UML HowTo

Introduction

Welcome to User Mode Linux

User Mode Linux is the first Open Source virtualization platform (first release date 1991) and second virtualization platform for an x86 PC.

How is UML Different from a VM using Virtualization package X?

We have come to assume that virtualization also means some level of hardware emulation. In fact, it does not. As long as a virtualization package provides the OS with devices which the OS can recognize and has a driver for, the devices do not need to emulate real hardware. Most OSes today have built-in support for a number of "fake" devices used only under virtualization. User Mode Linux takes this concept to the ultimate extreme - there is not a single real device in sight. It is 100% artificial or if we use the correct term 100% paravirtual. All UML devices are abstract concepts which map onto something provided by the host - files, sockets, pipes, etc.

The other major difference between UML and various virtualization packages is that there is a distinct difference between the way the UML kernel and the UML programs operate. The UML kernel is just a process running on Linux - same as any other program. It can be run by an unprivileged user and it does not require anything in terms of special CPU features. The UML userspace, however, is a bit different. The Linux kernel on the host machine assists UML in intercepting everything the program running on a UML instance is trying to do and making the UML kernel handle all of its requests. This is different from other virtualization packages which do not make any difference between the guest kernel and guest programs. This difference results in a number of advantages and disadvantages of UML over let's say QEMU which we will cover later in this document.

Why Would I Want User Mode Linux?

- If User Mode Linux kernel crashes, your host kernel is still fine. It is not accelerated in any way (vhost, kvm, etc) and it is not trying to access any devices directly. It is, in fact, a process like any other.
- You can run a usermode kernel as a non-root user (you may need to arrange appropriate permissions for some devices).
- You can run a very small VM with a minimal footprint for a specific task (for example 32M or less).

- You can get extremely high performance for anything which is a "kernel specific task" such as forwarding, firewalling, etc while still being isolated from the host kernel.
- You can play with kernel concepts without breaking things.
- You are not bound by "emulating" hardware, so you can try weird and wonderful concepts which are very difficult to support when emulating real hardware such as time travel and making your system clock dependent on what UML does (very useful for things like tests).
- It's fun.

Why not to run UML

- The syscall interception technique used by UML makes it inherently slower for any userspace applications. While it can do kernel tasks on par with most other virtualization packages, its userspace is **slow**. The root cause is that UML has a very high cost of creating new processes and threads (something most Unix/Linux applications take for granted).
- UML is strictly uniprocessor at present. If you want to run an application which needs many CPUs to function, it is clearly the wrong choice.

Building a UML instance

There is no UML installer in any distribution. While you can use off the shelf install media to install into a blank VM using a virtualization package, there is no UML equivalent. You have to use appropriate tools on your host to build a viable filesystem image.

This is extremely easy on Debian - you can do it using debootstrap. It is also easy on OpenWRT - the build process can build UML images. All other distros - YMMV.

Creating an image

Create a sparse raw disk image:

```
# dd if=/dev/zero of=disk_image_name bs=1 count=1 seek=16G
```

This will create a 16G disk image. The OS will initially allocate only one block and will allocate more as they are written by UML. As of kernel version 4.19 UML fully supports TRIM (as usually used by flash drives). Using TRIM inside the UML image by specifying discard as a mount option or by running `tune2fs -o discard /dev/ubdXX` will request UML to return any unused blocks to the OS.

Create a filesystem on the disk image and mount it:

```
# mkfs.ext4 ./disk_image_name && mount ./disk_image_name /mnt
```

This example uses ext4, any other filesystem such as ext3, btrfs, xfs, jfs, etc will work too.

Create a minimal OS installation on the mounted filesystem:

```
# debootstrap buster /mnt http://deb.debian.org/debian
```

debootstrap does not set up the root password, fstab, hostname or anything related to networking. It is up to the user to do that.

Set the root password - the easiest way to do that is to chroot into the mounted image:

```
# chroot /mnt
# passwd
# exit
```

Edit key system files

UML block devices are called ubds. The fstab created by debootstrap will be empty and it needs an entry for the root file system:

```
/dev/ubd0   ext4    discard,errors=remount-ro  0       1
```

The image hostname will be set to the same as the host on which you are creating its image. It is a good idea to change that to avoid "Oh, bummer, I rebooted the wrong machine".

UML supports two classes of network devices - the older uml_net ones which are scheduled for obsoletion. These are called ethX. It also supports the newer vector IO devices which are significantly faster and have support for some standard virtual network encapsulations like Ethernet over GRE and Ethernet over L2TPv3. These are called vec0.

Depending on which one is in use, `/etc/network/interfaces` will need entries like:

```
# legacy UML network devices
auto eth0
iface eth0 inet dhcp

# vector UML network devices
auto vec0
iface vec0 inet dhcp
```

We now have a UML image which is nearly ready to run, all we need is a UML kernel and modules for it.

