kernel	umview	umview-n	kmview	UML-hostfs	umview-mov	umview-n mov	kmview mov
Average	6.16	6.4	6.74	6.09	6.57	6.53	38.93	6.59
Std. Dev.	0.080	0.128	0.329	0.084	0.137	0.283	0.690	0.072

Table 4: The compute cryptographic digest experiment

The table 4 uses the same configurations as above to evaluate the time needed to compute a cryptographic digest SHA 512bit for a 100MB file containing random data.

The context switch overhead for umview when running without kernel support/optimizations is clearly evident in this experiment as well. (The "umview-n mov" configuration.) All the performance values, except the one affected by this problem, are all very similar. The pattern of system calls generated by this experiment is that there were very few calls to start the process and open the file along with

[3]kmview can only run on a host kernel with View-OS support.

several `read` system calls. Comparing these results with those of the previous experiment we conclude that the major overhead in supporting virtualization is related to the management of pathnames. System calls operating on open files take about the same time to run in View-OS as on the bare kernel.

TIME	kernel	Fuse-ext2	Umview-umfuse	Kmview-umfuse	UML-mount
Average	2.41	8.33	8.26	7.91	6.63
Std. Dev.	1.519	0.465	0.451	0.286	0.567

Table 5: The copying of a 200MB file of random data by FUSE/umfuse

While previous experiments estimated the overhead cost of a *potential virtuality*, table 5 include results of an *actual virtuality*. In this experiment a 200MB file containing random data is copied into a virtually mounted ext2 filesystem, using the *umfuse* module and the *umfuseext2* submodule.

The configurations compared here are the following:

1. kernel - The command ran without any virtualization. The file system was mounted as a loopback filesystem by the kernel.
2. Fuse-ext2 - The file system was mounted by *fuse*.
3. umview-umfuse: the command ran in a umview session, the file system was mounted by *umfuse*.
4. kmview-umfuse -The command ran in a kmview session with the file system mounted by *umfuse*.
5. UML-mount - The command ran on a User-Mode Linux virtual machine which mounted the ext2 filesystem.

The first conclusion one draws from this experiment is that there is a significant overhead cost for virtualization. It is interesting to observe, however, that the time values are similar both when the virtualization engine is at the kernel level, as in the Fuse-ext2 configuration, or at the user level, as in either umview or kmview configuration.

Our final experiment, as with the cryptographic digest experiment (Table 4), involves more data access system calls using file descriptors, than system calls with pathname arguments.

TIME	kernel	Fuse-ext2	Umview-umfuse	Kmview-umfuse	UML-mount
Average	0.36	3.27	10.35	9.73	1.36
Std. Dev.	0.005	2.561	0.093	0.160	0.466

Table 6: The time to run `ls -lR | wc`

When the number of pathname based system calls increases, there was a significant slowdown. Table 6 evaluates the time needed to scan an entire filesystem tree. Here FUSE seems to benefit from kernel optimizations as well as data caching. The standard deviation indicates that the time values measured in different experiments were very different: repeated experiments always ran faster than the first.

6 Desiderata

$\star$mview are reference implementations of View-OS. During the development of umview and kmview several limitations were encountered. This section includes a summary of these limitations as well as the solutions implemented (e.g. `PTRACE_SYSVM`, `PTRACE-MULTI`, `msocket`,*purelibc*). Since these issues are not uniquely related to View-OS other projects could benefit from the solutions we devised in addition to the various fixes or extensions proposed. The title of this section is therefore *desiderata*. It is our hope that the various solutions engineered by the VirtualSquare Lab will not only be useful to others, but will find their way into the mainstream code bases of their respective projects.

6.1 Linux Kernel

Umview uses ptrace to track process generated system calls. Ptrace has some limitations.

1. PTRACE-SYSVM. Ptrace was designed for debugging, not for virtualization. While with ptrace it is possible to change any given system call, it is not possible to skip it - a desired feature when the virtualization layer handles the system call making the context switch to the kernel unnecessary. Originally, User-Mode Linux created a workaround by turning all virtualized system calls into harmless `getpid()`. Later Giarrusso (Blaisorblade)[2] introduced `PTRACE_SYSEMU` to skip the *next* system call. This excellent approach only works for complete virtual machines aiming to virtualize all system calls; it is of no benefit to partial virtual machine approaches like View-OS.

   The VirtualSquare Lab proposes a different tag `PTRACE_SYSVM` that can be used by both complete virtual machines (e.g. User-Mode Linux) and by partial virtual machines (e.g. umview). `PTRACE_SYSVM` permits one to run or skip the current system call; not the next one. Furthermore it allows one to decide if the virtual machine monitor needs to retrieve the system call result or not.

2. PTRACE-MULTI. The standard ptrace interface allows one to load and store data from/to the memory of the controlled process. Unfortunately, each ptrace call is able to transfer just one memory word; 4 or 8 bytes depending on the hardware architecture. As illustrated in section 5, this is a hard performance bottleneck for umview.

