User Mode Linux HOWTO
User Mode Linux Core Team
Mon Nov 18 14:16:16 EST 2002

This document describes the use and abuse of Jeff Dike's User Mode Linux: a port of the Linux kernel as a normal Intel Linux process.

Table of Contents

1. Introduction

   1.1 How is User Mode Linux Different?
   1.2 Why Would I Want User Mode Linux?

2. Compiling the kernel and modules

   2.1 Compiling the kernel
   2.2 Compiling and installing kernel modules
   2.3 Compiling and installing uml_utilities

3. Running UML and logging in

   3.1 Running UML
   3.2 Logging in
   3.3 Examples

4. UML on 2G/2G hosts

   4.1 Introduction
   4.2 The problem
   4.3 The solution

5. Setting up serial lines and consoles

   5.1 Specifying the device
   5.2 Specifying the channel
   5.3 Examples

6. Setting up the network

   6.1 General setup
   6.2 Userspace daemons
   6.3 Specifying ethernet addresses
   6.4 UML interface setup
   6.5 Multicast
   6.6 TUN/TAP with the uml_net helper
   6.7 TUN/TAP with a preconfigured tap device
   6.8 Ethertap
   6.9 The switch daemon
   6.10 Slip
   6.11 Slirp
   6.12 pcap
   6.13 Setting up the host yourself

7. Sharing Filesystems between Virtual Machines

7.1 A warning
7.2 Using layered block devices
7.3 Note!
7.4 Another warning
7.5 uml_moo : Merging a COW file with its backing file

8. Creating filesystems

8.1 Create the filesystem file
8.2 Assign the file to a UML device
8.3 Creating and mounting the filesystem

9. Host file access

9.1 Using hostfs
9.2 hostfs as the root filesystem
9.3 Building hostfs

10. The Management Console
10.1 version
10.2 halt and reboot
10.3 config
10.4 remove
10.5 sysrq
10.6 help
10.7 cad
10.8 stop
10.9 go

11. Kernel debugging

11.1 Starting the kernel under gdb
11.2 Examining sleeping processes
11.3 Running ddd on UML
11.4 Debugging modules
11.5 Attaching gdb to the kernel
11.6 Using alternate debuggers

12. Kernel debugging examples

12.1 The case of the hung fsck
12.2 Episode 2: The case of the hung fsck

13. What to do when UML doesn't work

13.1 Strange compilation errors when you build from source
13.2 (obsolete)
13.3 A variety of panics and hangs with /tmp on a reiserfs filesystem
13.4 The compile fails with errors about conflicting types for 'open', 'dup', and 'waitpid'
13.5 UML doesn't work when /tmp is an NFS filesystem
13.6 UML hangs on boot when compiled with gprof support
13.7 syslogd dies with a SIGTERM on startup
13.8 TUN/TAP networking doesn't work on a 2.4 host
13.9 You can network to the host but not to other machines on the net

13.10 I have no root and I want to scream
13.11 UML build conflict between ptrace.h and ucontext.h
13.12 The UML BogoMips is exactly half the host's BogoMips
13.13 When you run UML, it immediately segfaults
13.14 xterms appear, then immediately disappear
13.15 Any other panic, hang, or strange behavior

14. Diagnosing Problems

14.1 Case 1 : Normal kernel panics
14.2 Case 2 : Tracing thread panics
14.3 Case 3 : Tracing thread panics caused by other threads
14.4 Case 4 : Hangs

15. Thanks

15.1 Code and Documentation
15.2 Flushing out bugs
15.3 Buglets and clean-ups
15.4 Case Studies
15.5 Other contributions

---

1 1. . I In nt tr ro od du uc ct ti io on n

Welcome to User Mode Linux. It's going to be fun.

1 1. .1 1. . H Ho ow w i is s U Us se er r M Mo od de e L Li in nu ux x D Di if ff fe er re en nt t? ?

Normally, the Linux Kernel talks straight to your hardware (video card, keyboard, hard drives, etc), and any programs which run ask the kernel to operate the hardware, like so:

```
+-----------+-----------+----+
| Process 1 | Process 2 | ...|
+-----------+-----------+----+
|       Linux Kernel         |
+----------------------------+
|         Hardware           |
+----------------------------+
```

The User Mode Linux Kernel is different; instead of talking to the hardware, it talks to a `real' Linux kernel (called the `host kernel' from now on), like any other program. Programs can then run inside User-Mode Linux as if they were running under a normal kernel, like so:

```
                  +-----------------+
                  | Process 2 | ...|
      +-----------+-----------------+
      | Process 1 | User-Mode Linux|
      +-----------------------------+
      |       Linux Kernel          |
      +-----------------------------+
      |         Hardware            |
      +-----------------------------+
```

1 1. .2 2. . W Wh hy y W Wo ou ul ld d I I W Wa an nt t U Us se er r M Mo od de e L Li in nu ux x? ?

1. If User Mode Linux crashes, your host kernel is still fine.

2. You can run a usermode kernel as a non-root user.

3. You can debug the User Mode Linux like any normal process.

4. You can run gprof (profiling) and gcov (coverage testing).

5. You can play with your kernel without breaking things.

6. You can use it as a sandbox for testing new apps.

7. You can try new development kernels safely.

8. You can run different distributions simultaneously.

9. It's extremely fun.

2 2. . C Co om mp pi il li in ng g t th he e k ke er rn ne el l a an nd d m mo od du ul le es s

2 2. .1 1. . C Co om mp pi il li in ng g t th he e k ke er rn ne el l

Compiling the user mode kernel is just like compiling any other kernel. Let's go through the steps, using 2.4.0-prerelease (current as of this writing) as an example:

1. Download the latest UML patch from

   the download page <http://user-mode-linux.sourceforge.net/dl-sf.html>

   In this example, the file is uml-patch-2.4.0-prerelease.bz2.

2. Download the matching kernel from your favourite kernel mirror, such as:

   ftp://ftp.ca.kernel.org/pub/kernel/v2.4/linux-2.4.0-prerelease.tar.bz2 <ftp://ftp.ca.kernel.org/pub/kernel/v2.4/linux-2.4.0-prerelease.tar.bz2>
   .

3. Make a directory and unpack the kernel into it.

```
host%
mkdir ~/uml
```

```
host%
cd ~/uml
```

```
host%
tar -xzvf linux-2.4.0-prerelease.tar.bz2
```

4. Apply the patch using

```
host%
cd ~/uml/linux
```

```
host%
bzcat uml-patch-2.4.0-prerelease.bz2 | patch -p1
```

5. Run your favorite config; `make xconfig ARCH=um' is the most convenient. `make config ARCH=um' and 'make menuconfig ARCH=um' will work as well. The defaults will give you a useful kernel. If you want to change something, go ahead, it probably won't hurt anything.

   Note: If the host is configured with a 2G/2G address space split rather than the usual 3G/1G split, then the packaged UML binaries will not run. They will immediately segfault. See ``UML on 2G/2G hosts'' for the scoop on running UML on your system.

6. Finish with `make linux ARCH=um': the result is a file called `linux' in the top directory of your source tree.

Make sure that you don't build this kernel in /usr/src/linux. On some distributions, /usr/include/asm is a link into this pool. The user-mode build changes the other end of that link, and things that include <asm/anything.h> stop compiling.

The sources are also available from cvs at the project's cvs page, which has directions on getting the sources. You can also browse the CVS pool from there.

If you get the CVS sources, you will have to check them out into an empty directory. You will then have to copy each file into the corresponding directory in the appropriate kernel pool.

If you don't have the latest kernel pool, you can get the corresponding user-mode sources with

```
host% cvs co -r v_2_3_x linux
```

where 'x' is the version in your pool. Note that you will not get the bug fixes and enhancements that have gone into subsequent releases.

2 2. .2 2. . C Co om mp pi il li in ng g a an nd d i in ns st ta al ll li in ng g k ke er rn ne el l m mo od du ul le es s

UML modules are built in the same way as the native kernel (with the exception of the 'ARCH=um' that you always need for UML):

```
host% make modules ARCH=um
```

Any modules that you want to load into this kernel need to be built in the user-mode pool. Modules from the native kernel won't work.

You can install them by using ftp or something to copy them into the virtual machine and dropping them into /lib/modules/`uname -r`.

You can also get the kernel build process to install them as follows:

1. with the kernel not booted, mount the root filesystem in the top level of the kernel pool:

```
host% mount root_fs mnt -o loop
```

2. run

```
host%
make modules_install INSTALL_MOD_PATH=`pwd`/mnt ARCH=um
```

3. unmount the filesystem

```
host% umount mnt
```

4. boot the kernel on it

When the system is booted, you can use insmod as usual to get the modules into the kernel. A number of things have been loaded into UML as modules, especially filesystems and network protocols and filters, so most symbols which need to be exported probably already are. However, if you do find symbols that need exporting, let us <http://user-mode-linux.sourceforge.net/contacts.html> know, and they'll be "taken care of".

