




































- 1. [Introduction](#) 
- 1.1 [Who needs that](#) 
- 2. [Principles](#) 
- 2.1 [Non reversible isolation](#) 
- 2.2 [Isolation areas](#) 
- 2.3 [New system calls](#) 
- 2.4 [Limiting super-user: The capabilities system](#) 
- 2.5 [Enhancing the capability system](#) 
- 2.6 [Playing with the new system calls](#) 
- 2.6.1 [Playing with /usr/sbin/chcontext](#) 
- 2.6.2 [Playing with /usr/sbin/chcontext as root](#) 
- 2.6.3 [Playing with /usr/sbin/chbind](#) 
- 2.6.4 [Playing with /usr/sbin/reducecap](#) 
- 2.7 [Unification](#) 
- 3. [Applications](#) 
- 3.1 [Virtual server](#) 
- 3.2 [Per user fire-wall](#) 
- 3.3 [Secure server/Intrusion detection](#) 
- 3.4 [Fail over servers](#) 
- 4. [Installation](#) 
- 4.1 [The packages](#) 
- 4.2 [Setting a virtual server](#) 
- 4.3 [Basic configuration of the virtual server](#) 
- 4.4 [Entering the virtual server](#) 
- 4.5 [Configuring the services](#) 
- 4.6 [Starting/Stopping the virtual server](#) 
- 4.7 [Starting/Stopping all the virtual servers](#) 
- 4.8 [Restarting a virtual server from inside](#) 
- 4.9 [Executing tasks at vserver start/stop time](#) 
- 4.10 [Issues](#) 
- 4.11 [How real is it ?](#) 
- 5. [Features](#) 
- 6. [Future directions](#) 
- 6.1 [User controlled security box](#) 
- 6.2 [Kernel enhancements](#) 

Virtual private servers and security contexts

Running independent Linux servers inside a single PC is now possible. They offer many advantages, including higher security, flexibility and cost reduction.

NEW

Introduction

Linux computers are getting faster every day. So we should probably end up with less, more powerful servers. Instead we are seeing more and more servers. While there are many reasons for this trend (more services offered), the major issue is more related to security and administrative concerns.

Is it possible to split a Linux server into virtual ones with as much isolation as possible between each one, looking like real servers, yet sharing some common tasks (monitoring, backup, ups, hardware configuration, ...) ?

We think so ... NEW

Who needs that

The short answer is **everybody**, or everybody managing a server. Here are some applications:















- Hosting: Complete general purpose hosting (Running many independent servers in one box).
- Experimentation: You are toying with a new services and do not want to impact the production services on the same machine.
- Education: Each student has its own server with root password.
- Personal security box: Run un-trusted applications with complete control over their interaction with the rest of the computer and the network.
- Managing several "versions" of the same server/project and turning on/off each version independantly.

Just think about all the viruses and worms out there, you end up with a big **everybody using a computer needs this**. :-) NEW

Principles

Amazon Ex. 1007

IPR Petition - USP 7,519,814

- 6.2.1 [Per context disk quota](#) 
- 6.2.2 [Global limits](#) 
- 6.2.3 [Scheduler](#) 
- 6.2.4 [Security issues](#) 
- 6.2.4.1 [/dev/random](#) 
- 6.2.4.2 [/dev/pts](#) 
- 6.2.4.3 [Network devices](#) 
- 7. [Alternative technologies](#) 
- 7.1 [Virtual machines](#) 
- 7.2 [Partitioning](#) 
- 7.3 [Limitation of those technologies](#) 
- 8. [Conclusion](#) 
- 9. [Download](#) 
- 10. [References](#) 

NEW

Non reversible isolation

Unix and Linux have always had the `chroot()` system call. This call was used to trap a process into a sub-directory. After the system-call, the process is led to believe that the sub-directory is now the root directory. This system call can't be reversed. In fact, the only thing a process can do is trap itself further and further in the file-system (calling `chroot()` again).

The strategy is to introduce new system calls trapping the processes in other areas within the server. NEW

Isolation areas

A virtual server is isolated from the rest of the server in 5 areas:

- File system

The vserver is trapped into a sub-directory of the main server and can't escape. This is done by the standard `chroot()` system call found on all Unix and Linux boxes.

- Processes

The vserver can only see the processes in the same security context. Even the root server can't see the processes in vservers, making the root server less "dangerous" to use. A special mechanism (context number 1) exists to view all processes though (Limited to root in the root server).

- Networking

The vserver is assigned a host name and an IP number. The server can only use this IP number to establish services and client connection. Further, this restriction is transparent.

- Super user capabilities

The super user running in a vserver has less privileges than the normal Linux root user. For example, it can't reconfigure the networking and many aspect of the system. It can't mount devices, can't access block devices and so on.

Roughly, the vserver super-user has full control over all files and processes in the vserver and that's pretty much it.

- System V inter process communications

Sysv IPC resources are private to each vserver. The security context is used as an extra key to select and assign resources.

Those facilities are used together to create a runtime environment for virtual servers. But they can be used independently to achieve various goals. NEW

New system calls

The new system calls, as well as the existing `chroot()` system call are sharing one common feature: Their effect can't be reversed. Once you have executed one of those system call (`chroot`, `new_s_context`, `set_ipv4root`), you can't get back. This affects the current process and all the child processes. The parent process is not influenced.

- `new_s_context` (int ctx)

This system call sets a new security context for the current process. It will be inherited by all child processes. The security context is just an id, but the system call makes sure a new unused one is allocated.

A process can only see other processes sharing the same security context. When the system boot, the original security context is 0. But this one is not privileged in anyway. Processes member of the security context 0 can only interact (and see) processes member of context 0.

This system call isolates the processes space.