   In theory it is also possible to load and store large chunks of memory by opening `/proc/${pid}/mem`, where `${pid}` is the pid of the process. Unfortunately this latter solution has several drawbacks; primarily that it cannot be used for writing, since this would be considered a serious security hazard.[4]

   The VirtualSquare Lab proposes a new ptrace tag named `PTRACE-MULTI` that allows one to evaluate several ptrace requests in a single atomic system call. This would be analogous to how `readv` allows several `read` operations to run in a single call. `PTRACE-MULTI` would use an array of requests, with each data transfer request allowing for the transfer of several memory words. This approach would provide umview two kinds of speedup: Data transfers would now only require one request regardless of their size, and a sequence of ptrace calls would require only one user-kernel context switch. Moreover there would be no need to open and close `/proc/${pid}/mem` for each memory access or to keep as many open file descriptors as the number of processes, limiting the scalability.

   Unfortunately, this proposal/solution has been criticized as an example of *system call batching*. Furthermore, in spite of the performance values reported in section 5 some believe that the speedup gained would be negligible. Given the importance of performance to virtual system acceptance and usability, any implementation leading to a relaxing of the hard bottleneck imposed by ptrace is welcome.

[4]Presently in the kernel there is a hack for the `sparc` architecture where the tags `PTRACE_{READ,WRITE}{TEXT,DATA}` use an extra parameter (addr2) passed to the system call in a register – inconsistent with ptrace's signature. The presence of this hack implies that other developers faced this bottleneck and had a workaround hack accepted in the mainstream kernel.

6.2 POSIX/Open Group

View-OS demonstrates the feasibility of managing several network stacks simultaneously. The Berkeley sockets interface unfortunately was designed to support at most one stack implementation per protocol family. Both with `socket` and `socketpair` there is no way to specify which stack to use in case several are available for the same protocol family.

We propose two new but backward compatible system calls, `msocket` (and `msocketpair`)

```
int msocket(char * path, int domain, int type, int protocol);
```

.

This call has the same signature of `socket` except from a new first argument.

All available network stacks will appear in the file system as special files; maybe as a new type of special file. Programmers can decide which networking stack to use simply by using the stack's pathname in `msocket`.

Each process has a (re-definable) default stack for each protocol family. When path is NULL, or the process is using the old `socket` system call, the default stack is used.

This proposal has several advantages:

- It is possible to use multiple networking stacks simultaneously.
- `msocket` allows each process to use several stacks.
- It is backward compatible, and existing programs can use different stacks, one at a time.
- The file system becomes the global naming facility for networking as well - currently the one notable exception.
- It is very easy to understand and use (see section 3.2).

6.3 C library

The VirtualSquare Lab designed *purelibc* for process self virtualization: The system calls generated by a process can be traced and virtualized by a function defined in the same process.

This feature could have been implemented using the C library. Purelibc runs on `glibc` and `eglibc`. It redefines the system call interfacing functions as well as many stdio and directory management functions.

We posit that a (C) library should provide a way for it to run as a "pure" library which processes data and prepares the arguments for a system call without actually making the call.

Here are two different approaches: The feature can be added to the C library itself or the pure library can be implemented independently, sharing a large amount of code with the system call making ("impure") library. In the former case all processes (those needing the "pure" function variation and those needing the actual system calls being executed) would use the same shared library. In the latter case, while the two libraries get loaded independently, the performance of the C library is completely preserved.

`purelibc` is currently implemented as a library to be preloaded (`LD_PRELOAD`) on the standard C library. Though this approach is quite fast, there are some limitations. `freopen` is consistent to its specifications, but only for standard files (input, output or error); it can return a different file descriptor otherwise. `dlopen` simply cannot be virtualized.

6.4 Utrace

Kmview uses utrace to track process generated system calls. Utrace too has some limitations with regard to a View-OS implementation.

1. Management of UTRACE_STOP. When a tracing engine returns `UTRACE_STOP` the process is suspended until it is resumed (by `utrace_control(...,UTRACE_RESUME)`). Since several utrace engines can trace the same process, when one utrace engine returns a `UTRACE_STOP`, the process is not suspended until after all the the other utrace engines tracing the process have received the system call notification. This significantly complicated the management of nested virtualizations in kmview.

   The kmview kernel module registers a utrace engine on behalf of the processes running in the virtual machine. When one starts a kmview machine as a process of an existing kmview machine, the processes running in the *nested* virtual machine have two utrace engines registered for them.

   Normally, when kmview manages a system call request it suspends the calling process until its completion; maybe parameters need altering before passing up the call, or the call itself is virtualized. When a process running in a nested kmview machine makes a system call request, the *outer* machine must receive the system call as modified by the *inner* machine, not the original request of the process. This means that utrace must wait for the changes made by the inner machine to complete before forwarding the system call request to the outer machine, hence `UTRACE_STOP` cannot be used to suspend the calling process. The workaround implemented in the current kmview implementation is to block the calling process on a semaphore inside the system call notification function.

   Kmview also supports the strace utility. (Strace cannot be used in umview because nested ptrace'ing is not allowed.) Unfortunately due to the `UTRACE_STOP` management policy, strace erroneously reports the system calls as seen by the real kernel.