2 2. .3 3. . C Co om mp pi il li in ng g a an nd d
i in ns st ta al ll li in ng g u um ml l_ _u ut ti il li it ti ie es s

Many features of the UML kernel require a user-space helper program, so a uml_utilities package is distributed separately from the kernel patch which provides these helpers. Included within this is:

+ o port-helper - Used by consoles which connect to xterms or ports

+ o tunctl - Configuration tool to create and delete tap devices

+ o uml_net - Setuid binary for automatic tap device configuration

+ o uml_switch - User-space virtual switch required for daemon transport

The uml_utilities tree is compiled with:

```
host#
make && make install
```

Note that UML kernel patches may require a specific version of the uml_utilities distribution. If you don't keep up with the mailing lists, ensure that you have the latest release of uml_utilities if you are experiencing problems with your UML kernel, particularly when dealing with consoles or command-line switches to the helper programs

3 3. . R Ru un nn ni in ng g U UM ML L a an nd d l lo og gg gi in ng g i in n

3 3. .1 1. . R Ru un nn ni in ng g U UM ML L

It runs on 2.2.15 or later, and all 2.4 kernels.

Booting UML is straightforward. Simply run 'linux': it will try to mount the file `root_fs' in the current directory. You do not need to run it as root. If your root filesystem is not named `root_fs', then you need to put a `ubd0=root_fs_whatever' switch on the linux command line.

You will need a filesystem to boot UML from. There are a number available for download from here <http://user-mode-linux.sourceforge.net/dl-sf.html> . There are also several tools <http://user-mode-linux.sourceforge.net/fs_making.html> which can be used to generate UML-compatible filesystem images from media.
The kernel will boot up and present you with a login prompt.

Note: If the host is configured with a 2G/2G address space split rather than the usual 3G/1G split, then the packaged UML binaries will not run. They will immediately segfault. See ``UML on 2G/2G hosts'' for the scoop on running UML on your system.

3 3. .2 2. . L Lo og gg gi in ng g i in n

The prepackaged filesystems have a root account with password 'root' and a user account with password 'user'. The login banner will generally tell you how to log in. So, you log in and you will find yourself inside a little virtual machine. Our filesystems have a variety of commands and utilities installed (and it is fairly easy to add more), so you will have a lot of tools with which to poke around the system.

There are a couple of other ways to log in:

+ o On a virtual console

   Each virtual console that is configured (i.e. the device exists in /dev and /etc/inittab runs a getty on it) will come up in its own xterm. If you get tired of the xterms, read ``Setting up serial lines and consoles'' to see how to attach the consoles to something else, like host ptys.

+ o Over the serial line

   In the boot output, find a line that looks like:

```
serial line 0 assigned pty /dev/ptyp1
```

Attach your favorite terminal program to the corresponding tty. I.e. for minicom, the command would be host% minicom -o -p /dev/ttyp1

+ o Over the net

If the network is running, then you can telnet to the virtual machine and log in to it. See ``Setting up the network'' to learn about setting up a virtual network.

When you're done using it, run halt, and the kernel will bring itself down and the process will exit.

3 3. .3 3. . E Ex xa am mp pl le es s

Here are some examples of UML in action:

+ o A login session <http://user-mode-linux.sourceforge.net/login.html>

+ o A virtual network <http://user-mode-linux.sourceforge.net/net.html>

4 4. . U UM ML L o on n 2 2G G/ /2 2G G h ho os st ts s

4 4. .1 1. . I In nt tr ro od du uc ct ti io on n

Most Linux machines are configured so that the kernel occupies the upper 1G (0xc0000000 - 0xffffffff) of the 4G address space and processes use the lower 3G (0x00000000 - 0xbfffffff). However, some machine are configured with a 2G/2G split, with the kernel occupying the upper 2G (0x80000000 - 0xffffffff) and processes using the lower 2G (0x00000000 - 0x7fffffff).

4 4. .2 2. . T Th he e p pr ro ob bl le em m

The prebuilt UML binaries on this site will not run on 2G/2G hosts because UML occupies the upper .5G of the 3G process address space

(0xa0000000 - 0xbfffffff). Obviously, on 2G/2G hosts, this is right in the middle of the kernel address space, so UML won't even load - it will immediately segfault.

4 4. .3 3. . T Th he e s so ol lu ut ti io on n

The fix for this is to rebuild UML from source after enabling CONFIG_HOST_2G_2G (under 'General Setup'). This will cause UML to load itself in the top .5G of that smaller process address space, where it will run fine. See ``Compiling the kernel and modules'' if you need help building UML from source.

5 5. . S Se et tt ti in ng g u up p s se er ri ia al l l li in ne es s a an nd d c co on ns so ol le es s

It is possible to attach UML serial lines and consoles to many types of host I/O channels by specifying them on the command line.

You can attach them to host ptys, ttys, file descriptors, and ports. This allows you to do things like

+ o have a UML console appear on an unused host console,

+ o hook two virtual machines together by having one attach to a pty and having the other attach to the corresponding tty

+ o make a virtual machine accessible from the net by attaching a console to a port on the host.

The general format of the command line option is device=channel.

5 5. .1 1. . S Sp pe ec ci if fy yi in ng g t th he e d de ev vi ic ce e

Devices are specified with "con" or "ssl" (console or serial line, respectively), optionally with a device number if you are talking about a specific device.

Using just "con" or "ssl" describes all of the consoles or serial lines. If you want to talk about console #3 or serial line #10, they would be "con3" and "ssl10", respectively.

A specific device name will override a less general "con=" or "ssl=". So, for example, you can assign a pty to each of the serial lines except for the first two like this:

```
ssl=pty ssl0=tty:/dev/tty0 ssl1=tty:/dev/tty1
```

The specificity of the device name is all that matters; order on the command line is irrelevant.

5 5. .2 2. . S Sp pe ec ci if fy yi in ng g t th he e c ch ha an nn ne el l

There are a number of different types of channels to attach a UML device to, each with a different way of specifying exactly what to attach to.

+ o pseudo-terminals - device=pty pts terminals - device=pts

This will cause UML to allocate a free host pseudo-terminal for the device. The terminal that it got will be announced in the boot log. You access it by attaching a terminal program to the corresponding tty:

+ o screen /dev/pts/n

+ o screen /dev/ttyxx

+ o minicom -o -p /dev/ttyxx - minicom seems not able to handle pts devices

+ o kermit - start it up, 'open' the device, then 'connect'

+ o terminals - device=tty:tty device file

This will make UML attach the device to the specified tty (i.e

```
con1=tty:/dev/tty3
```

will attach UML's console 1 to the host's /dev/tty3). If the tty that you specify is the slave end of a tty/pty pair, something else must have already opened the corresponding pty in order for this to work.

+ o xterms - device=xterm

UML will run an xterm and the device will be attached to it.

+ o Port - device=port:port number

This will attach the UML devices to the specified host port. Attaching console 1 to the host's port 9000 would be done like this:

```
con1=port:9000
```

Attaching all the serial lines to that port would be done similarly:

```
ssl=port:9000
```

You access these devices by telnetting to that port. Each active tel-net session gets a different device. If there are more telnets to a port than UML devices attached to it, then the extra telnet sessions will block until an existing telnet detaches, or until another device becomes active (i.e. by being activated in /etc/inittab).

This channel has the advantage that you can both attach multiple UML devices to it and know how to access them without reading the UML boot log. It is also unique in allowing access to a UML from remote machines without requiring that the UML be networked. This could be useful in allowing public access to UMLs because they would be accessible from the net, but wouldn't need any kind of network filtering or access control because they would have no network access.

If you attach the main console to a portal, then the UML boot will appear to hang. In reality, it's waiting for a telnet to connect, at which point the boot will proceed.

+ o already-existing file descriptors - device=file descriptor

If you set up a file descriptor on the UML command line, you can attach a UML device to it. This is most commonly used to put the main console back on stdin and stdout after assigning all the other consoles to something else:

```
con0=fd:0,fd:1 con=pts
```

+ o Nothing - device=null

This allows the device to be opened, in contrast to 'none', but reads will block, and writes will succeed and the data will be thrown out.

+ o None - device=none

This causes the device to disappear.

You can also specify different input and output channels for a device by putting a comma between them:

```
ssl3=tty:/dev/tty2,xterm
```

will cause serial line 3 to accept input on the host's /dev/tty3 and display output on an xterm. That's a silly example - the most common use of this syntax is to reattach the main console to stdin and stdout as shown above.

If you decide to move the main console away from stdin/stdout, the initial boot output will appear in the terminal that you're running UML in. However, once the console driver has been officially initialized, then the boot output will start appearing wherever you specified that console 0 should be. That device will receive all subsequent output.

5 5. .3 3. . E Ex xa am mp pl le es s

There are a number of interesting things you can do with this capability.

First, this is how you get rid of those bleeding console xterms by attaching them to host ptys:

```
con=pty con0=fd:0,fd:1
```

This will make a UML console take over an unused host virtual console, so that when you switch to it, you will see the UML login prompt rather than the host login prompt:

```
con1=tty:/dev/tty6
```

You can attach two virtual machines together with what amounts to a serial line as follows:

Run one UML with a serial line attached to a pty -

```
ssl1=pty
```

Look at the boot log to see what pty it got (this example will assume that it got /dev/ptyp1).

Boot the other UML with a serial line attached to the corresponding tty -

```
ssl1=tty:/dev/ttyp1
```

Log in, make sure that it has no getty on that serial line, attach a terminal program like minicom to it, and you should see the login prompt of the other virtual machine.

6 6. . S Se et tt ti in ng g u up p t th he e n ne et tw wo or rk k

This page describes how to set up the various transports and to provide a UML instance with network access to the host, other machines on the local net, and the rest of the net.

As of 2.4.5, UML networking has been completely redone to make it much easier to set up, fix bugs, and add new features.