- Setting the capabilities ceiling

This is handle by the `new_s_context` system call as well. This reduces the ceiling capabilities of the current process. Even `setuid` sub-process can't grab more capabilities. The capability system found since Linux 2.2 is explained later in this document.

- `set_ipv4root`(unsigned long ip)

This system call locks the process (and children) into using a single IP when they communicate and when they installs a service. This system call is a one shot. Once a process have set its IPV4 (Internet Protocol Version 4) address to something different from 0.0.0.0, it can't change it anymore. Children can't change it either.

If a process tries to bind a specific IP number, it will succeed only if this corresponds to the `ipv4root` (if different from 0.0.0.0). If the process bind to any address, it will get the `ipv4root`.

Basically, once a process is locked to a given `ipv4root` it is forced to use this IP address to establish a service and communicate. The restriction on services is handy: Most service (Web servers, SQL servers) are binding to address 0.0.0.0. With the `ipv4root` sets to a given IP you can have two virtual servers using the exact same general/vanilla configuration for a given services and running without any conflict.

This system calls isolate the IP network space.

Those system calls are not privileged. Any user may issue them. NEW

Limiting super-user: The capabilities system

Once you have created a virtual environment where processes have a limited view of the file-system, can't see processes outside of their world and can only use a single IP number, you still must limit the damages those processes can do. The goal is to run virtual environments and provide some **root** privileges.

How do you limit those **root processes** from taking over the system, or even just re-booting it. Enter the capability system. This is not new, but we suspect many people have never heard of it.

In the old Unix/Linux days, user root (user ID 0) could do things other user ID could not. All over the place in the kernel, system calls were denying access to some resources unless the user ID of the process (effective ID in fact) was 0. Plain zero.

The only way a process with user ID 0 could loose some privileges was by changing to another ID. Unfortunately this was an all or nothing deal. Enter the capabilities.

Today, the difference between root and the other users is the capability set. User root has all capabilities and the other users have none. The user ID 0 does not mean anything special anymore. There are around 30 capabilities defined currently. A process may request to loose a capability forever. It won't be able to get it back.

Capabilities allows a root process to diminish its power. This is exactly what we need to create **custom super-user**. A super-user process in a virtual server would have some privileges such as binding port below 1024, but would not be able to reconfigure the network or reboot the machine. Check the file `/usr/include/linux/capability.h` to learn which one are available.

Note that the new system calls (`new_s_context` and `set_ipv4root`) are not controlled by capabilities. They are by nature irreversible. Once a virtual server is trapped in a `chroot/s_context/ipv4root` box, it can't escape from the parameters of this trap.

NEW

Enhancing the capability system

The Linux capability system, is still a work in progress. At some point, we expect to see capabilities attached to programs, generalizing the setuid concept. A setuid program would become a program with all capability granted.

For now, this is not available. As explained above a process may request to loose capabilities and its child process will be trapped with a smaller capability set.

Well, ..., it does not work that way. Unfortunately, until capabilities could be assigned to program, we still need a way to get back capabilities even in a child process. So the irreversible logic of the capabilities is kind of short circuited in the kernel.

To solve this, we have introduced a new per-process capability ceiling (cap_bset). This one represents the capability set inherited by child process, including setuid root child process. Lowering this ceiling is irreversible for a process and all its child.

This ceiling is handled by the new `_s_context` system call and the `reducecap` and `chcontext` utilities (part of the `vserver` package).

Using this, we can setup a virtual server environment where root has less capabilities, so can't reconfigure the main server.

NEW

Playing with the new system calls

The `vserver` package provides 3 utilities to make use of the new system calls. We will describe shortly how they work and provide few example. We invite the reader to try those example, so it has a better feel and trust.

After re-booting with a kernel implementing the new system calls, and installing the `vserver` package, one is ready to do experiment. You do not need to be root to test those new utilities. None of them is setuid either. NEW

Playing with `/usr/sbin/chcontext`

The `/usr/sbin/chcontext` utility is used to enter into a new security context. The utility switch the security context and execute a program, specified on the command line. This program is now isolated and can't see the other processes running on the server.

The experiment with this, start two command windows (xterm), as the same user ID. In each window execute the following commands:

```
xterm
```

Using chcontext: first window

```
/usr/sbin/chcontext /bin/sh  
pstree  
killall xterm  
exit
```

Using chcontext: second window

In the first window, you start the xterm command (or any command you like). In the second window you execute chcontext. This starts a new shell. You execute pstree and see very little. You attempt to kill the xterm and you fail. You exit this shell and you are back seeing all processes.

Here is another example. You switch context and you get a new shell. In this shell you start an xterm. Then you switch context again and start another sub-shell. Now the sub-shell is again isolated.

```
/usr/sbin/chcontext /bin/sh  
pstree  
xterm &  
pstree  
# Ok now you see your xterm  
/usr/sbin/chcontext /bin/sh  
pstree  
# the xterm is not there, killall will fail  
killall xterm  
# Now we exit the shell  
exit  
pstree  
# the xterm is there  
killall xterm  
# Ok, it is gone  
exit  
# We are back to the initial security context
```

Using chcontext several times

Processes isolated using chcontext are doubly isolated: They can't see the other processes on the server, but the other processes can't see them either. The original security context (0) when you boot is no better than the other: It sees only process in security context 0.

While playing with `chcontext`, you will notice an exception. The process 1 is visible from every security context. It is visible to please utilities like `ps`. But only root processes in security context 0 are allowed to interact with it.
NEW

Playing with `/usr/sbin/chcontext` as root

The `new_s_context` system call has a special semantic for root processes running in security context 0 and having the `CAP_SYS_ADMIN` capability: They can switch to any context they want.

Normally, `new_s_context` allocates a new security context by selecting an unused one. It walks all processes and find an ID (an integer) not currently in use.