Most distributions have a UML package. Even if you intend to use your own kernel, testing the image with a stock one is always a good start. These packages come with a set of modules which should be copied to the target filesystem. The location is distribution dependent. For Debian these reside under /usr/lib/uml/modules. Copy recursively the content of this directory to the mounted UML filesystem:

```
# cp -rax /usr/lib/uml/modules /mnt/lib/modules
```

If you have compiled your own kernel, you need to use the usual "install modules to a location" procedure by running:

```
# make INSTALL_MOD_PATH=/mnt/lib/modules modules_install
```

This will install modules into /mnt/lib/modules/$(KERNELRELEASE). To specify the full module installation path, use:

```
# make MODLIB=/mnt/lib/modules modules_install
```

At this point the image is ready to be brought up.

Setting Up UML Networking

UML networking is designed to emulate an Ethernet connection. This connection may be either point-to-point (similar to a connection between machines using a back-to-back cable) or a connection to a switch. UML supports a wide variety of means to build these connections to all of: local machine, remote machine(s), local and remote UML and other VM instances.

Transport	Type	Capabilities	Throughput

Transport	Type	Capabilities	Throughput
tap	vector	checksum, tso	> 8Gbit
hybrid	vector	checksum, tso, multipacket rx	> 6GBit
raw	vector	checksum, tso, multipacket rx, tx"	> 6GBit
EoGRE	vector	multipacket rx, tx	> 3Gbit
Eol2tpv3	vector	multipacket rx, tx	> 3Gbit
bess	vector	multipacket rx, tx	> 3Gbit
fd	vector	dependent on fd type	varies
tuntap	legacy	none	~ 500Mbit
daemon	legacy	none	~ 450Mbit
socket	legacy	none	~ 450Mbit
pcap	legacy	rx only	~ 450Mbit
ethertap	legacy	obsolete	~ 500Mbit
vde	legacy	obsolete	~ 500Mbit

- All transports which have tso and checksum offloads can deliver speeds approaching 10G on TCP streams.
- All transports which have multi-packet rx and/or tx can deliver pps rates of up to 1Mps or more.
- All legacy transports are generally limited to ~600-700MBit and 0.05Mps.
- GRE and L2TPv3 allow connections to all of: local machine, remote machines, remote network devices and remote UML instances.
- Socket allows connections only between UML instances.
- Daemon and bess require running a local switch. This switch may be connected to the host as well.

Network configuration privileges

The majority of the supported networking modes need `root` privileges. For example, in the legacy tuntap networking mode, users were required to be part of the group associated with the tunnel device.

For newer network drivers like the vector transports, `root` privilege is required to fire an ioctl to setup the tun interface and/or use raw sockets where needed.

This can be achieved by granting the user a particular capability instead of running UML as root. In case of vector transport, a user can add the capability `CAP_NET_ADMIN` or `CAP_NET_RAW` to the uml binary. Thenceforth, UML can be run with normal user privilges, along with full networking.

For example:

```
# sudo setcap cap_net_raw,cap_net_admin+ep linux
```

Configuring vector transports

All vector transports support a similar syntax:

If X is the interface number as in vec0, vec1, vec2, etc, the general syntax for options is:

```
vecX:transport="Transport Name",option=value,option=value,...,option=value
```

Common options

These options are common for all transports:

- `depth=int` - sets the queue depth for vector IO. This is the amount of packets UML will attempt to read or write in a single system call. The default number is 64 and is generally sufficient for most applications that need throughput in the 2-4 Gbit range. Higher speeds may require larger values.
- `mac=XX:XX:XX:XX:XX:XX` - sets the interface MAC address value.
- `gro=[0,1]` - sets GRO off or on. Enables receive/transmit offloads. The effect of this option depends on the host side support in the transport which is being configured. In most cases it will enable TCP segmentation and RX/TX checksumming offloads. The setting must be identical on the host side and the UML side. The UML kernel will produce warnings if it is not. For example, GRO is enabled by default on local machine interfaces (e.g. veth pairs, bridge, etc), so it should be enabled in UML in the corresponding UML transports (raw, tap, hybrid) in order for networking to operate correctly.
- `mtu=int` - sets the interface MTU
- `headroom=int` - adjusts the default headroom (32 bytes) reserved if a packet will need to be re-encapsulated into for instance VXLAN.
- `vec=0` - disable multipacket IO and fall back to packet at a time mode

Shared Options

- `ifname=str` Transports which bind to a local network interface have a shared option - the name of the interface to bind to.
- `src, dst, src_port, dst_port` - all transports which use sockets which have the notion of source and destination and/or source port and destination port use these to specify them.
- `v6=[0,1]` to specify if a v6 connection is desired for all transports which operate over IP. Additionally, for transports that have some differences in the way they operate over v4 and v6 (for example EoL2TPv3), sets the correct mode of operation. In the absense of this option, the socket type is determined based on what do the src and dst arguments resolve/parse to.

tap transport

Example:

```
vecX:transport=tap,ifname=tap0,depth=128,gro=1
```

This will connect vec0 to tap0 on the host. Tap0 must already exist (for example created using tunctl) and UP.

tap0 can be configured as a point-to-point interface and given an IP address so that UML can talk to the host. Alternatively, it is possible to connect UML to a tap interface which is connected to a bridge.