There is a new helper, uml_net, which does the host setup that requires root privileges.

There are currently five transport types available for a UML virtual machine to exchange packets with other hosts:

+ o ethertap

+ o TUN/TAP

+ o Multicast

+ o a switch daemon

+ o slip

+ o slirp

+ o pcap

The TUN/TAP, ethertap, slip, and slirp transports allow a UML instance to exchange packets with the host. They may be directed to the host or the host may just act as a router to provide access to other physical or virtual machines.

The pcap transport is a synthetic read-only interface, using the libpcap binary to collect packets from interfaces on the host and filter them. This is useful for building preconfigured traffic monitors or sniffers.

The daemon and multicast transports provide a completely virtual network to other virtual machines. This network is completely

disconnected from the physical network unless one of the virtual
machines on it is acting as a gateway.

With so many host transports, which one should you use? Here's when
you should use each one:

+ o ethertap - if you want access to the host networking and it is
running 2.2

+ o TUN/TAP - if you want access to the host networking and it is
running 2.4. Also, the TUN/TAP transport is able to use a
preconfigured device, allowing it to avoid using the setuid uml_net
helper, which is a security advantage.

+ o Multicast - if you want a purely virtual network and you don't want
to set up anything but the UML

+ o a switch daemon - if you want a purely virtual network and you
don't mind running the daemon in order to get somewhat better
performance

+ o slip - there is no particular reason to run the slip backend unless
ethertap and TUN/TAP are just not available for some reason

+ o slirp - if you don't have root access on the host to setup
networking, or if you don't want to allocate an IP to your UML

+ o pcap - not much use for actual network connectivity, but great for
monitoring traffic on the host

Ethertap is available on 2.4 and works fine. TUN/TAP is preferred
to it because it has better performance and ethertap is officially
considered obsolete in 2.4. Also, the root helper only needs to
run occasionally for TUN/TAP, rather than handling every packet, as
it does with ethertap. This is a slight security advantage since
it provides fewer opportunities for a nasty UML user to somehow
exploit the helper's root privileges.

6 6. .1 1. . G Ge en ne er ra al l s se et tu up p

First, you must have the virtual network enabled in your UML. If are
running a prebuilt kernel from this site, everything is already
enabled. If you build the kernel yourself, under the "Network device
support" menu, enable "Network device support", and then the three
transports.

The next step is to provide a network device to the virtual machine.
This is done by describing it on the kernel command line.

The general format is

```
     eth <n> = <transport> , <transport args>
```

For example, a virtual ethernet device may be attached to a host
ethertap device as follows:

```
     eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254
```

This sets up eth0 inside the virtual machine to attach itself to the
host /dev/tap0, assigns it an ethernet address, and assigns the host
tap0 interface an IP address.

Note that the IP address you assign to the host end of the tap device
must be different than the IP you assign to the eth device inside UML.
If you are short on IPs and don't want to consume two per UML, then
you can reuse the host's eth IP address for the host ends of the tap
devices. Internally, the UMLs must still get unique IPs for their eth
devices. You can also give the UMLs non-routable IPs (192.168.x.x or
10.x.x.x) and have the host masquerade them. This will let outgoing
connections work, but incoming connections won't without more work,
such as port forwarding from the host.
Also note that when you configure the host side of an interface, it is
only acting as a gateway. It will respond to pings sent to it
locally, but is not useful to do that since it's a host interface.
You are not talking to the UML when you ping that interface and get a
response.

You can also add devices to a UML and remove them at runtime. See the
``The Management Console'' page for details.

The sections below describe this in more detail.

Once you've decided how you're going to set up the devices, you boot
UML, log in, configure the UML side of the devices, and set up routes
to the outside world. At that point, you will be able to talk to any
other machines, physical or virtual, on the net.

If ifconfig inside UML fails and the network refuses to come up, run
tell you what went wrong.

6 6. .2 2. . U Us se er rs sp pa ac ce e d da ae em mo on ns s

You will likely need the setuid helper, or the switch daemon, or both.

They are both installed with the RPM and deb, so if you've installed either, you can skip the rest of this section.

If not, then you need to check them out of CVS, build them, and install them. The helper is uml_net, in CVS /tools/uml_net, and the daemon is uml_switch, in CVS /tools/uml_router. They are both built with a plain 'make'. Both need to be installed in a directory that's in your path - /usr/bin is recommend. On top of that, uml_net needs to be setuid root.

6 6. .3 3. . S Sp pe ec ci if fy yi in ng g e et th he er rn ne et t a ad dd dr re es ss se es s

Below, you will see that the TUN/TAP, ethertap, and daemon interfaces allow you to specify hardware addresses for the virtual ethernet devices. This is generally not necessary. If you don't have a specific reason to do it, you probably shouldn't. If one is not specified on the command line, the driver will assign one based on the device IP address. It will provide the address fe:fd:nn:nn:nn:nn where nn.nn.nn.nn is the device IP address. This is nearly always sufficient to guarantee a unique hardware address for the device. A couple of exceptions are:

+ o Another set of virtual ethernet devices are on the same network and they are assigned hardware addresses using a different scheme which may conflict with the UML IP address-based scheme

+ o You aren't going to use the device for IP networking, so you don't assign the device an IP address

  If you let the driver provide the hardware address, you should make sure that the device IP address is known before the interface is brought up. So, inside UML, this will guarantee that:

```
UML#
ifconfig eth0 192.168.0.250 up
```

If you decide to assign the hardware address yourself, make sure that the first byte of the address is even. Addresses with an odd first byte are broadcast addresses, which you don't want assigned to a device.

6 6. .4 4. . U UM ML L i in nt te er rf fa ac ce e s se et tu up p

Once the network devices have been described on the command line, you should boot UML and log in.

The first thing to do is bring the interface up:

```
UML# ifconfig ethn ip-address up
```

You should be able to ping the host at this point.

To reach the rest of the world, you should set a default route to the host:

```
UML# route add default gw host ip
```

Again, with host ip of 192.168.0.4:

```
UML# route add default gw 192.168.0.4
```

This page used to recommend setting a network route to your local net. This is wrong, because it will cause UML to try to figure out hardware addresses of the local machines by arping on the interface to the host. Since that interface is basically a single strand of ethernet with two nodes on it (UML and the host) and arp requests don't cross networks, they will fail to elicit any responses. So, what you want is for UML to just blindly throw all packets at the host and let it figure out what to do with them, which is what leaving out the network route and adding the default route does.

Note: If you can't communicate with other hosts on your physical ethernet, it's probably because of a network route that's automatically set up. If you run 'route -n' and see a route that looks like this:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0      0   eth0
```

with a mask that's not 255.255.255.255, then replace it with a route to your host:

```
UML#
route del -net 192.168.0.0 dev eth0 netmask 255.255.255.0
```

```
UML#
route add -host 192.168.0.4 dev eth0
```

This, plus the default route to the host, will allow UML to exchange
packets with any machine on your ethernet.

6 6. .5 5. . M Mu ul lt ti ic ca as st t

The simplest way to set up a virtual network between multiple UMLs is
to use the mcast transport. This was written by Harald Welte and is
present in UML version 2.4.5-5um and later. Your system must have
multicast enabled in the kernel and there must be a multicast-capable
network device on the host. Normally, this is eth0, but if there is
no ethernet card on the host, then you will likely get strange error
messages when you bring the device up inside UML.

To use it, run two UMLs with

```
eth0=mcast
```

on their command lines. Log in, configure the ethernet device in each
machine with different IP addresses:

```
UML1# ifconfig eth0 192.168.0.254
```

```
UML2# ifconfig eth0 192.168.0.253
```

and they should be able to talk to each other.

The full set of command line options for this transport are

```
ethn=mcast,ethernet address,multicast
address,multicast port,ttl
```

Harald's original README is here <http://user-mode-linux.source-forge.net/text/mcast.txt> and explains these in detail, as well as some other issues.

6 6. .6 6. . T TU UN N/ /T TA AP P w wi it th h t th he e
u um ml l_ _n ne et t h he el lp pe er r

TUN/TAP is the preferred mechanism on 2.4 to exchange packets with the host. The TUN/TAP backend has been in UML since 2.4.9-3um.

The easiest way to get up and running is to let the setuid uml_net helper do the host setup for you. This involves insmod-ing the tun.o module if necessary, configuring the device, and setting up IP forwarding, routing, and proxy arp. If you are new to UML networking, do this first. If you're concerned about the security implications of the setuid helper, use it to get up and running, then read the next section to see how to have UML use a preconfigured tap device, which avoids the use of uml_net.

If you specify an IP address for the host side of the device, the uml_net helper will do all necessary setup on the host - the only requirement is that TUN/TAP be available, either built in to the host kernel or as the tun.o module.

The format of the command line switch to attach a device to a TUN/TAP device is

```
eth <n> =tuntap,,, <IP address>
```

For example, this argument will attach the UML's eth0 to the next available tap device and assign an ethernet address to it based on its IP address

```
eth0=tuntap,,,192.168.0.254
```

Note that the IP address that must be used for the eth device inside UML is fixed by the routing and proxy arp that is set up on the TUN/TAP device on the host. You can use a different one, but it won't work because reply packets won't reach the UML. This is a feature. It prevents a nasty UML user from doing things like setting the UML IP to the same as the network's nameserver or mail server.

There are a couple potential problems with running the TUN/TAP transport on a 2.4 host kernel

+ o TUN/TAP seems not to work on 2.4.3 and earlier. Upgrade the host kernel or use the ethertap transport.

+ o With an upgraded kernel, TUN/TAP may fail with