But root in security context 0 is allowed to select the context it wants. This allow the main server to control the virtual server. The `chcontext` utility has the `--ctx` option to specify the context ID you want.

To help manage several virtual server, given that the security context 0 can't see processes in other security context, it is a good thing root in the main server (security context 0) is allowed to select a specific context. Cool. But we also need a way to have a global picture showing all processes in all security context. The security context 1 is reserved for this. Security context 1 is allowed to see all processes on the server but is not allowed to interact with them (kill them).

This special feature was allocated to security context 1 and not 0 (the default when you boot) to isolate virtual servers from the main. This way, while maintaining services on the main server, you won't kill service in vservers accidentally.

Here is an example showing those concepts:

```
# You must be root, running X
# We start an xterm in another security context
/usr/sbin/chcontext xterm &
# We check, there is no xterm running, yet we can
# see it.
ps ax | grep xterm
# Are we running in security context 0
# We check the s_context line in /proc/self/status
cat /proc/self/status
# Ok we in security context 0
# Try the security context 1
/usr/sbin/chcontext --ctx 1 ps ax | grep xterm
# Ok, we see the xterm, we try to kill it
/usr/sbin/chcontext --ctx 1 killall xterm
```

```
# No, security context 1 can see, but can't kill
# let's find out in which security context this
# xterm is running
/usr/sbin/chcontext --ctx 1 ps ax | grep xterm
# Ok, this is PID XX. We need the security context
/usr/sbin/chcontext --ctx 1 cat /proc/XX/status
# We see the s_context, this is SS.
# We want to kill this process
/usr/sbin/chcontext --ctx SS killall xterm
```

chcontext as root

The `/usr/sbin/vpstree` and `/usr/sbin/vps` commands are supplied by the `vserver` package. They simply runs `ps` and `pstree` in security context 1. NEW

Playing with /usr/sbin/chbind

The `chbind` utility is used to lock a process and its children into using a specific IP number. This applies to services and client connection as well. Here are few examples. Execute them as root:

```
# httpd is running
netstat -atn | grep ":80 "
# We see a line like this
# tcp    0  0 0.0.0.0:80  0.0.0.0:*      LISTEN
# Now we restart httpd, but we lock it so it
# uses only the main IP of eth0
/usr/sbin/chbind --ip eth0 /etc/rc.d/init.d/httpd restart
netstat -atn | grep ":80 "
# Now we see a line like
# tcp    0  192.168.1.1:80  0.0.0.0:*      LISTEN
# httpd.conf was not changed.
# Now, restart it normally
/etc/rc.d/init.d/httpd restart
# Now we experiment with client socket
# Log using telnet
telnet localhost
# Once logged in
netstat -atn | grep ":23 "
# You should see a line showing a connection from
# 127.0.0.1 to 127.0.0.1 like this
# tcp    0  0 127.0.0.1:23  127.0.0.1:32881  ESTABLISHED
exit
# Now we do the telnet again, but forcing it to select a specific IP
/usr/sbin/chbind --ip eth0 telnet localhost
# Log in and then execute
netstat -atn | grep ":23 "
```

```
# You will get something like
# tcp 0 0 127.0.0.1:23 192.168.3.9:32883 ESTABLISHED
```

Using /usr/sbin/chbind

NEW

Playing with /usr/sbin/reducecap

The reducecap utility is used to lower the capability ceiling of a process and child process. Even **setuid** program won't be able to grab more capabilities.

```
# You are not root now
# What is the current capability ceiling
cat /proc/self/status
# The capBset line presents mostly 1s.
/usr/sbin/reducecap --secure /bin/sh
cat /proc/self/status
# The capBset now shows many more 0s.
# The capEff shows all 0s, you have no privilege now
# We su to root
su
cat /proc/self/status
# capEff is much better now, but there are still many 0s
# Now we try to see if we are really root
tail /var/log/messages
# So far so good, we see the content
/sbin/ifconfig eth0
/sbin/ifconfig eth0 down
# No way, we can't configure the interface. In fact
# we have lost most privilege normally assigned to root
exit
```

Using /usr/sbin/reducecap

NEW

Unification

Installing a virtual private server copies a linux installation inside a sub-directory. It is a linux inside linux. If you intend to run several vservers on the same box (which you will certainly do :-), you will end up using a lot of disk space needlessly: Each vserver is made up hundreds of megabytes of the same stuff. This is a big waste of disk space.

A solution is to use hard links to connect together common files. Using the package information, we can tell which packages are shared between various vservers, which files are configuration files and which are not (binaries, libraries, resource files, ...). Non configuration files may be linked together saving a huge amount of disk space: A 2 GIG rh7.2 installation shrinks to 38megs.

Using hard links is cool, but may be a problem. If one vserver overwrite one file, say /bin/ls, then every vserver will inherit that change. Not fun! The solution is to set the **immutable** bit on every linked file. A file with such a bit on can't be modified, even by root. The only way to modify it is to turn off the bit first. But within a vserver environment, even root is not allowed to perform this task. So linked file, turned immutable, are now safe: They can be shared between vservers without side effects: Cool!

Well, there is still another side effect. All vservers are now locked with the same files. We are saving a lot of disk space, but we pay a very heavy price for that: Vservers can't evolve independantly.

A solution was found. A new bit call **immutable-linkage-invert** was added. Combined with the immutable bit, a file may not be modified, but may be unlinked. Unlinking a file in Unix/Linux means disconnecting the name from the data and removing the name from the directory. If the data is still referenced by one or more vservers, it continue to exist on disk. So doing "rm /bin/ls" on a vserver, removes the file /bin/ls for that vserver and that's all. If all vservers perform this task, then /bin/ls (the data) will be forgotten completely and the disk space will be recovered.