While tap relies on the vector infrastructure, it is not a true vector transport at this point, because Linux does not support multi-packet IO on tap file descriptors for normal userspace apps like UML. This is a privilege which is offered only to something which can hook up to it at kernel level via specialized interfaces like vhost-net. A vhost-net like helper for UML is planned at some point in the future.

Privileges required: tap transport requires either:

- tap interface to exist and be created persistent and owned by the UML user using tunctl. Example `tunctl -u uml-user -t tap0`
- binary to have `CAP_NET_ADMIN` privilege

hybrid transport

Example:

```
vecX:transport=hybrid,ifname=tap0,depth=128,gro=1
```

This is an experimental/demo transport which couples tap for transmit and a raw socket for receive. The raw socket allows multi-packet receive resulting in significantly higher packet rates than normal tap.

Privileges required: hybrid requires `CAP_NET_RAW` capability by the UML user as well as the requirements for the tap transport.

raw socket transport

Example:

```
vecX:transport=raw,ifname=p-veth0,depth=128,gro=1
```

This transport uses vector IO on raw sockets. While you can bind to any interface including a physical one, the most common use it to bind to the "peer" side of a veth pair with the other side configured on the host.

Example host configuration for Debian:

/etc/network/interfaces:

```
auto veth0
iface veth0 inet static
     address 192.168.4.1
     netmask 255.255.255.252
     broadcast 192.168.4.3
     pre-up ip link add veth0 type veth peer name p-veth0 && \
       ifconfig p-veth0 up
```

UML can now bind to p-veth0 like this:

```
vec0:transport=raw,ifname=p-veth0,depth=128,gro=1
```

If the UML guest is configured with 192.168.4.2 and netmask 255.255.255.0 it can talk to the host on 192.168.4.1

The raw transport also provides some support for offloading some of the filtering to the host. The two options to control it are:

- `bpffile=str` filename of raw bpf code to be loaded as a socket filter
- `bpfflash=int` 0/1 allow loading of bpf from inside User Mode Linux. This option allows the use of the ethtool load firmware command to load bpf code.

In either case the bpf code is loaded into the host kernel. While this is presently limited to legacy bpf syntax (not ebpf), it is still a security risk. It is not recommended to allow this unless the User Mode Linux instance is considered trusted.

Privileges required: raw socket transport requires *CAP_NET_RAW* capability.

GRE socket transport

Example:

```
vecX:transport=gre,src=$src_host,dst=$dst_host
```

This will configure an Ethernet over `GRE` (aka `GRETAP` or `GREIRB`) tunnel which will connect the UML instance to a `GRE` endpoint at host dst_host. `GRE` supports the following additional options:

- `rx_key=int` - GRE 32-bit integer key for rx packets, if set, `txkey` must be set too
- `tx_key=int` - GRE 32-bit integer key for tx packets, if set `rx_key` must be set too
- `sequence=[0,1]` - enable GRE sequence
- `pin_sequence=[0,1]` - pretend that the sequence is always reset on each packet (needed to interoperate with some really broken implementations)
- `v6=[0,1]` - force IPv4 or IPv6 sockets respectively
- GRE checksum is not presently supported

GRE has a number of caveats:

- You can use only one GRE connection per IP address. There is no way to multiplex connections as each GRE tunnel is terminated directly on the UML instance.
- The key is not really a security feature. While it was intended as such its "security" is laughable. It is, however, a useful feature to ensure that the tunnel is not misconfigured.

An example configuration for a Linux host with a local address of 192.168.128.1 to connect to a UML instance at 192.168.129.1

/etc/network/interfaces:

```
auto gt0
iface gt0 inet static
 address 10.0.0.1
 netmask 255.255.255.0
 broadcast 10.0.0.255
 mtu 1500
 pre-up ip link add gt0 type gretap local 192.168.128.1 \
        remote 192.168.129.1 || true
 down ip link del gt0 || true
```

Additionally, GRE has been tested versus a variety of network equipment.

Privileges required: GRE requires `CAP_NET_RAW`

l2tpv3 socket transport

Warning. L2TPv3 has a "bug". It is the "bug" known as "has more options than GNU ls". While it has some advantages, there are usually easier (and less verbose) ways to connect a UML instance to something. For example, most devices which support L2TPv3 also support GRE.

Example:

```
vec0:transport=l2tpv3,udp=1,src=$src_host,dst=$dst_host,srcport=$src_port,dstport=$dst_port,depth=128,rx_session=0xffffffff
```

This will configure an Ethernet over L2TPv3 fixed tunnel which will connect the UML instance to a L2TPv3 endpoint at host $dst_host using the L2TPv3 UDP flavour and UDP destination port $dst_port.