```
File descriptor in bad state
```

This is due to a header mismatch between the upgraded kernel and the kernel that was originally installed on the machine. The fix is to make sure that /usr/src/linux points to the headers for the running kernel.

These were pointed out by Tim Robinson <timro at trkr dot net> in <http://www.geocrawler.com/lists/3/SourceForge/597/0/> name="this uml-user post"> .

6 6. .7 7. . T TU UN N/ /T TA AP P w wi it th h a a
p pr re ec co on nf fi ig gu ur re ed d t ta ap p d de ev vi ic ce e

If you prefer not to have UML use uml_net (which is somewhat insecure), with UML 2.4.17-11, you can set up a TUN/TAP device beforehand. The setup needs to be done as root, but once that's done, there is no need for root assistance. Setting up the device is done as follows:

+ o Create the device with tunctl (available from the UML utilities tarball)

```
host#  tunctl -u uid
```

where uid is the user id or username that UML will be run as. This will tell you what device was created.

+ o Configure the device IP (change IP addresses and device name to suit)

```
host#  ifconfig tap0 192.168.0.254 up
```

+ o Set up routing and arping if desired - this is my recipe, there are other ways of doing the same thing

```
host#
bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'

host#
route add -host 192.168.0.253 dev tap0
```

```
host#
bash -c 'echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp'
```

```
host#
arp -Ds 192.168.0.253 eth0 pub
```

Note that this must be done every time the host boots - this configuration is not stored across host reboots. So, it's probably a good idea to stick it in an rc file. An even better idea would be a little utility which reads the information from a config file and sets up devices at boot time.

+ o Rather than using up two IPs and ARPing for one of them, you can also provide direct access to your LAN by the UML by using a bridge.

```
host#
brctl addbr br0
```

```
host#
ifconfig eth0 0.0.0.0 promisc up
```

```
host#
ifconfig tap0 0.0.0.0 promisc up
```

```
host#
ifconfig br0 192.168.0.1 netmask 255.255.255.0 up
```

```
host#
brctl stp br0 off
```

```
host#
brctl setfd br0 1
```

```
host#
brctl sethello br0 1
```

```
host#
brctl addif br0 eth0
```

```
host#
brctl addif br0 tap0
```

Note that 'br0' should be setup using ifconfig with the existing IP address of eth0, as eth0 no longer has its own IP.

+ o

Also, the /dev/net/tun device must be writable by the user running UML in order for the UML to use the device that's been configured for it. The simplest thing to do is

```
host#  chmod 666 /dev/net/tun
```

Making it world-writable looks bad, but it seems not to be exploitable as a security hole. However, it does allow anyone to create useless tap devices (useless because they can't configure them), which is a DOS attack. A somewhat more secure alternative would to be to create a group containing all the users who have preconfigured tap devices and chgrp /dev/net/tun to that group with mode 664 or 660.

+ o Once the device is set up, run UML with 'eth0=tuntap,device name' (i.e. 'eth0=tuntap,tap0') on the command line (or do it with the mconsole config command).

+ o Bring the eth device up in UML and you're in business.

If you don't want that tap device any more, you can make it non-persistent with

```
host#  tunctl -d tap device
```

Finally, tunctl has a -b (for brief mode) switch which causes it to output only the name of the tap device it created. This makes it suitable for capture by a script:

```
host#  TAP=`tunctl -u 1000 -b`
```

6 6. .8 8. . E Et th he er rt ta ap p

Ethertap is the general mechanism on 2.2 for userspace processes to exchange packets with the kernel.

To use this transport, you need to describe the virtual network device on the UML command line. The general format for this is

```
eth <n> =ethertap, <device> , <ethernet address> , <tap IP address>
```

So, the previous example

```
eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254
```

attaches the UML eth0 device to the host /dev/tap0, assigns it the ethernet address fe:fd:0:0:0:1, and assigns the IP address 192.168.0.254 to the tap device.

The tap device is mandatory, but the others are optional. If the ethernet address is omitted, one will be assigned to it.

The presence of the tap IP address will cause the helper to run and do whatever host setup is needed to allow the virtual machine to communicate with the outside world. If you're not sure you know what you're doing, this is the way to go.

If it is absent, then you must configure the tap device and whatever arping and routing you will need on the host. However, even in this case, the uml_net helper still needs to be in your path and it must be setuid root if you're not running UML as root. This is because the tap device doesn't support SIGIO, which UML needs in order to use something as a source of input. So, the helper is used as a convenient asynchronous IO thread.

If you're using the uml_net helper, you can ignore the following host setup - uml_net will do it for you. You just need to make sure you have ethertap available, either built in to the host kernel or available as a module.

If you want to set things up yourself, you need to make sure that the appropriate /dev entry exists. If it doesn't, become root and create it as follows:

```
mknod /dev/tap <minor>  c 36  <minor>  + 16
```

For example, this is how to create /dev/tap0:

```
mknod /dev/tap0 c 36 0 + 16
```

You also need to make sure that the host kernel has ethertap support. If ethertap is enabled as a module, you apparently need to insmod ethertap once for each ethertap device you want to enable. So,

```
host#
insmod ethertap
```

will give you the tap0 interface. To get the tap1 interface, you need to run

```
host#
insmod ethertap unit=1 -o ethertap1
```

6 6. .9 9. . T Th he e s sw wi it tc ch h d da ae em mo on n

N No ot te e: This is the daemon formerly known as uml_router, but which was renamed so the network weenies of the world would stop growling at me.

The switch daemon, uml_switch, provides a mechanism for creating a totally virtual network. By default, it provides no connection to the host network (but see -tap, below).

The first thing you need to do is run the daemon. Running it with no arguments will make it listen on a default pair of unix domain sockets.

If you want it to listen on a different pair of sockets, use

```
-unix control socket data socket
```

If you want it to act as a hub rather than a switch, use

```
-hub
```

If you want the switch to be connected to host networking (allowing the umls to get access to the outside world through the host), use

```
-tap tap0
```

Note that the tap device must be preconfigured (see "TUN/TAP with a preconfigured tap device", above). If you're using a different tap device than tap0, specify that instead of tap0.

uml_switch can be backgrounded as follows

```
host%
uml_switch [ options ] < /dev/null > /dev/null
```

The reason it doesn't background by default is that it listens to stdin for EOF. When it sees that, it exits.

The general format of the kernel command line switch is

```
     ethn=daemon,ethernet address,socket
     type,control socket,data socket
```

You can leave off everything except the 'daemon'. You only need to specify the ethernet address if the one that will be assigned to it isn't acceptable for some reason. The rest of the arguments describe how to communicate with the daemon. You should only specify them if you told the daemon to use different sockets than the default. So, if you ran the daemon with no arguments, running the UML on the same machine with

```
     eth0=daemon
```

will cause the eth0 driver to attach itself to the daemon correctly.

6 6. .1 10 0. . S Sl li ip p

Slip is another, less general, mechanism for a process to communicate with the host networking. In contrast to the ethertap interface, which exchanges ethernet frames with the host and can be used to transport any higher-level protocol, it can only be used to transport IP.

The general format of the command line switch is

```
     ethn=slip,slip IP
```

The slip IP argument is the IP address that will be assigned to the host end of the slip device. If it is specified, the helper will run and will set up the host so that the virtual machine can reach it and the rest of the network.

There are some oddities with this interface that you should be aware of. You should only specify one slip device on a given virtual machine, and its name inside UML will be 'umn', not 'eth0' or whatever you specified on the command line. These problems will be fixed at some point.

6 6. .1 11 1. . S Sl li ir rp p

slirp uses an external program, usually /usr/bin/slirp, to provide IP only networking connectivity through the host. This is similar to IP masquerading with a firewall, although the translation is performed in user-space, rather than by the kernel. As slirp does not set up any interfaces on the host, or changes routing, slirp does not require root access or setuid binaries on the host.

The general format of the command line switch for slirp is:

```
ethn=slirp,ethernet address,slirp path
```

The ethernet address is optional, as UML will set up the interface with an ethernet address based upon the initial IP address of the interface. The slirp path is generally /usr/bin/slirp, although it will depend on distribution.

The slirp program can have a number of options passed to the command line and we can't add them to the UML command line, as they will be parsed incorrectly. Instead, a wrapper shell script can be written or the options inserted into the /.slirprc file. More information on all of the slirp options can be found in its man pages.

The eth0 interface on UML should be set up with the IP 10.2.0.15, although you can use anything as long as it is not used by a network you will be connecting to. The default route on UML should be set to use

```
UML#
route add default dev eth0
```

slirp provides a number of useful IP addresses which can be used by UML, such as 10.0.2.3 which is an alias for the DNS server specified in /etc/resolv.conf on the host or the IP given in the 'dns' option for slirp.

Even with a baudrate setting higher than 115200, the slirp connection is limited to 115200. If you need it to go faster, the slirp binary needs to be compiled with FULL_BOLT defined in config.h.

6 6. .1 12 2. . p pc ca ap p

The pcap transport is attached to a UML ethernet device on the command line or with uml_mconsole with the following syntax:

```
ethn=pcap,host interface,filter
expression,option1,option2
```

The expression and options are optional.

The interface is whatever network device on the host you want to sniff. The expression is a pcap filter expression, which is also what tcpdump uses, so if you know how to specify tcpdump filters, you will use the same expressions here. The options are up to two of 'promisc', control whether pcap puts the host interface into promiscuous mode. 'optimize' and 'nooptimize' control whether the pcap expression optimizer is used.

Example:

```
eth0=pcap,eth0,tcp

eth1=pcap,eth0,!tcp
```

will cause the UML eth0 to emit all tcp packets on the host eth0 and the UML eth1 to emit all non-tcp packets on the host eth0.

6 6. .1 13 3. . S Se et tt ti in ng g u up p t th he e h ho os st t
y yo ou ur rs se el lf f

If you don't specify an address for the host side of the ethertap or slip device, UML won't do any setup on the host. So this is what is needed to get things working (the examples use a host-side IP of 192.168.0.251 and a UML-side IP of 192.168.0.250 - adjust to suit your own network):

+ o The device needs to be configured with its IP address. Tap devices are also configured with an mtu of 1484. Slip devices are configured with a point-to-point address pointing at the UML ip address.