Using this trick, a vserver gets back its independance. It becomes possible to update packages by using the unlink/update sequence: Unlink /bin/ls first and then put a new copy in place. Luckily, package manager works this way.

To keep this story short (probably too late :-) , a unified vserver:

- Uses little disk space
- Can't interfere with other vservers by changing one of its file.
- Can perform package update/deletion normally using standard practice.

NEW

Applications

NEW

Virtual server

The first goal of this project is to create virtual servers sharing the same machine. A virtual server operate like a normal Linux server. It runs normal services such as telnet, mail servers, web servers, SQL servers. In most cases, the services run using standard configuration: The services are running unaware of the virtual server concept.

Normal system administration is performed with ordinary admin tool. Virtual servers have users account and a root account.

Packages are installed using standard packages (RPMs for example).

There are few exceptions. Some configuration can't be done inside a virtual server. Notably the network configuration and the file-system (mount/umount) can't be performed from a virtual server.

NEW

Per user fire-wall

The `set_ipv4root()` system call may be used to differentiate the various users running on an application server. If you want to setup a fire-wall limiting what each user is doing, you have to assign one IP per user, even if they are running application on the same server. The **chbind** utility may be used to achieve that. NEW

Secure server/Intrusion detection

While it can be interesting to run several virtual servers in one box, there is one concept potentially more generally useful. Imagine a physical server running a single virtual server. The goal is isolate the main environment from any service, any network. You boot in the main environment, start very few services and then continue in the virtual server. The service in the main environment could be

- Un-reachable from the network.
- The system log daemon. While virtual server could log messages, they would be unable to change/erase the logs. So even a cracked virtual server would not be able the edit the log.
- Some intrusion detection facilities, potentially spying the state of the virtual server. For example tripwire could run there and it would be impossible to circumvent its operation or trick it.

NEW

Fail over servers

One key feature of a virtual server is the independence from the actual hardware. Most hardware issues are irrelevant for the virtual server installation. For example:

- Disk layout, partitions and the `/etc/fstab` configuration. The virtual server has a dummy `/etc/fstab`.
- Network devices.
- Processor type, number of processor (kernel selection).

The main server acts as a host and takes care of all those details. The virtual server is just a client and ignores all the details. As such, the client can be moved to another physical server with very few manipulations. For example, to move the virtual server `v1` from one physical one computer to another, you do

- Turn it off
`/usr/sbin/vserver v1 stop`
- Copy the file `/etc/vservers/v1.conf` to the new server.
- Copy all files in `/vservers/v1` to the new server
- On the new server, start the `vserver v1`
`/usr/sbin/vserver v1 start`

As you see, there is no adjustment to do:

- No user account merging.
- No hardware configuration to fix.

This opens the door to fail over servers. Imagine a backup server having a copy of many virtual servers. It can take over their tasks with a single command. Various options exist for managing this backup server:

- `rsync` to synchronize the data.
- Network block devices to synchronize the data in real time.
- Sharing the installation over a SAN (storage area network).
- Heartbeat for automatic monitoring/fail over.

NEW

Installation

NEW

The packages

- The kernel

We are supplying a patched 2.4.18 kernel. You will find [here](#) the kernel, the .config and the patch.

To install the kernel, just untar it. This will create a file /boot/kernel-2.4.18ctx-10 (ctx stands for security context) and a directory /lib/modules/2.4.18ctx-10.

Then, you need to update your boot loader. For **lilo**, you add a section like this at the end of the file **/etc/lilo.conf**

```
image=/boot/vmlinuz-2.4.18ctx-10
    label=linux2.4.18ctx-10
    read-only
    root=current
```

lilo.conf section to add

Change the /dev/XXXX to your root partition. Then execute **/sbin/lilo**.

Reboot and select the proper kernel. This kernel is fully compatible with a normal 2.4.18 and will perform without any new configuration. Note that the supplied kernel does not carry all the features and modules found on the various distributions.

- The vserver package

This package provides the various utilities to make use of those new system calls. The package also provides a complete solution to implement virtual servers. We describe the major components here.

- /usr/sbin/chcontext

This is the utility to request a new security context. It can be used to lower the capability ceiling. Execute it to learn more.

- /usr/sbin/chbind

This is the utility to select one IP number and assign it to a process and its children.

- /usr/sbin/newvserver (in vserver-admin)

Front-end to help create new virtual servers.

- o /usr/sbin/reducecap

This utility is used to lower the capability ceiling of children processes.

- o /usr/sbin/vdu

A trimmed down "du" command reporting space usage of files with a single link. Useful to tell how much space a unified vserver is using.

- o /usr/sbin/vkill

Locate the security context associated with a process, enter it and kill the process. Generally used after you have located a process with vtop, vps tree or vps.

- o /usr/sbin/vps

Execute the **ps** command in security context 1 so all processes in all vservers are shown. The security context and vserver name are mapped inside the report.

- o /usr/sbin/vps tree

Execute the **ps tree** command in security context 1 so all processes in all vservers are shown.

- o /usr/sbin/vrpm

Apply an **rpm** command in several (or all) vservers. Useful when you wish to update many vservers with the same package.

```
/usr/sbin/vrpm server1 server2 -- -Uvh /tmp/*.rpm
```

- o /usr/sbin/vserver

This is the wrapper to start, stop and administer virtual servers.

- o /usr/sbin/vserver-stat

Produce a small report showing the activity in active security context. The report presents the number of process in each active security context as well as the name of the vserver associated with this context.

- o /usr/sbin/vtop

Execute the **top** command in security context 1 so all processes in all vservers are shown.

- o /etc/rc.d/init.d/vservers

This is an init script used to start all virtual servers at boot time and stop them at shutdown time. Only virtual servers with ONBOOT=yes are started at boot time. All vservers are stopped at shutdown time.