L2TPv3 always requires the following additional options:

- `rx_session=int` - l2tpv3 32-bit integer session for rx packets
- `tx_session=int` - l2tpv3 32-bit integer session for tx packets

As the tunnel is fixed these are not negotiated and they are preconfigured on both ends.

Additionally, L2TPv3 supports the following optional parameters.

- `rx_cookie=int` - l2tpv3 32-bit integer cookie for rx packets - same functionality as GRE key, more to prevent misconfiguration than provide actual security
- `tx_cookie=int` - l2tpv3 32-bit integer cookie for tx packets
- `cookie64=[0,1]` - use 64-bit cookies instead of 32-bit.
- `counter=[0,1]` - enable l2tpv3 counter
- `pin_counter=[0,1]` - pretend that the counter is always reset on each packet (needed to interoperate with some really broken implementations)
- `v6=[0,1]` - force v6 sockets
- `udp=[0,1]` - use raw sockets (0) or UDP (1) version of the protocol

L2TPv3 has a number of caveats:

- you can use only one connection per IP address in raw mode. There is no way to multiplex connections as each L2TPv3 tunnel is terminated directly on the UML instance. UDP mode can use different ports for this purpose.

Here is an example of how to configure a Linux host to connect to UML via L2TPv3:

/etc/network/interfaces:

```
auto l2tp1
iface l2tp1 inet static
 address 192.168.126.1
 netmask 255.255.255.0
 broadcast 192.168.126.255
 mtu 1500
 pre-up ip l2tp add tunnel remote 127.0.0.1 \
        local 127.0.0.1 encap udp tunnel_id 2 \
        peer_tunnel_id 2 udp_sport 1706 udp_dport 1707 && \
        ip l2tp add session name l2tp1 tunnel_id 2 \
        session_id 0xffffffff peer_session_id 0xffffffff
 down ip l2tp del session tunnel_id 2 session_id 0xffffffff && \
        ip l2tp del tunnel tunnel_id 2
```

Privileges required: L2TPv3 requires `CAP_NET_RAW` for raw IP mode and no special privileges for the UDP mode.

BESS socket transport

BESS is a high performance modular network switch.

https://github.com/NetSys/bess

It has support for a simple sequential packet socket mode which in the more recent versions is using vector IO for high performance.

Example:

```
vecX:transport=bess,src=$unix_src,dst=$unix_dst
```

This will configure a BESS transport using the unix_src Unix domain socket address as source and unix_dst socket address as destination.

For BESS configuration and how to allocate a BESS Unix domain socket port please see the BESS documentation.

https://github.com/NetSys/bess/wiki/Built-In-Modules-and-Ports

BESS transport does not require any special privileges.

Configuring Legacy transports

Legacy transports are now considered obsolete. Please use the vector versions.

Running UML

This section assumes that either the user-mode-linux package from the distribution or a custom built kernel has been installed on the host.

These add an executable called linux to the system. This is the UML kernel. It can be run just like any other executable. It will take most normal linux kernel arguments as command line arguments. Additionally, it will need some UML-specific arguments in order to do something useful.

Arguments

Mandatory Arguments:

- `mem=int[K,M,G]` - amount of memory. By default in bytes. It will also accept K, M or G qualifiers.
- `ubdX[s,d,c,t]=` virtual disk specification. This is not really mandatory, but it is likely to be needed in nearly all cases so we can specify a root file system. The simplest possible image specification is the name of the image file for the filesystem (created using one of the methods described in Creating an image).
  - UBD devices support copy on write (COW). The changes are kept in a separate file which can be discarded allowing a rollback to the original pristine image. If COW is desired, the UBD image is specified as: `cow_file,master_image`. Example:`ubd0=Filesystem.cow,Filesystem.img`
  - UBD devices can be set to use synchronous IO. Any writes are immediately flushed to disk. This is done by adding `s` after the `ubdX` specification.
  - UBD performs some heuristics on devices specified as a single filename to make sure that a COW file has not been specified as the image. To turn them off, use the `d` flag after `ubdX`.
  - UBD supports TRIM - asking the Host OS to reclaim any unused blocks in the image. To turn it off, specify the `t` flag after `ubdX`.
- `root=` root device - most likely `/dev/ubd0` (this is a Linux filesystem image)

Important Optional Arguments

If UML is run as "linux" with no extra arguments, it will try to start an xterm for every console configured inside the image (up to 6 in most Linux distributions). Each console is started inside an xterm. This makes it nice and easy to use UML on a host with a GUI. It is, however, the wrong approach if UML is to be used as a testing harness or run in a text-only environment.

In order to change this behaviour we need to specify an alternative console and wire it to one of the supported "line" channels. For this we need to map a console to use something different from the default xterm.