```
host#  ifconfig tap0 arp mtu 1484 192.168.0.251 up
```

```
host#
ifconfig sl0 192.168.0.251 pointopoint 192.168.0.250 up
```

+ o If a tap device is being set up, a route is set to the UML IP.

```
UML# route add -host 192.168.0.250 gw 192.168.0.251
```

+ o To allow other hosts on your network to see the virtual machine,
proxy arp is set up for it.

```
host#  arp -Ds 192.168.0.250 eth0 pub
```

+ o Finally, the host is set up to route packets.

```
host#  echo 1 > /proc/sys/net/ipv4/ip_forward
```

7 7. . S Sh ha ar ri in ng g F Fi il le es sy ys st te em ms s
b be et tw we ee en n V Vi ir rt tu ua al l M Ma ac ch hi in ne es s

7 7. .1 1. . A A w wa ar rn ni in ng g

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.

7 7. .2 2. . U Us si in ng g l la ay ye er re ed d b bl lo oc ck k
d de ev vi ic ce es s

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. As of 2.4.6-2um, the driver supports layering a read-write private device over a read-only shared device. A machine's writes are stored in the private device, 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. It 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.

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. Once the COW file has been initialized, it can be used on its own on the command line:

```
ubd0=root_fs_cow
```

The name of the backing file is stored in the COW file header, so it would be redundant to continue specifying it on the command line.

7 7. .3 3. . N No ot te e! !

When checking the size of the COW file in order to see the gobs of space that you're saving, make sure you use 'ls -ls' to see the actual disk consumption rather than the length of the file. The COW file is sparse, so the length will be very different from the disk usage. Here is a 'ls -l' of a COW file and backing file from one boot and shutdown:

```
host% ls -l cow.debian debian2.2
-rw-r--r--    1 jdike    jdike    492504064 Aug  6 21:16 cow.debian
-rwxrw-rw-    1 jdike    jdike    537919488 Aug  6 20:42 debian2.2
```

Doesn't look like much saved space, does it? Well, here's 'ls -ls':

```
host% ls -ls cow.debian debian2.2
   880 -rw-r--r--    1 jdike    jdike    492504064 Aug  6 21:16
cow.debian
525832 -rwxrw-rw-    1 jdike    jdike    537919488 Aug  6 20:42 debian2.2
```

Now, you can see that the COW file has less than a meg of disk, rather than 492 meg.

7 7. .4 4. . A An no ot th he er r w wa ar rn ni in ng g

Once a filesystem is being used as a readonly backing file for a COW file, do not boot directly from it or modify it in any way. Doing so will invalidate any COW files that are using it. The mtime and size of the backing file are stored in the COW file header at its creation, and they must continue to match. If they don't, the driver will refuse to use the COW file.

If you attempt to evade this restriction by changing either the backing file or the COW header by hand, you will get a corrupted filesystem.

Among other things, this means that upgrading the distribution in a backing file and expecting that all of the COW files using it will see the upgrade will not work.

7 7. .5 5. . u um ml l_ _m mo oo o : : M Me er rg gi in ng g a a C CO OW W

f fi il le e w wi it th h i it ts s b ba ac ck ki in ng g f fi il le e

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

```
host% 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 deb and RPM. If you didn't install UML from one of those packages, you can also get it from the UML utilities <http://user-mode-linux.sourceforge.net/dl-sf.html#UML utilities> tar file in tools/moo.

8 8. . C Cr re ea at ti in ng g f fi il le es sy ys st te em ms s

You may want to create and mount new UML filesystems, either because your root filesystem isn't large enough or because you want to use a filesystem other than ext2.

This was written on the occasion of reiserfs being included in the 2.4.1 kernel pool, and therefore the 2.4.1 UML, so the examples will talk about reiserfs. This information is generic, and the examples should be easy to translate to the filesystem of your choice.

8 8. .1 1. . C Cr re ea at te e t th he e f fi il le es sy ys st te em m f fi il le e

dd is your friend. All you need to do is tell dd to create an empty file of the appropriate size. I usually make it sparse to save time and to avoid allocating disk space until it's actually used. For example, the following command will create a sparse 100 meg file full of zeroes.

```
host%
dd if=/dev/zero of=new_filesystem seek=100 count=1 bs=1M
```

8 8. .2 2. . A As ss si ig gn n t th he e f fi il le e t to o a a U UM ML L d de ev vi ic ce e

Add an argument like the following to the UML command line:

ubd4=new_filesystem

making sure that you use an unassigned ubd device number.

8 8. .3 3. . C Cr re ea at ti in ng g a an nd d m mo ou un nt ti in ng g t th he e f fi il le es sy ys st te em m

Make sure that the filesystem is available, either by being built into the kernel, or available as a module, then boot up UML and log in. If the root filesystem doesn't have the filesystem utilities (mkfs, fsck, etc), then get them into UML by way of the net or hostfs.

Make the new filesystem on the device assigned to the new file:

```
host#  mkreiserfs /dev/ubd/4


<----------- MKREISERFSv2 ----------->
```

```
ReiserFS version 3.6.25
Block size 4096 bytes
Block count 25856
Used blocks 8212
        Journal - 8192 blocks (18-8209), journal header is in block 8210
        Bitmaps: 17
        Root block 8211
Hash function "r5"
ATTENTION: ALL DATA WILL BE LOST ON '/dev/ubd/4'! (y/n)y
journal size 8192 (from 18)
Initializing journal - 0%....20%....40%....60%....80%....100%
Syncing..done.
```

Now, mount it:

```
UML#
mount /dev/ubd/4 /mnt
```

and you're in business.

9. 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 or rcp. 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 needing to use the network.

This is now 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.

9.1. Using hostfs

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

```
UML# 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:

```
UML# 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:

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

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

9 9. .2 2. . h ho os st tf fs s a as s t th he e r ro oo ot t f fi il le es sy ys st te em m

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:

```
host#  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:

```
host#  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.

9 9. .3 3. . B Bu ui il ld di in ng g h ho os st tf fs s

If you need to build hostfs because it's not in your kernel, you have two choices:

+ o Compiling hostfs into the kernel:

   Reconfigure the kernel and set the 'Host filesystem' option under

+ o Compiling hostfs as a module:

   Reconfigure the kernel and set the 'Host filesystem' option under
   be in arch/um/fs/hostfs/hostfs.o. Install that in
   /lib/modules/`uname -r`/fs in the virtual machine, boot it up, and

```
UML# insmod hostfs
```

1 10 0. . T Th he e M Ma an na ag ge em me en nt t C Co on ns so ol le e

The UML management console is a low-level interface to the kernel, 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:

+ o get the kernel version

+ o add and remove devices

+ o halt or reboot the machine

+ o Send SysRq commands

+ o Pause and resume the UML

You need the mconsole client (uml_mconsole) which is present in CVS (/tools/mconsole) in 2.4.5-9um and later, and will be in the RPM in 2.4.6.

You also need CONFIG_MCONSOLE (under 'General Setup') enabled in UML. 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 one 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:

```
     host% uml_mconsole debian
```

or

```
     host% uml_mconsole /home/jdike/.uml/debian/mconsole
```

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

+ o version

+ o halt

+ o reboot

+ o config

+ o remove

+ o sysrq

+ o help

+ o cad

+ o stop

+ o go

1 10 0. .1 1. . v ve er rs si io on n

This takes no arguments. It prints the UML version.

```
     (mconsole)  version
     OK Linux usermode 2.4.5-9um #1 Wed Jun 20 22:47:08 EDT 2001 i686
```

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 an interrupt to the UML. This is sometimes useful on SMP hosts, where there's a bug which causes signals to UML to be lost, often causing it to appear to hang. Sending such a UML the mconsole version command is a good way to 'wake it up' before networking has been enabled, as it does not do anything to the function of the UML.

1 10 0. .2 2. . h ha al lt t a an nd d r re eb bo oo ot t

These 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
```

1 10 0. .3 3. . c co on nf fi ig g

"config" adds a new device to the virtual machine. Currently the ubd and network drivers support this. 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

OK
(mconsole)  config eth1=mcast
OK
```

1 10 0. .4 4. . r re em mo ov ve e

"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
OK
(mconsole)  remove eth1
OK
```

1 10 0. .5 5. . s sy ys sr rq q

This 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/sysrq.txt in your favorite kernel tree to see what letters are valid and what they do.

1 10 0. .6 6. . h he el lp p

"help" returns a string listing the valid commands and what each one does.

1 10 0. .7 7. . c ca ad d

This invokes the Ctl-Alt-Del action on init. What exactly this ends up doing is up to /etc/inittab. Normally, it reboots the machine. With UML, this is usually not desired, so if a halt would be better, then find the section of inittab that looks like this

```
# What to do when CTRL-ALT-DEL is pressed.
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

and change the command to halt.