- o /etc/rc.d/init.d/rebootmgr

This is a daemon listening to requests from virtual servers. It can either restart or stop a virtual server. The /sbin/vreboot and /sbin/vhalt utilities are used to send request to the reboot manager.

- o /sbin/vreboot and /sbin/vhalt

Those utilities are copied in each virtual server. They connect to the reboot manager (rebootmgr) server using the /dev/reboot Unix domain socket and request either a restart or a stop of the virtual server. The reboot manager issue either a "/usr/sbin/vserver vserver restart" or "/usr/sbin/vserver vserver stop" command. This allows the virtual server administrator to test if all automatic service are properly restarted at boot time.

- o /usr/lib/vserver/vdu

This is a limited clone of the **du** command. It skips file with more than one link. It is used to evaluate the disk usage of an unified vserver. Using the normal du for this task is misleading since it will count all unified files.

NEW

Setting a virtual server

To set a virtual server, you need to copy in a sub-directory a Linux installation. One way to achieve that is to copy some parts of the the current server by issuing the command **vserver XX build**, where XX is the name of the virtual server (pick one). This basically does (Well, it does a little more than that, but this give you an idea):

```
mkdir /vservers/XX
cd /vservers/XX
cp -ax /sbin /bin /etc /usr /var /dev /lib .
mkdir proc tmp home
chmod 1777 tmp
```

Building a virtual server

This is normally done using the command `/usr/sbin/newvserver`. This is a text mode/graphical front-end allowing to setup the vserver runtime and configure it. NEW

Basic configuration of the virtual server

A virtual private server has a few settings. They are defined in the file `/etc/vservers/XX.conf` where XX is the name of the virtual server. This is a simple script like configuration. Here are the various parameters:

- IPROOT

In general, each virtual server is tied to one IP using the new `set_ipv4root` system call. This way several servers may run the same services without conflict. Enter the IP number (a name is also valid if defined in the DNS).

- IPROOTDEV

This is the network device use to setup the IP alias defined by IPROOT. This is generally `eth0`. If you define this variable, the IP alias will be configured when you start the virtual server and un-configure when you stop it.

- IPROOTMASK

Netmask used to setup the IP alias. Uses the netmask of the IPROOTDEV device by default. Seldom used.

- IPROOTBCAST

Broadcast address used to setup the IP alias. Uses the broadcast of the IPROOTDEV device by default. Seldom used.

- ONBOOT

The vserver package supplies the vservers service. This service is installed in the main server. It is used to start and stop the virtual server at boot and shutdown time.

Virtual server with **ONBOOT=yes** will be started and stopped properly like any other services of the main server.

Once a virtual server is properly configured, it is a good idea to set this parameter to **yes**.

- S_CAPS

You enter here the various capability you want to grant to the vserver. By default, a vserver is left with much less capabilities than the root server. For example, a vserver is not allowed to use raw socket. This explains why the ping command fails. S_CAPS lets you enumerate the capability you want to keep in the vserver. CAP_NET_RAW will give back ping ability for example.

- S_CONTEXT

This is optional. In general the security context ID is selected by the kernel. An unused one is selected. If you select an ID (an integer greater than 1), make sure you select a different one for each server. Again, in most cases, you do not need to select one. Leave the line commented.

- S_DOMAINNAME

A virtual server may have a different NIS domainname than the main server. You set it here.

- S_HOSTNAME

Many services (Apache for one) use the host name to setup some defaults. A virtual server may have a different host name than the main server. It is a good idea to fill this line.

- S_NICE

This is an optional priority level. It is an integer ranging between from -20 to 19. Well it is the **nice** level in fact, so -20 means the highest priority and 19 the lowest (the highest nice level). All processes in the virtual server will be locked to this level (or higher niceness).

Event root is locked and can't get more priority.

Note that this has limited usefulness. The kernel does not differentiate processes running in different security context for scheduling (for now :-). This means that a virtual servers running many low priority processes may nevertheless claim a large share of CPU.

- S_FLAGS

This is used to set various flags. Here are the supported flags:

- lock

This flag prevents the vserver from setting new security contexts.

- sched

It kind of unifies the processes in a vserver from a scheduler view point. All processes are weighted as

single one when compared to other processes in the real server. This prevents a vserver from taking too much CPU resources.

- o nproc

Make the ulimit maximum user process global to the vserver.

- o private

Once set on a vserver security context, no other process can enter it. Even the root server is unable to enter the context. It can see the process list using security context 1, but can't signal or trace the process.

- o fakeinit

This assigned the current process so it works like the process number 1. Using this trick, a normal /sbin/init may be run in a vserver. The **/usr/sbin/vserver** command will use /sbin/init to start and stop a vserver. A properly configured **/etc/inittab** is needed though.

- Processes losing their parent are reparent to this process.
- getppid() done by child process of this process returns 1.
- getpid() done by this process returns 1.
- This process is not shown in /proc since process number 1 is always shown.
- An "initpid" entry is available in /proc/*/status to tell which process is the fake init process.

- ULIMIT

This contains the command line option to the ulimit shell command. You enter here whatever parameters accepted by ulimit. Here is the default when you create a new vserver:

```
ULIMIT="-H -u 1000"
```

Default vserver ulimit

Normally ulimit settings only affects users independently. So limiting a vserver this way, limit each user processes independently in the vserver. Using special flags in the S_FLAGS option, you can make those ulimit settings global to the vserver. The example above used with the **nproc** parameter make the maximum number of process global. In this case, a maximum of 1000 processes is available to all users in the vserver.

NEW

Entering the virtual server

It is possible to enter a virtual server context from the main server just by executing **/usr/sbin/vserver XX enter** (where XX is the virtual server name).

This creates a shell. From there you can execute anything administrative you normally do on a Linux server.