Example which will divert console number 1 to stdin/stdout:

```
con1=fd:0,fd:1
```

UML supports a wide variety of serial line channels which are specified using the following syntax

conX=channel_type:options[,channel_type:options]

If the channel specification contains two parts separated by comma, the first one is input, the second one output.

- The null channel - Discard all input or output. Example `con=null` will set all consoles to null by default.
- The fd channel - use file descriptor numbers for input/output. Example: `con1=fd:0,fd:1`.
- The port channel - start a telnet server on TCP port number. Example: `con1=port:4321`. The host must have /usr/sbin/in.telnetd (usually part of a telnetd package) and the port-helper from the UML utilities (see the information for the xterm channel below). UML will not boot until a client connects.
- The pty and pts channels - use system pty/pts.
- The tty channel - bind to an existing system tty. Example: `con1=/dev/tty8` will make UML use the host 8th console (usually unused).
- The xterm channel - this is the default - bring up an xterm on this channel and direct IO to it. Note that in order for xterm to work, the host must have the UML distribution package installed. This usually contains the port-helper and other utilities needed for UML to communicate with the xterm. Alternatively, these need to be complied and installed from source. All options applicable to consoles also apply to UML serial lines which are presented as ttyS inside UML.

Starting UML

We can now run UML.

```
# linux mem=2048M umid=TEST \
 ubd0=Filesystem.img \
 vec0:transport=tap,ifname=tap0,depth=128,gro=1 \
 root=/dev/ubda con=null con0=null,fd:2 con1=fd:0,fd:1
```

This will run an instance with `2048M RAM` and try to use the image file called `Filesystem.img` as root. It will connect to the host using tap0. All consoles except `con1` will be disabled and console 1 will use standard input/output making it appear in the same terminal it was started.

Logging in

If you have not set up a password when generating the image, you will have to shut down the UML instance, mount the image, chroot into it and set it - as described in the Generating an Image section. If the password is already set, you can just log in.

The UML Management Console

In addition to managing the image from "the inside" using normal sysadmin tools, it is possible to perform a number of low-level operations using the UML management console. The UML management console is a low-level interface to the kernel on a running UML instance, somewhat like the i386 SysRq interface. Since there is a full-blown operating system under UML, there is much greater flexibility possible than with the SysRq mechanism.

There are a number of things you can do with the mconsole interface:

- get the kernel version
- add and remove devices
- halt or reboot the machine
- Send SysRq commands
- Pause and resume the UML
- Inspect processes running inside UML
- Inspect UML internal /proc state

You need the mconsole client (uml_mconsole) which is a part of the UML tools package available in most Linux distritions.

You also need `CONFIG_MCONSOLE` (under 'General Setup') enabled in the UML kernel. When you boot UML, you'll see a line like:

```
mconsole initialized on /home/jdike/.uml/umlNJ32yL/mconsole
```

If you specify a unique machine id on the UML command line, i.e. `umid=debian`, you'll see this:

```
mconsole initialized on /home/jdike/.uml/debian/mconsole
```

That file is the socket that uml_mconsole will use to communicate with UML. Run it with either the umid or the full path as its argument:

```
# uml_mconsole debian
```

or

uml_mconsole /home/jdike/.uml/debian/mconsole

You'll get a prompt, at which you can run one of these commands:

- version
- help
- halt
- reboot
- config
- remove
- sysrq
- help
- cad
- stop
- go
- proc
- stack

version

This command takes no arguments. It prints the UML version:

```
(mconsole)  version
OK Linux OpenWrt 4.14.106 #0 Tue Mar 19 08:19:41 2019 x86_64
```

There are a couple actual uses for this. It's a simple no-op which can be used to check that a UML is running. It's also a way of sending a device interrupt to the UML. UML mconsole is treated internally as a UML device.

help

This command takes no arguments. It prints a short help screen with the supported mconsole commands.

halt and reboot

These commands take no arguments. They shut the machine down immediately, with no syncing of disks and no clean shutdown of userspace. So, they are pretty close to crashing the machine:

```
(mconsole)  halt
OK
```

config

"config" adds a new device to the virtual machine. This is supported by most UML device drivers. It takes one argument, which is the device to add, with the same syntax as the kernel command line:

```
(mconsole) config ubd3=/home/jdike/incoming/roots/root_fs_debian22
```

remove

"remove" deletes a device from the system. Its argument is just the name of the device to be removed. The device must be idle in whatever sense the driver considers necessary. In the case of the ubd driver, the removed block device must not be mounted, swapped on, or otherwise open, and in the case of the network driver, the device must be down:

```
(mconsole)  remove ubd3
```

sysrq

This command takes one argument, which is a single letter. It calls the generic kernel's SysRq driver, which does whatever is called for by that argument. See the SysRq documentation in Documentation/admin-guide/sysrq.rst in your favorite kernel tree to see what letters are valid and what they do.

cad

This invokes the `Ctl-Alt-Del` action in the running image. What exactly this ends up doing is up to init, systemd, etc. Normally, it reboots the machine.

stop

This puts the UML in a loop reading mconsole requests until a 'go' mconsole command is received. This is very useful as a debugging/snapshotting tool.

go

This resumes a UML after being paused by a 'stop' command. Note that when the UML has resumed, TCP connections may have timed out and if the UML is paused for a long period of time, crond might go a little crazy, running all the jobs it didn't do earlier.

proc

This takes one argument - the name of a file in /proc which is printed to the mconsole standard output

stack

This takes one argument - the pid number of a process. Its stack is printed to a standard output.