1 10 0. .8 8. . s st to op p

This puts the UML in a loop reading mconsole requests until a 'go' mconsole command is received. This is very useful for making backups of UML filesystems, as the UML can be stopped, then synced via 'sysrq s', so that everything is written to the filesystem. You can then copy the filesystem and then send the UML 'go' via mconsole.

Note that a UML running with more than one CPU will have problems after you send the 'stop' command, as only one CPU will be held in a mconsole loop and all others will continue as normal. This is a bug, and will be fixed.

1 10 0. .9 9. . g go o

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.

1 11 1. . K Ke er rn ne el l d de eb bu ug gg gi in ng g

N No ot te e: : The interface that makes debugging, as described here, possible
is present in 2.4.0-test6 kernels and later.

Since the user-mode kernel runs as a normal Linux process, it is possible to debug it with gdb almost like any other process. It is slightly different because the kernel's threads are already being ptraced for system call interception, so gdb can't ptrace them. However, a mechanism has been added to work around that problem.

In order to debug the kernel, you need build it from source. See ``Compiling the kernel and modules'' for information on doing that. Make sure that you enable CONFIG_DEBUGSYM and CONFIG_PT_PROXY during the config. These will compile the kernel with -g, and enable the ptrace proxy so that gdb works with UML, respectively.

1 11 1. .1 1. . S St ta ar rt ti in ng g t th he e k ke er rn ne el l
u un nd de er r g gd db b

You can have the kernel running under the control of gdb from the beginning by putting 'debug' on the command line. You will get an xterm with gdb running inside it. The kernel will send some commands to gdb which will leave it stopped at the beginning of start_kernel. At this point, you can get things going with 'next', 'step', or 'cont'.

There is a transcript of a debugging session here <debug-session.html> , with breakpoints being set in the scheduler and in an interrupt handler.

1 11 1. .2 2. . E Ex xa am mi in ni in ng g s sl le ee ep pi in ng g
p pr ro oc ce es ss se es s

Not every bug is evident in the currently running process. Sometimes, processes hang in the kernel when they shouldn't because they've deadlocked on a semaphore or something similar. In this case, when you ^C gdb and get a backtrace, you will see the idle thread, which isn't very relevant.

What you want is the stack of whatever process is sleeping when it shouldn't be. You need to figure out which process that is, which is generally fairly easy. Then you need to get its host process id, which you can do either by looking at ps on the host or at task.thread.extern_pid in gdb.

Now what you do is this:

+ o detach from the current thread

```
(UML gdb)  det
```

+ o attach to the thread you are interested in

```
(UML gdb)  att <host pid>
```

+ o look at its stack and anything else of interest

```
(UML gdb)  bt
```

Note that you can't do anything at this point that requires that a process execute, e.g. calling a function

+ o when you're done looking at that process, reattach to the current thread and continue it

```
(UML gdb)
att 1
```

```
(UML gdb)
c
```

Here, specifying any pid which is not the process id of a UML thread will cause gdb to reattach to the current thread. I commonly use 1, but any other invalid pid would work.

1 11 1. .3 3. . R Ru un nn ni in ng g d dd dd d o on n U UM ML L

ddd works on UML, but requires a special kludge. The process goes like this:

+ o Start ddd

```
host% ddd linux
```

+ o With ps, get the pid of the gdb that ddd started. You can ask the gdb to tell you, but for some reason that confuses things and causes a hang.

+ o run UML with 'debug=parent gdb-pid=<pid>' added to the command line - it will just sit there after you hit return

+ o type 'att 1' to the ddd gdb and you will see something like

```
0xa013dc51 in __kill ()

(gdb)
```

+ o At this point, type 'c', UML will boot up, and you can use ddd just as you do on any other process.

1 11 1. .4 4. . D De eb bu ug gg gi in ng g m mo od du ul le es s

gdb has support for debugging code which is dynamically loaded into the process. This support is what is needed to debug kernel modules under UML.

Using that support is somewhat complicated. You have to tell gdb what object file you just loaded into UML and where in memory it is. Then, it can read the symbol table, and figure out where all the symbols are from the load address that you provided. It gets more interesting when you load the module again (i.e. after an rmmod). You have to tell gdb to forget about all its symbols, including the main UML ones for some reason, then load then all back in again.

There's an easy way and a hard way to do this. The easy way is to use the umlgdb expect script written by Chandan Kudige. It basically automates the process for you.

First, you must tell it where your modules are. There is a list in the script that looks like this:

```
set MODULE_PATHS {
"fat" "/usr/src/uml/linux-2.4.18/fs/fat/fat.o"
"isofs" "/usr/src/uml/linux-2.4.18/fs/isofs/isofs.o"
"minix" "/usr/src/uml/linux-2.4.18/fs/minix/minix.o"
}
```

You change that to list the names and paths of the modules that you are going to debug. Then you run it from the toplevel directory of your UML pool and it basically tells you what to do:

```
            ******** GDB pid is 21903 ********
Start UML as: ./linux <kernel switches> debug gdb-pid=21903



GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux"...
(gdb) b sys_init_module
Breakpoint 1 at 0xa0011923: file module.c, line 349.
(gdb) att 1
```

After you run UML and it sits there doing nothing, you hit return at the 'att 1' and continue it:

```
Attaching to program: /home/jdike/linux/2.4/um/./linux, process 1
```

```
     0xa00f4221 in __kill ()
     (UML gdb)  c
     Continuing.
```

At this point, you debug normally. When you insmod something, the expect magic will kick in and you'll see something like:

```
 *** Module hostfs loaded ***
Breakpoint 1, sys_init_module (name_user=0x805abb0 "hostfs",
    mod_user=0x8070e00) at module.c:349
349             char *name, *n_name, *name_tmp = NULL;
(UML gdb)  finish
Run till exit from #0  sys_init_module (name_user=0x805abb0 "hostfs",
    mod_user=0x8070e00) at module.c:349
0xa00e2e23 in execute_syscall (r=0xa8140284) at syscall_kern.c:411
411             else res = EXECUTE_SYSCALL(syscall, regs);
Value returned is $1 = 0
(UML gdb)
p/x (int)module_list + module_list->size_of_struct

$2 = 0xa9021054
(UML gdb)  symbol-file ./linux
Load new symbol table from "./linux"? (y or n) y
Reading symbols from ./linux...
done.
(UML gdb)
add-symbol-file /home/jdike/linux/2.4/um/arch/um/fs/hostfs/hostfs.o 0xa9021054

add symbol table from file
"/home/jdike/linux/2.4/um/arch/um/fs/hostfs/hostfs.o" at
        .text_addr = 0xa9021054
 (y or n) y

Reading symbols from /home/jdike/linux/2.4/um/arch/um/fs/hostfs/hostfs.o...
done.
(UML gdb)  p *module_list
$1 = {size_of_struct = 84, next = 0xa0178720, name = 0xa9022de0 "hostfs",
```

```
  size = 9016, uc = {usecount = {counter = 0}, pad = 0}, flags = 1,
  nsyms = 57, ndeps = 0, syms = 0xa9023170, deps = 0x0, refs = 0x0,
  init = 0xa90221f0 <init_hostfs>, cleanup = 0xa902222c <exit_hostfs>,
  ex_table_start = 0x0, ex_table_end = 0x0, persist_start = 0x0,
  persist_end = 0x0, can_unload = 0, runsize = 0, kallsyms_start = 0x0,
  kallsyms_end = 0x0,
  archdata_start = 0x1b855 <Address 0x1b855 out of bounds>,
  archdata_end = 0xe5890000 <Address 0xe5890000 out of bounds>,
  kernel_data = 0xf689c35d <Address 0xf689c35d out of bounds>}
>> Finished loading symbols for hostfs ...
```

That's the easy way. It's highly recommended. The hard way is described below in case you're interested in what's going on.

Boot the kernel under the debugger and load the module with insmod or modprobe. With gdb, do:

```
     (UML gdb)  p module_list
```

This is a list of modules that have been loaded into the kernel, with the most recently loaded module first. Normally, the module you want is at module_list. If it's not, walk down the next links, looking at the name fields until find the module you want to debug. Take the address of that structure, and add module.size_of_struct (which in 2.4.10 kernels is 96 (0x60)) to it. Gdb can make this hard addition for you :-):

```
(UML gdb)
printf "%#x\n", (int)module_list module_list->size_of_struct
```

The offset from the module start occasionally changes (before 2.4.0, it was module.size_of_struct + 4), so it's a good idea to check the init and cleanup addresses once in a while, as describe below. Now do:

```
     (UML gdb)
     add-symbol-file /path/to/module/on/host that_address
```

Tell gdb you really want to do it, and you're in business.

If there's any doubt that you got the offset right, like breakpoints appear not to work, or they're appearing in the wrong place, you can check it by looking at the module structure. The init and cleanup fields should look like:

```
init = 0x588066b0 <init_hostfs>, cleanup = 0x588066c0 <exit_hostfs>
```

with no offsets on the symbol names. If the names are right, but they are offset, then the offset tells you how much you need to add to the address you gave to add-symbol-file.