NEW

Configuring the services

The virtual server can run pretty much any services. Many pseudo services, such as network configuration are useless (the server is already configured). After building the environment, enter it (without starting the virtual server) using the **vserver name enter** command. Then using a tool like Linuxconf (control/control service activity) , or ntsysv, browse all service and keep only the needed ones.

So after building the server, you enter it and you select the service you need in that server. Many services such as **network**, and **apmd** are either useless or won't run at all in the virtual server. They will fail to start completely. NEW

Starting/Stopping the virtual server

Virtual server with **ONBOOT=yes** will be started and stopped like any other services of the main server. But you can stop and start a virtual server at any time. Starting a server means that all configured service will be started. Stopping it means that all configured services will be stopped and then all remaining process will be killed.

Oddly, starting a virtual server does not mean much. There is no overhead. No monitoring process or proxy or emulator. Starting a virtual server with 4 services is the same as running those 4 services in the main server, at least performance wise (the service inside a virtual server are locked inside the security context).

The following commands may be used to control a virtual server:

- /usr/sbin/vserver server start
- /usr/sbin/vserver server stop
- /usr/sbin/vserver server restart
- /usr/sbin/vserver server running
- /usr/sbin/vserver server enter
- /usr/sbin/vserver server exec some commands ...
- /usr/sbin/vserver server service service-name start/stop/restart/status

The **running** command prints if there are any processes running in the virtual server context.

Please note

The processes running in a virtual server are invisible from the main server. The opposite is true. This is very important. Managing the main server must not cause problems to the various virtual servers. For example, doing **killall httpd** will kill all the httpd processes in the current context (the main server or a virtual one).

NEW

Starting/Stopping all the virtual servers

The sysv script `/etc/rc.d/init.d/vserver` is used to start and stop the virtual server at boot and shutdown time. It may be used at any time to operate all virtual servers. The following commands are supported:

- `/etc/rc.d/init.d/vservers start`
- `/etc/rc.d/init.d/vservers stop`
- `/etc/rc.d/init.d/vservers restart`
- `/etc/rc.d/init.d/vservers status`

The **status** command reports the running status of every virtual server. NEW

Restarting a virtual server from inside

A virtual server administrator is not allowed to reboot the machine (the kernel). But it is useful to restart his virtual server from scratch. This allow him to make sure all the services are properly configured to start at boot time.

The `/sbin/vreboot` and `/sbin/vhalt` utilities are installed in each virtual server so they can request a restart or stop.

The **rebootmgr** service must be enabled in the main server.

NEW

Executing tasks at vserver start/stop time

You can setup a script called `/etc/vservers/XX.sh` where XX is the name of the virtual server. This script will be called four time:

- Before starting the vserver
- After starting it.
- Before stopping it.
- After stopping it.

You generally perform tasks such as mounting file system (mapping some directory in the vserver root using "mount --bind").

Here is an example where you map the `/home` directory as the vserver `/home` directory.

```
#!/bin/sh
case $1 in
pre-start)
    mount --bind /home /vservers/$2/home
    ;;
post-start)
    ;;
pre-stop)
    ;;
post-stop)
    umount /vservers/$2/home
    ;;
esac
```

/etc/vservers/XX.sh

NEW

Issues

There are some common problem you may encounter. Here they are.

- The main server is not tied (by default) to any ipv4root. So if the main server has already some service running they are probably binding some UDP or TCP ports using the address 0.0.0.0. Once a process has bound a service with the address 0.0.0.0 (see the LISTEN lines when executing the "netstat -a" command), no other process can bind the same port, even with a specific address.

The solution is to start the services of the main server using the chbind utility to trap them in one ipv4root. For example

```
/sbin/chbind --ip eth0 /etc/rc.d/init.d/httpd start
```

Assigning on IP to a service

will limit Apache to the IP address of the eth0 interface. without configuration changes (in httpd.conf). It is probably a good idea to start the following services in the main server this way, because they will be run by virtual servers as well.

- httpd
- sshd
- xinetd

- To ease this, the **vserver** package supplies the following services: `v_httpd`, `v_sshd`, `v_smb` and `v_xinetd`. Disable the corresponding services and enable the `v_` services and you will lock those services on a single IP.
- Cleanup `rc.local`. This is probably not doing anything useful.

NEW

How real is it ?

The project is new. So far, experiments have shown very little restrictions. Service works the same in a virtual server. Further, performance is the same. And there is a high level of isolation between the various virtual servers and the main server. NEW

Features

There are various tricks one can use to make the virtual servers more secure.

- Putting a fire-wall on the box and limiting access to a virtual services from another one.
- Using port redirection to allow one virtual server to logically bind several IPs. One virtual server could run several web virtual host this way.

NEW

Future directions

NEW

User controlled security box

By combining the capabilities, the `s_context`, the `ipv4root` and the `AcIFs` (component of the [virtualfs](#) package), we can produce a user level tool allowing controlled access to the user own resources. For example the user may download any program he wants and execute them under control. Whenever the program tries to access something not specified by the user, a popup is presented and the user may choose to terminate the program or allow the access.

NEW

Kernel enhancements

We expect to see some wider usage of the virtual servers. As usage grows, we expect to see needs for more control. Here are some ideas.

NEW

Per context disk quota

If one installs virtual servers and grants access to less trusted users, he may want to limit the disk space used. Since a virtual server may create new user accounts and run processes with any user ID it wants, the current kernel disk quota is not powerful enough. First, it can't differentiate between user ID 100 in one virtual server and user ID 100 in another one.

Further, the main administrator may want to control disk space allocated to the virtual server on a server per server basis, unrelated to the various user IDs in use in those virtual servers.

The kernel has already user and group disk quota. Adding security context disk quota should be easily done.