Advanced UML Topics

Sharing Filesystems between Virtual Machines

Don't attempt to share filesystems simply by booting two UMLs from the same file. That's the same thing as booting two physical machines from a shared disk. It will result in filesystem corruption.

Using layered block devices

The way to share a filesystem between two virtual machines is to use the copy-on-write (COW) layering capability of the ubd block driver. Any changed blocks are stored in the private COW file, while reads come from either device - the private one if the requested block is valid in it, the shared one if not. Using this scheme, the majority of data which is unchanged is shared between an arbitrary number of virtual machines, each of which has a much smaller file containing the changes that it has made. With a large number of UMLs booting from a large root filesystem, this leads to a huge disk space saving.

Sharing file system data will also help performance, since the host will be able to cache the shared data using a much smaller amount of memory, so UML disk requests will be served from the host's memory rather than its disks. There is a major caveat in doing this on multisocket NUMA machines. On such hardware, running many UML instances with a shared master image and COW changes may cause issues like NMIs from excess of inter-socket traffic.

If you are running UML on high-end hardware like this, make sure to bind UML to a set of logical CPUs residing on the same socket using the `taskset` command or have a look at the "tuning" section.

To add a copy-on-write layer to an existing block device file, simply add the name of the COW file to the appropriate ubd switch:

```
ubd0=root_fs_cow,root_fs_debian_22
```

where `root_fs_cow` is the private COW file and `root_fs_debian_22` is the existing shared filesystem. The COW file need not exist. If it doesn't, the driver will create and initialize it.

Disk Usage

UML has TRIM support which will release any unused space in its disk image files to the underlying OS. It is important to use either ls -ls or du to verify the actual file size.

COW validity.

Any changes to the master image will invalidate all COW files. If this happens, UML will *NOT* automatically delete any of the COW files and will refuse to boot. In this case the only solution is to either restore the old image (including its last modified timestamp) or remove all COW files which will result in their recreation. Any changes in the COW files will be lost.

Cows can moo - uml_moo : Merging a COW file with its backing file

Depending on how you use UML and COW devices, it may be advisable to merge the changes in the COW file into the backing file

every once in a while.

The utility that does this is uml_moo. Its usage is:

```
uml_moo COW_file new_backing_file
```

There's no need to specify the backing file since that information is already in the COW file header. If you're paranoid, boot the new merged file, and if you're happy with it, move it over the old backing file.

`uml_moo` creates a new backing file by default as a safety measure. It also has a destructive merge option which will merge the COW file directly into its current backing file. This is really only usable when the backing file only has one COW file associated with it. If there are multiple COWs associated with a backing file, a -d merge of one of them will invalidate all of the others. However, it is convenient if you're short of disk space, and it should also be noticeably faster than a non-destructive merge.

`uml_moo` is installed with the UML distribution packages and is available as a part of UML utilities.

Host file access

If you want to access files on the host machine from inside UML, you can treat it as a separate machine and either nfs mount directories from the host or copy files into the virtual machine with scp. However, since UML is running on the host, it can access those files just like any other process and make them available inside the virtual machine without the need to use the network. This is possible with the hostfs virtual filesystem. With it, you can mount a host directory into the UML filesystem and access the files contained in it just as you would on the host.

SECURITY WARNING

Hostfs without any parameters to the UML Image will allow the image to mount any part of the host filesystem and write to it. Always confine hostfs to a specific "harmless" directory (for example `/var/tmp`) if running UML. This is especially important if UML is being run as root.

Using hostfs

To begin with, make sure that hostfs is available inside the virtual machine with:

```
# cat /proc/filesystems
```

`hostfs` should be listed. If it's not, either rebuild the kernel with hostfs configured into it or make sure that hostfs is built as a module and available inside the virtual machine, and insmod it.