When you want to load in a new version of the module, you need to get gdb to forget about the old one. The only way I've found to do that is to tell gdb to forget about all symbols that it knows about:

```
(UML gdb)  symbol-file
```

Then reload the symbols from the kernel binary:

```
(UML gdb)  symbol-file /path/to/kernel
```

and repeat the process above. You'll also need to re-enable breakpoints. They were disabled when you dumped all the symbols because gdb couldn't figure out where they should go.

1 11 1. .5 5. . A At tt ta ac ch hi in ng g g gd db b t to o t th he e
k ke er rn ne el l

If you don't have the kernel running under gdb, you can attach gdb to it later by sending the tracing thread a SIGUSR1. The first line of the console output identifies its pid:

```
tracing thread pid = 20093
```

When you send it the signal:

```
host% kill -USR1 20093
```

you will get an xterm with gdb running in it.

If you have the mconsole compiled into UML, then the mconsole client can be used to start gdb:

```
(mconsole)  (mconsole) config gdb=xterm
```

will fire up an xterm with gdb running in it.

1 11 1. .6 6. . U Us si in ng g a al lt te er rn na at te e d de eb bu ug gg ge er rs s

UML has support for attaching to an already running debugger rather than starting gdb itself. This is present in CVS as of 17 Apr 2001. I sent it to Alan for inclusion in the ac tree, and it will be in my 2.4.4 release.

This is useful when gdb is a subprocess of some UI, such as emacs or ddd. It can also be used to run debuggers other than gdb on UML. Below is an example of using strace as an alternate debugger.

To do this, you need to get the pid of the debugger and pass it in with the

If you are using gdb under some UI, then tell it to 'att 1', and you'll find yourself attached to UML.

If you are using something other than gdb as your debugger, then you'll need to get it to do the equivalent of 'att 1' if it doesn't do it automatically.

An example of an alternate debugger is strace. You can strace the actual kernel as follows:

+ o Run the following in a shell

```
host%
sh -c 'echo pid=$$; echo -n hit return; read x; exec strace -p 1 -o
```

strace.out'

+ o Run UML with 'debug' and 'gdb-pid=<pid>' with the pid printed out
   by the previous command

+ o Hit return in the shell, and UML will start running, and strace
   output will start accumulating in the output file.

   Note that this is different from running

```
host% strace ./linux
```

That will strace only the main UML thread, the tracing thread, which doesn't do any of the actual kernel work. It just oversees the virtual machine. In contrast, using strace as described above will show you the low-level activity of the virtual machine.

12. Kernel debugging examples

12.1. The case of the hung fsck

When booting up the kernel, fsck failed, and dropped me into a shell to fix things up. I ran fsck -y, which hung:

```
  Setting hostname uml                    [ OK ]
  Checking root filesystem
  /dev/fhd0 was not cleanly unmounted, check forced.
  Error reading block 86894 (Attempt to read block from filesystem resulted in
short read) while reading indirect blocks of inode 19780.

  /dev/fhd0: UNEXPECTED INCONSISTENCY; RUN fsck MANUALLY.
          (i.e., without -a or -p options)
  [ FAILED ]

  *** An error occurred during the file system check.
  *** Dropping you to a shell; the system will reboot
  *** when you leave the shell.
  Give root password for maintenance
  (or type Control-D for normal startup):

  [root@uml /root]# fsck -y /dev/fhd0
  fsck -y /dev/fhd0
  Parallelizing fsck version 1.14 (9-Jan-1999)
  e2fsck 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
  /dev/fhd0 contains a file system with errors, check forced.
  Pass 1: Checking inodes, blocks, and sizes
  Error reading block 86894 (Attempt to read block from filesystem resulted in
short read) while reading indirect blocks of inode 19780.  Ignore error? yes

  Inode 19780, i_blocks is 1548, should be 540.  Fix? yes

  Pass 2: Checking directory structure
  Error reading block 49405 (Attempt to read block from filesystem resulted in
short read).  Ignore error? yes

  Directory inode 11858, block 0, offset 0: directory corrupted
  Salvage? yes

  Missing '.' in directory inode 11858.
  Fix? yes

  Missing '..' in directory inode 11858.
  Fix? yes
```

The standard drill in this sort of situation is to fire up gdb on the
signal thread, which, in this case, was pid 1935. In another window,
I run gdb and attach pid 1935.

```
       ~/linux/2.3.26/um 1016: gdb linux
       GNU gdb 4.17.0.11 with Linux support
       Copyright 1998 Free Software Foundation, Inc.
       GDB is free software, covered by the GNU General Public License, and you
are
       welcome to change it and/or distribute copies of it under certain
conditions.
       Type "show copying" to see the conditions.
       There is absolutely no warranty for GDB.  Type "show warranty" for
details.
       This GDB was configured as "i386-redhat-linux"...

       (gdb) att 1935
       Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 1935
       0x100756d9 in __wait4 ()
```

Let's see what's currently running:

```
       (gdb) p current_task.pid
       $1 = 0
```

It's the idle thread, which means that fsck went to sleep for some
reason and never woke up.

Let's guess that the last process in the process list is fsck:

```
       (gdb) p current_task.prev_task.comm
       $13 = "fsck.ext2\000\000\000\000\000\000"
```

It is, so let's see what it thinks it's up to:

```
    (gdb) p current_task.prev_task.thread
    $14 = {extern_pid = 1980, tracing = 0, want_tracing = 0, forking = 0,
      kernel_stack_page = 0, signal_stack = 1342627840, syscall = {id = 4,
args = {
          3, 134973440, 1024, 0, 1024}, have_result = 0, result = 50590720},
      request = {op = 2, u = {exec = {ip = 1350467584, sp = 2952789424}, fork
= {
            regs = {1350467584, 2952789424, 0 <repeats 15 times>}, sigstack =
0,
            pid = 0}, switch_to = 0x507e8000, thread = {proc = 0x507e8000,
            arg = 0xaffffdb0, flags = 0, new_pid = 0}, input_request = {
            op = 1350467584, fd = -1342177872, proc = 0, pid = 0}}}}
```

The interesting things here are the fact that its .thread.syscall.id is __NR_write (see the big switch in arch/um/kernel/syscall_kern.c or the defines in include/asm-um/arch/unistd.h), and that it never returned. Also, its .request.op is OP_SWITCH (see arch/um/include/user_util.h). These mean that it went into a write, and, for some reason, called schedule().

The fact that it never returned from write means that its stack should be fairly interesting. Its pid is 1980 (.thread.extern_pid). That process is being ptraced by the signal thread, so it must be detached before gdb can attach it:

```
(gdb) call detach(1980)

Program received signal SIGSEGV, Segmentation fault.
<function called from gdb>
The program being debugged stopped while in a function called from GDB.
When the function (detach) is done executing, GDB will silently
stop (instead of continuing to evaluate the expression containing
the function call).
(gdb) call detach(1980)
$15 = 0
```

The first detach segfaults for some reason, and the second one succeeds.

Now I detach from the signal thread, attach to the fsck thread, and look at its stack:

```
     (gdb) det
     Detaching from program: /home/dike/linux/2.3.26/um/linux Pid 1935
     (gdb) att 1980
     Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 1980
     0x10070451 in __kill ()
     (gdb) bt
     #0  0x10070451 in __kill ()
     #1  0x10068ccd in usr1_pid (pid=1980) at process.c:30
     #2  0x1006a03f in _switch_to (prev=0x50072000, next=0x507e8000)
         at process_kern.c:156
     #3  0x1006a052 in switch_to (prev=0x50072000, next=0x507e8000,
last=0x50072000)
         at process_kern.c:161
     #4  0x10001d12 in schedule () at sched.c:777
     #5  0x1006a744 in __down (sem=0x507d241c) at semaphore.c:71
     #6  0x1006aa10 in __down_failed () at semaphore.c:157
     #7  0x1006c5d8 in segv_handler (sc=0x5006e940) at trap_user.c:174
     #8  0x1006c5ec in kern_segv_handler (sig=11) at trap_user.c:182
     #9  <signal handler called>
     #10 0x10155404 in errno ()
     #11 0x1006c0aa in segv (address=1342179328, is_write=2) at trap_kern.c:50
     #12 0x1006c5d8 in segv_handler (sc=0x5006eaf8) at trap_user.c:174
     #13 0x1006c5ec in kern_segv_handler (sig=11) at trap_user.c:182
     #14 <signal handler called>
     #15 0xc0fd in ?? ()
     #16 0x10016647 in sys_write (fd=3,
         buf=0x80b8800 <Address 0x80b8800 out of bounds>, count=1024)
         at read_write.c:159
     #17 0x1006d5b3 in execute_syscall (syscall=4, args=0x5006ef08)
         at syscall_kern.c:254
     #18 0x1006af87 in really_do_syscall (sig=12) at syscall_user.c:35
     #19 <signal handler called>
     #20 0x400dc8b0 in ?? ()
```

The interesting things here are :

+ o There are two segfaults on this stack (frames 9 and 14)

+ o The first faulting address (frame 11) is 0x50000800

```
(gdb) p (void *)1342179328
$16 = (void *) 0x50000800
```

The initial faulting address is interesting because it is on the idle thread's stack. I had been seeing the idle thread segfault for no apparent reason, and the cause looked like stack corruption. In hopes of catching the culprit in the act, I had turned off all protections to that stack while the idle thread wasn't running. This apparently tripped that trap.