To differentiate between user IDs in virtual servers, the kernel could coin together the security context and the user ID to create a unique ID. The kernel 2.4 now supports 32 user IDs, so combining security context and user ID in a single 32-bit number should be acceptable.

NEW

Global limits

The kernel has support for user limits (memory, processes, file handles). With virtual servers, we may want to limit the resources used by all processes in the virtual server. The security context would be used as the key here. The following resources could be limited on a security context basis (as opposed to user or process basis)

- Memory used
- Processes number
(Done: This is now supported with the `nproc` flag in the kernel 2.4.16ctx-4. By default a new vserver is limited to 1000 processes maximum, configurable).
- File handles

NEW

Scheduler

The scheduler may become security context aware. It could potentially use this to provide some fairness and control priority based on context. Currently the scheduler is process oriented and does not group process together to qualify their priorities. For example, a user running 10 compilations will get more CPU than another user running a single compilation.

Currently, it is possible to raise the nice (lower priority) for all processes in a virtual server. This can't be reversed, so you are setting an upper limit on priority (Just set the S_NICE variable in the vserver configuration file). Note that a virtual server may still start many low priority processes and this can grab significant share of the CPU. A global per security context might be needed to really provide more control and fairness between the various virtual servers.

Done: The sched security context flag group all process in a vserver so their priority is kind of unified. If you have 50 processes running full speed in one vserver, they will take as much CPU resource than a single process in the root server. A vserver can't starve the other... NEW

Security issues

The current kernel + patch provides a fair level of isolation between the virtual servers. User **root** can't take over the system: He sees only his processes, has only access to his area of the file system (chroot) and can't reconfigure the kernel. Yet there are some potential problems. They are fixable. As usage grows, we will know if they are real problems. Comments are welcome:

NEW

/dev/random

Writing to **/dev/random** is not limited by any capability. Any root user (virtual included) is allowed to write there. Is this a problem ?

(kernel expert think it is ok) NEW

/dev/pts

/dev/pts is a virtual file-system used to allocate pseudo-tty. It presents all the pseudo-tty in use on the server (including all virtual server). User root is allowed to read and write to any pseudo-tty, potentially causing problems on other vservers.

Starting with the ctx-6 patch, **/dev/pts** is virtualised. Although the file numbers are allocated from a single pool, a vserver only see the pseudo-tty it owns. NEW

Network devices

Anyone can list the network devices configurations. This may inform a virtual user that another vserver is on the same physical server. By using as much resources as possible in his own vservers, a malicious user could slow down the other server. Modification to the scheduler explained above could stop this.

Starting with the ctx-6 patch, a vserver only see the device corresponding to its IP number. NEW

Alternative technologies

Using virtual servers may be a cost effective alternative to several independent real servers. You get the administrative independence of independent servers, but share some costs including operation costs.

Other technologies exist offering some of the advantages talked in this document as well as other. Two technologies are available on various hardware platform: Virtual machines and partitioning, NEW

Virtual machines

This has been available for mainframes for a while now. You can boot several different OS at once on the same server. This is mainly used to isolate environments. For example, you can install the new version of an OS on the same server, even while the server is running the current version. This allows you to test and do a roll-out gracefully.

The advantages of virtual machines are:

- Total flexibility. You can run many different OS and different version of the same OS, all at once.
- Robustness. You have total isolation. One OS may crash without affecting the other.
- Resource management. You can effectively limit the resources taken by one OS.
- Hardware Independence. The client OS is using virtual disks provided by the host OS.

This technology is not directly available on PCs. The Intel x86 architecture does not support virtualization natively. Some products nevertheless have appeared and provide this. You can run Linux inside Linux, or this other OS (Which BTW has a logo showing a window flying in pieces, which quite frankly tells everything about it).

The solutions available on PCs carry most of the advantages of the virtual machines found on mainframe, except for performance. You can't run that many virtual Linux's using this technology and expect it to fly. One example of this technology is [vmware](#), which is quite useful, especially if you must run this other OS... vmware may be used to run Linux inside Linux, even test Linux installation while running linux... NEW

Partitioning

Partitioning (domains ?) is a way to split the resources of a large server so you end up with independent servers. For example, you can take a 20 CPUs server and create 3 servers, two with 4 CPUs and one with 12. You can very easily re-assign CPUs to servers in case you need more for a given tasks.

This technology provides full Independence, but much less flexibility. If your 12 CPUs server is not working much, the 4 CPUs one can't borrow some CPUs for 5 minutes. NEW

Limitation of those technologies

Oddly, one disadvantage of those technologies is a side effect of their major advantage: Total Independence. Each virtual server is running its own kernel. Cool. This makes the following tasks more difficult or impossible:

- Sharing administrative tasks such as backup. The virtual servers are using volumes in the host server. The host server can't handle the files in those volumes directly without interfering with the client OS. It has to use some services of the client OS to access the file.

The vservers solution does not have this limitation since the virtual servers are living in the same file-system, sharing the same kernel.

- Task monitoring. The virtual servers run their own kernel. As such, the host OS can't spy on the tasks and check for intrusion for example.
- Disk space. Virtual servers are using either volumes or full devices in the host server. This space is pre-allocated to the maximum needed by the server. You end up with a lot of wasted disk space. Imagine running 100 virtual servers this way and allocating say 10 gigs to each. You get the picture. The vservers solution is sharing a common file-system so the free disk space is available to all.

Further, if you are running the same Linux distribution in the virtual servers, you can unify the disk space using hard link and immutable attributes. The `/usr/lib/vserver/vunify` was created to test that. Using information found in the rpm package the script links the files, except configuration ones.

Testing vunify on a vservers installed with a RedHat 6.2 distribution, unifying the packages glibc, binutils, perl, and bash saved 60 megs. Quite a few packages are not changing often and could be unified.