Now all you need to do is run mount:

```
# mount none /mnt/host -t hostfs
```

will mount the host's `/` on the virtual machine's `/mnt/host`. If you don't want to mount the host root directory, then you can specify a subdirectory to mount with the -o switch to mount:

```
# mount none /mnt/home -t hostfs -o /home
```

will mount the host's /home on the virtual machine's /mnt/home.

hostfs as the root filesystem

It's possible to boot from a directory hierarchy on the host using hostfs rather than using the standard filesystem in a file. To start, you need that hierarchy. The easiest way is to loop mount an existing root_fs file:

```
#  mount root_fs uml_root_dir -o loop
```

You need to change the filesystem type of `/` in `etc/fstab` to be 'hostfs', so that line looks like this:

```
/dev/ubd/0       /        hostfs      defaults          1   1
```

Then you need to chown to yourself all the files in that directory that are owned by root. This worked for me:

```
#  find . -uid 0 -exec chown jdike {} \;
```

Next, make sure that your UML kernel has hostfs compiled in, not as a module. Then run UML with the boot device pointing at that directory:

```
ubd0=/path/to/uml/root/directory
```

UML should then boot as it does normally.

Hostfs Caveats

Hostfs does not support keeping track of host filesystem changes on the host (outside UML). As a result, if a file is changed without UML's knowledge, UML will not know about it and its own in-memory cache of the file may be corrupt. While it is possible to fix this, it is not something which is being worked on at present.

Tuning UML

UML at present is strictly uniprocessor. It will, however spin up a number of threads to handle various functions.

The UBD driver, SIGIO and the MMU emulation do that. If the system is idle, these threads will be migrated to other processors on a SMP host. This, unfortunately, will usually result in LOWER performance because of all of the cache/memory synchronization traffic between cores. As a result, UML will usually benefit from being pinned on a single CPU, especially on a large system. This can result in performance differences of 5 times or higher on some benchmarks.

Similarly, on large multi-node NUMA systems UML will benefit if all of its memory is allocated from the same NUMA node it will run on. The OS will *NOT* do that by default. In order to do that, the sysadmin needs to create a suitable tmpfs ramdisk bound to a particular node and use that as the source for UML RAM allocation by specifying it in the TMP or TEMP environment variables. UML will look at the values of `TMPDIR`, `TMP` or `TEMP` for that. If that fails, it will look for shmfs mounted under `/dev/shm`. If everything else fails use `/tmp/` regardless of the filesystem type used for it:

```
mount -t tmpfs -ompol=bind:X none /mnt/tmpfs-nodeX
TEMP=/mnt/tmpfs-nodeX taskset -cX linux options options options..
```

Contributing to UML and Developing with UML

UML is an excellent platform to develop new Linux kernel concepts - filesystems, devices, virtualization, etc. It provides unrivalled opportunities to create and test them without being constrained to emulating specific hardware.

Example - want to try how Linux will work with 4096 "proper" network devices?

Not an issue with UML. At the same time, this is something which is difficult with other virtualization packages - they are constrained

by the number of devices allowed on the hardware bus they are trying to emulate (for example 16 on a PCI bus in qemu).

If you have something to contribute such as a patch, a bugfix, a new feature, please send it to `linux-um@lists.infradead.org`.

Please follow all standard Linux patch guidelines such as cc-ing relevant maintainers and run `./scripts/checkpatch.pl` on your patch. For more details see `Documentation/process/submitting-patches.rst`

Note - the list does not accept HTML or attachments, all emails must be formatted as plain text.

Developing always goes hand in hand with debugging. First of all, you can always run UML under gdb and there will be a whole section later on on how to do that. That, however, is not the only way to debug a Linux kernel. Quite often adding tracing statements and/or using UML specific approaches such as ptracing the UML kernel process are significantly more informative.

Tracing UML

When running, UML consists of a main kernel thread and a number of helper threads. The ones of interest for tracing are NOT the ones that are already ptraced by UML as a part of its MMU emulation.