2. Management of UTRACE_KILL. The use of the notification functions `send_sig` or `force_sig` may interfere with the internal status of processes maintained by utrace. A new return tag `UTRACE_KILL` (similarly to `PTRACE_KILL` for ptrace) would ease the writing of utrace modules: utrace can terminate the process while keeping the consistency of its state.

7 Conclusion

Adherence to the *global view assumption* forces the Linux kernel (and more generally, the POSIX standard) to impose a number of strict limitations on (user) processes. Some of these limitations are due to security considerations while others are simply the shortcomings of what is possible under the *global view assumption*. UnionFS, fakeroot, chroot, fuse, VPN, binfmt are just some of the tools created to overcome some of these limitations. View-OS, through its unified model of virtualization, provides a novel, integrated, and modular approach to overcoming these same limitations.

Complete (or system level) virtual machines like User-Mode Linux, Qemu, and Kvm can also be used to overcome these limitations; security considerations and the *global view assumption*. Using these tools a user can mount filesystems, provide multiple networking stacks (one per virtual machine), create user-level sandboxes, etc. Complete virtual machines require the booting of an entire kernel and the allocation of a large chuck of memory to emulate the central memory of the target machine. Their kernels, since they are complete, contain a scheduler, a memory manager along with other components that require memory and processing power. The goal of these virtual machines is the complete emulation of a computing system, and therefore provide for very inefficient solutions to the *global view assumption* problem.

View-OS is unique in that like complete virtual machines (e.g. User-Mode Linux), it provides for a unified solution; it is a single tool which through the provided virtualizations not only eliminates the *global view assumption*, but also addresses possible security concerns by running completely in user-mode. Furthermore, like UnionFS, fakeroot, chroot, fuse, VPN, binfmt, etc, View-OS is lightweight. Given its modular design, one need only load the specific modules needed to support the desired virtualizations; hence its memory footprint is only as big as is needed. But unlike these separate tools, which often do not interoperate, View-OS is designed to allow a user to combine different virtualizations together to solve new and interesting problems. In this regard, View-OS is unlike any other similar tool that we know of.

One unintended advantage to View-OS's modular approach and concomitant "only as big as you need" memory footprint is that View-OS can be used to provide partial virtualizations on netbooks or other small systems. Any one who has ever tried to boot User-Mode Linux on an underpowered netbook (relatively slow processor, slow disk drive and small quantity of installed RAM) will certainly appreciate this.

There are several similarities between View-OS modules and microkernel servers. In a pure microkernel design, file system or networking stack modules are user-level servers. These servers provide the abstractions which are usually implemented inside a monolithic kernel. It is also possible to write device-driver modules, provided the kernel grants access to the raw device.

View-OS captures some of the benefits of microkernels such as flexibility and global system reliability, while retaining backward compatibility with the existing kernels. A system administrator could elect to provide some new or experimental services via View-OS while keeping the traditional direct monolithic kernel support for all other services. Hence by using View-OS one can overcome the primary roadblocks for microkernel implementation; hardware or software compatibility such as the lack of device drivers or specific servers. Here View-OS acts as a kind of microkernel dispatch engine, but only for the specific services/devices. View-OS is, in some sense, the missing ring between monolithic kernels and microkernels, providing a way for these different approaches to coexist on the same system.

Finally, `umview` and `kmview` are *not* final, complete, perfect implementations of View-OS. They are a starting point, not a final result. However `umview` and `kmview`, even at this stage of development, clearly reveal their effectiveness and the wide range of possible applications.

References

[1] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, 2008.

[2] Paolo Giarrusso (blaisorblade). System call emulation patch. `http://www.user-mode-linux.org/~blaisorblade/faq.html`.

[3] Renzo Davoli. Vde: Virtual distributed ethernet. *Testbeds and Research Infrastructures for the Development of Networks & Communities, International Conference on*, 0:213–220, 2005.

[4] Renzo Davoli, Michael Goldweber, et al. Virtual square lab wiki. `http://wiki.virtualsquare.org/`.

[5] Michael Goldweber and Renzo Davoli. Vde: an emulation environment for supporting computer networking courses. *SIGCSE Bull.*, 40(3):138–142, 2008.

[6] Roland McGrath. utrace. `http://people.redhat.com/roland/utrace/`.

[7] Simon Peter. Klik. `http://klik.atekon.de/`.

[8] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proc. of LISA '04: 18th Systems Administration Conf.*, pages 241–254, November 2004. Atlanta, GA.

[9] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 275–287, New York, NY, USA, 2007. ACM.

[10] Miklos Szeredi. FUSE: filesystem in user space. `http://fuse.sourceforge.net`.

[11] Theodore Ts'o. E2fsprogs: Ext2/3/4 filesystem utilities. `http://e2fsprogs.sourceforge.net/`.

[12] Joost Witteveen, Clint Adams, and Timo Savola. The fakeroot utility. `http://packages.debian.org/stable/utils/fakeroot.html`.