Vservers do not need kernel packages and hardware configuration tools. This also contribute to save disk space.

- File system sharing

A little the same as above. You can't share file system easily between vservers unless you use network services (often slower). Using "mount --bind", it is very easy to "map" any directory of the root server in several

vservers, providing raw speed access (and even sharing the disk cache).

NEW

Conclusion

Virtual servers are interesting because they can provide a higher level of security while potentially reducing the administration task. Common operation such as backup, are shared between all servers. Services such as monitoring may be configured once.

A **Linux** server can run many services at once with a high level of reliability. As servers are evolving, more and more services are added, often unrelated. Unfortunately there are few details here and there, making the server more complex than it is in reality. When one wants to move one service to another server, it is always a little pain: Some user accounts have to be moved and some configuration files. A lot of hand tweaking.

By installing services in separate virtual servers, it becomes much easier to move services around (just by moving a directory although a big one).

Virtual servers may become a preferred way to install common Linux servers. NEW

Download

The ftp site for this project is <ftp://ftp.solucorp.qc.ca/pub/vserver>. You will find there the following components.

- [kernel-2.4.18ctx-10.tar.gz](#)
[kernel-2.4.18ctxsmp-10.tar.gz](#)

A pre-compiled kernel for Pentium class machine and up. An SMP kernel is also supplied.

- [vserver-0.17-1.src.rpm](#)

The source RPM for the vserver utilities

- [vserver-0.17-1.i386.rpm](#)

A compiled rpm for RedHat 7.x and up. Should work on any recent distribution (glibc 2.2). You need a recent distribution to operate a kernel 2.4 anyway.

- [vserver-admin-0.17-1.i386.rpm](#)

Contains the command `/usr/sbin/newsvserver`. It is a GUI to create vservers. It requires the `linuxconf-utils` and `linuxconf-lib` packages. You can get them from [here](#). `linuxconf` itself is not needed though.

- [vserver-0.17.src.tar.gz](#)

The vserver utilities source

- [patch-2.4.18ctx-10](#)

The patch against Linux 2.4.18

- [patches](#)

The various relative patches (ctxN-ctxN+1)

NEW

References

This project is maintained by Jacques Gelinas jack@solucorp.qc.ca

The vserver package is licensed under the GNU PUBLIC LICENSE.

A FAQ can be found at <http://www.solucorp.qc.ca/howto.hc?projet=vserver>

A mailing list has been created to exchange about this project. It is vserver@solucorp.qc.ca. You can subscribe [here](#)

The mailing list is archived [here](#).

The change logs for the vserver package are [here](#).

The official copy of this document is found at http://www.solucorp.qc.ca/miscprj/s_context.hc

This document was produced using the [TLMP documentation system](#)

[Top](#)

Last update: Thu Apr 4 11:47:29 2002

[Back to project page](#)

[About tlpdoc and cookies](#)

Document maintained by Jacques Gélinas (jack@solucorp.qc.ca)

vserver Howto/FAQ

[Howto index](#)

Is it possible to run vservers based on different distro on ?

Yes, no problem. For now the vserver project is a little redhat/mandrake aware, but some are using it with other distro. Once a service is running in a vserver, it is talking directly to the kernel. So a debian vserver could be running on a redhat server or the reverse. The only issues are

- A vserver is normally created from the root server. If you intend to run a different distribution inside a vserver, you will have to copy it from another server, or find a way to install the package somehow.
- The `/usr/sbin/vserver` is redhat-ish and assumes the services are configured in runlevel 3. We intend to solve this by having each vservers runs it owns `/sbin/init`. As such each vserver will have its own way of enabling the services. From `/usr/sbin/vserver`, the only thing to do would be to start/stop/signal `/sbin/init` in the vserver.

vserver Howto/FAQ

[Howto index](#)

Is it possible to move a vserver from one physical server to another

Yes. In fact, a vserver is fairly hardware independent. You can move it from an IDE + uniprocessor server to SCSI + multiprocessor server without any reconfiguration. Just copy `/etc/vservers/XX.conf` and `/vservers/XX` to the new physical server and start it there. To move `/vservers/XX`, you may want to use `rsync`. For example

```
rsync -e ssh -avHl /vservers/XX new-server:/vserver/XX
```

will do. You can use this to have fail-over. If a vserver is kept updated on a regular basis, using either `rsync`, or even network raid, a vserver may be started on a new machine without any fixes.

vserver Howto/FAQ

[Howto index](#)

Is it possible to share one area of a file system between vservers

Vservers are running in chroot environment. As such, they can only see what is under their / directory. So it does not sounds possible to share one area or one file-system between several virtual servers.

There is an option. Kernel 2.4 allows one volume to be mounted several time with different mount point. Say you have a volume /dev/hda3 and you would like to share it between vserver v1 and v2. You can do the following

```
mkdir /vservers/v1/data
mkdir /vservers/v2/data
mount /dev/hda3 /vservers/v1/data
mount /dev/hda3 /vservers/v2/data
```

You can fill the /etc/fstab file so that /dev/hda3 is mounted at boot time.

This is not completely flexible since you can only share a full partition. If you want to share a smaller area (and potentially several of those small area), you can loopback mount a file and share it. For example:

```
dd if=/dev/zero bs=1024k count=10 of=/var/data
/sbin/losetup /dev/loop0 /var/data
/sbin/mke2fs /dev/loop0
mount /dev/loop0 /vservers/v1/data
mount /dev/loop0 /vservers/v2/data
```

Kernel 2.4 also support the mount --bind option. This allows one to connect a directory in multiple places in the file system, even if this directory is not a mount point. For example, you may want to create several vservers to tests various distributions, yet you want to share the /home directory between each. The following command will do the job. This is probably the easiest way to share data between vservers.

```
mount --bind /home /vservers/name/home
```