These are usually the first three threads visible in a ps display. The one with the lowest PID number and using most CPU is usually the kernel thread. The other threads are the disk (ubd) device helper thread and the SIGIO helper thread. Running ptrace on this thread usually results in the following picture:

```
host$ strace -p 16566
--- SIGIO {si_signo=SIGIO, si_code=POLL_IN, si_band=65} ---
epoll_wait(4, [{EPOLLIN, {u32=3721159424, u64=3721159424}}], 64, 0) = 1
epoll_wait(4, [], 64, 0)                = 0
rt_sigreturn({mask=[PIPE]})             = 16967
ptrace(PTRACE_GETREGS, 16967, NULL, 0xd5f34f38) = 0
ptrace(PTRACE_GETREGSET, 16967, NT_X86_XSTATE, [{iov_base=0xd5f35010, iov_len=832}]) = 0
ptrace(PTRACE_GETSIGINFO, 16967, NULL, {si_signo=SIGTRAP, si_code=0x85, si_pid=16967, si_uid=0}) = 0
ptrace(PTRACE_SETREGS, 16967, NULL, 0xd5f34f38) = 0
ptrace(PTRACE_SETREGSET, 16967, NT_X86_XSTATE, [{iov_base=0xd5f35010, iov_len=2696}]) = 0
ptrace(PTRACE_SYSEMU, 16967, NULL, 0)   = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_TRAPPED, si_pid=16967, si_uid=0, si_status=SIGTRAP, si_utime=65, si_stime=89} -
wait4(16967, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP | 0x80}], WSTOPPED|__WALL, NULL) = 16967
ptrace(PTRACE_GETREGS, 16967, NULL, 0xd5f34f38) = 0
ptrace(PTRACE_GETREGSET, 16967, NT_X86_XSTATE, [{iov_base=0xd5f35010, iov_len=832}]) = 0
ptrace(PTRACE_GETSIGINFO, 16967, NULL, {si_signo=SIGTRAP, si_code=0x85, si_pid=16967, si_uid=0}) = 0
timer_settime(0, 0, {it_interval={tv_sec=0, tv_nsec=0}, it_value={tv_sec=0, tv_nsec=2830912}}, NULL) = 0
getpid()                                = 16566
clock_nanosleep(CLOCK_MONOTONIC, 0, {tv_sec=1, tv_nsec=0}, NULL) = ? ERESTART_RESTARTBLOCK (Interrupted by signal)
--- SIGALRM {si_signo=SIGALRM, si_code=SI_TIMER, si_timerid=0, si_overrun=0, si_value={int=1631716592, ptr=0x614204f0}} --
rt_sigreturn({mask=[PIPE]})             = -1 EINTR (Interrupted system call)
```

This is a typical picture from a mostly idle UML instance.

- UML interrupt controller uses epoll - this is UML waiting for IO interrupts:

  epoll_wait(4, [{EPOLLIN, {u32=3721159424, u64=3721159424}}], 64, 0) = 1

- The sequence of ptrace calls is part of MMU emulation and running the UML userspace.
- `timer_settime` is part of the UML high res timer subsystem mapping timer requests from inside UML onto the host high resolution timers.
- `clock_nanosleep` is UML going into idle (similar to the way a PC will execute an ACPI idle).

As you can see UML will generate quite a bit of output even in idle. The output can be very informative when observing IO. It shows the actual IO calls, their arguments and returns values.

Kernel debugging

You can run UML under gdb now, though it will not necessarily agree to be started under it. If you are trying to track a runtime bug, it is much better to attach gdb to a running UML instance and let UML run.

Assuming the same PID number as in the previous example, this would be:

```
# gdb -p 16566
```

This will STOP the UML instance, so you must enter *cont* at the GDB command line to request it to continue. It may be a good idea to make this into a gdb script and pass it to gdb as an argument.

Developing Device Drivers

Nearly all UML drivers are monolithic. While it is possible to build a UML driver as a kernel module, that limits the possible functionality to in-kernel only and non-UML specific. The reason for this is that in order to really leverage UML, one needs to write a piece of userspace code which maps driver concepts onto actual userspace host calls.

This forms the so-called "user" portion of the driver. While it can reuse a lot of kernel concepts, it is generally just another piece of userspace code. This portion needs some matching "kernel" code which resides inside the UML image and which implements the Linux kernel part.

Note: There are very few limitations in the way "kernel" and "user" interact.

UML does not have a strictly defined kernel-to-host API. It does not try to emulate a specific architecture or bus. UML's "kernel" and "user" can share memory, code and interact as needed to implement whatever design the software developer has in mind. The only limitations are purely technical. Due to a lot of functions and variables having the same names, the developer should be careful which includes and libraries they are trying to refer to.

As a result a lot of userspace code consists of simple wrappers. E.g. `os_close_file()` is just a wrapper around `close()` which ensures that the userspace function close does not clash with similarly named function(s) in the kernel part.

Using UML as a Test Platform

UML is an excellent test platform for device driver development. As with most things UML, "some user assembly may be required". It is up to the user to build their emulation environment. UML at present provides only the kernel infrastructure.

Part of this infrastructure is the ability to load and parse fdt device tree blobs as used in Arm or Open Firmware platforms. These are supplied as an optional extra argument to the kernel command line:

```
dtb=filename
```

The device tree is loaded and parsed at boottime and is accessible by drivers which query it. At this moment in time this facility is intended solely for development purposes. UML's own devices do not query the device tree.

Security Considerations

Drivers or any new functionality should default to not accepting arbitrary filename, bpf code or other parameters which can affect the host from inside the UML instance. For example, specifying the socket used for IPC communication between a driver and the host at the UML command line is OK security-wise. Allowing it as a loadable module parameter isn't.

If such functionality is desireable for a particular application (e.g. loading BPF "firmware" for raw socket network transports), it should be off by default and should be explicitly turned on as a command line parameter at startup.

Even with this in mind, the level of isolation between UML and the host is relatively weak. If the UML userspace is allowed to load arbitrary kernel drivers, an attacker can use this to break out of UML. Thus, if UML is used in a production application, it is recommended that all modules are loaded at boot and kernel module loading is disabled afterwards.