However, the more immediate problem is that second segfault and I'm going to concentrate on that. First, I want to see where the fault happened, so I have to go look at the sigcontent struct in frame 8:

```
     (gdb) up
     #1  0x10068ccd in usr1_pid (pid=1980) at process.c:30
     30        kill(pid, SIGUSR1);
     (gdb)
     #2  0x1006a03f in _switch_to (prev=0x50072000, next=0x507e8000)
         at process_kern.c:156
     156       usr1_pid(getpid());
     (gdb)
     #3  0x1006a052 in switch_to (prev=0x50072000, next=0x507e8000,
last=0x50072000)
         at process_kern.c:161
     161       _switch_to(prev, next);
     (gdb)
     #4  0x10001d12 in schedule () at sched.c:777
     777             switch_to(prev, next, prev);
     (gdb)
     #5  0x1006a744 in __down (sem=0x507d241c) at semaphore.c:71
     71                      schedule();
     (gdb)
     #6  0x1006aa10 in __down_failed () at semaphore.c:157
     157     }
     (gdb)
     #7  0x1006c5d8 in segv_handler (sc=0x5006e940) at trap_user.c:174
     174       segv(sc->cr2, sc->err & 2);
     (gdb)
     #8  0x1006c5ec in kern_segv_handler (sig=11) at trap_user.c:182
     182       segv_handler(sc);
     (gdb) p *sc
     Cannot access memory at address 0x0.
```

That's not very useful, so I'll try a more manual method:

```
      (gdb) p *((struct sigcontext *) (&sig + 1))
      $19 = {gs = 0, __gsh = 0, fs = 0, __fsh = 0, es = 43, __esh = 0, ds = 43,
        __dsh = 0, edi = 1342179328, esi = 1350378548, ebp = 1342630440,
        esp = 1342630420, ebx = 1348150624, edx = 1280, ecx = 0, eax = 0,
        trapno = 14, err = 4, eip = 268480945, cs = 35, __csh = 0, eflags =
66118,
        esp_at_signal = 1342630420, ss = 43, __ssh = 0, fpstate = 0x0, oldmask
= 0,
        cr2 = 1280}
```

The ip is in handle_mm_fault:

```
      (gdb) p (void *)268480945
      $20 = (void *) 0x1000b1b1
      (gdb) i sym $20
      handle_mm_fault + 57 in section .text
```

Specifically, it's in pte_alloc:

```
      (gdb) i line *$20
      Line 124 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
         starts at address 0x1000b1b1 <handle_mm_fault+57>
         and ends at 0x1000b1b7 <handle_mm_fault+63>.
```

To find where in handle_mm_fault this is, I'll jump forward in the code until I see an address in that procedure:

```
      (gdb) i line *0x1000b1c0
      Line 126 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
         starts at address 0x1000b1b7 <handle_mm_fault+63>
         and ends at 0x1000b1c3 <handle_mm_fault+75>.
      (gdb) i line *0x1000b1d0
      Line 131 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
         starts at address 0x1000b1d0 <handle_mm_fault+88>
         and ends at 0x1000b1da <handle_mm_fault+98>.
      (gdb) i line *0x1000b1e0
      Line 61 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
         starts at address 0x1000b1da <handle_mm_fault+98>
         and ends at 0x1000b1e1 <handle_mm_fault+105>.
      (gdb) i line *0x1000b1f0
      Line 134 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
```

```
          starts at address 0x1000b1f0 <handle_mm_fault+120>
          and ends at 0x1000b200 <handle_mm_fault+136>.
       (gdb) i line *0x1000b200
       Line 135 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
          starts at address 0x1000b200 <handle_mm_fault+136>
          and ends at 0x1000b208 <handle_mm_fault+144>.
       (gdb) i line *0x1000b210
       Line 139 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
          starts at address 0x1000b210 <handle_mm_fault+152>
          and ends at 0x1000b219 <handle_mm_fault+161>.
       (gdb) i line *0x1000b220
       Line 1168 of "memory.c" starts at address 0x1000b21e
<handle_mm_fault+166>
          and ends at 0x1000b222 <handle_mm_fault+170>.
```

Something is apparently wrong with the page tables or vma_structs, so
lets go back to frame 11 and have a look at them:

```
  #11 0x1006c0aa in segv (address=1342179328, is_write=2) at trap_kern.c:50
  50        handle_mm_fault(current, vma, address, is_write);
  (gdb) call pgd_offset_proc(vma->vm_mm, address)
  $22 = (pgd_t *) 0x80a548c
```

That's pretty bogus. Page tables aren't supposed to be in process
text or data areas. Let's see what's in the vma:

```
       (gdb) p *vma
       $23 = {vm_mm = 0x507d2434, vm_start = 0, vm_end = 134512640,
         vm_next = 0x80a4f8c, vm_page_prot = {pgprot = 0}, vm_flags = 31200,
         vm_avl_height = 2058, vm_avl_left = 0x80a8c94, vm_avl_right =
0x80d1000,
         vm_next_share = 0xaffffdb0, vm_pprev_share = 0xaffffe63,
         vm_ops = 0xaffffe7a, vm_pgoff = 2952789626, vm_file = 0xafffffec,
         vm_private_data = 0x62}
       (gdb) p *vma.vm_mm
       $24 = {mmap = 0x507d2434, mmap_avl = 0x0, mmap_cache = 0x8048000,
         pgd = 0x80a4f8c, mm_users = {counter = 0}, mm_count = {counter =
134904288},
         map_count = 134909076, mmap_sem = {count = {counter = 135073792},
           sleepers = -1342177872, wait = {lock = <optimized out or zero
length>,
             task_list = {next = 0xaffffe63, prev = 0xaffffe7a},
             __magic = -1342177670, __creator = -1342177300}, __magic = 98},
         page_table_lock = {}, context = 138, start_code = 0, end_code = 0,
         start_data = 0, end_data = 0, start_brk = 0, brk = 0, start_stack = 0,
```

```
        arg_start = 0, arg_end = 0, env_start = 0, env_end = 0, rss =
1350381536,
        total_vm = 0, locked_vm = 0, def_flags = 0, cpu_vm_mask = 0, swap_cnt =
0,
        swap_address = 0, segments = 0x0}
```

This also pretty bogus. With all of the 0x80xxxxx and 0xaffffxxx addresses, this is looking like a stack was plonked down on top of these structures. Maybe it's a stack overflow from the next page:

```
(gdb) p vma
$25 = (struct vm_area_struct *) 0x507d2434
```

That's towards the lower quarter of the page, so that would have to have been pretty heavy stack overflow:

```
(gdb) x/100x $25
0x507d2434:     0x507d2434      0x00000000      0x08048000      0x080a4f8c
0x507d2444:     0x00000000      0x080a79e0      0x080a8c94      0x080d1000
0x507d2454:     0xaffffdb0      0xaffffe63      0xaffffe7a      0xaffffe7a
0x507d2464:     0xafffffec      0x00000062      0x0000008a      0x00000000
0x507d2474:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2484:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2494:     0x00000000      0x00000000      0x507d2fe0      0x00000000
0x507d24a4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d24b4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d24c4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d24d4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d24e4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d24f4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2504:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2514:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2524:     0x00000000      0x00000000      0x00000000      0x00000000
```

```
0x507d2534:     0x00000000      0x00000000      0x507d25dc      0x00000000
0x507d2544:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2554:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2564:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2574:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2584:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d2594:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d25a4:     0x00000000      0x00000000      0x00000000      0x00000000
0x507d25b4:     0x00000000      0x00000000      0x00000000      0x00000000
```

It's not stack overflow. The only "stack-like" piece of this data is the vma_struct itself.

At this point, I don't see any avenues to pursue, so I just have to admit that I have no idea what's going on. What I will do, though, is stick a trap on the segfault handler which will stop if it sees any writes to the idle thread's stack. That was the thing that happened first, and it may be that if I can catch it immediately, what's going on will be somewhat clearer.

1 12 2. .2 2. . E Ep pi is so od de e 2 2: : T Th he e c ca as se e o of f t th he e h hu un ng g f fs sc ck k

After setting a trap in the SEGV handler for accesses to the signal thread's stack, I reran the kernel.

fsck hung again, this time by hitting the trap:

```
Setting hostname uml                                  [ OK ]
Checking root filesystem
/dev/fhd0 contains a file system with errors, check forced.
Error reading block 86894 (Attempt to read block from filesystem resulted in
short read) while reading indirect blocks of inode 19780.
```

```
/dev/fhd0: UNEXPECTED INCONSISTENCY; RUN fsck MANUALLY.
        (i.e., without -a or -p options)
[ FAILED ]

*** An error occurred during the file system check.
*** Dropping you to a shell; the system will reboot
*** when you leave the shell.
Give root password for maintenance
(or type Control-D for normal startup):

[root@uml /root]# fsck -y /dev/fhd0
fsck -y /dev/fhd0
Parallelizing fsck version 1.14 (9-Jan-1999)
e2fsck 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
/dev/fhd0 contains a file system with errors, check forced.
Pass 1: Checking inodes, blocks, and sizes
Error reading block 86894 (Attempt to read block from filesystem resulted in
short read) while reading indirect blocks of inode 19780.  Ignore error? yes

Pass 2: Checking directory structure
Error reading block 49405 (Attempt to read block from filesystem resulted in
short read).  Ignore error? yes

Directory inode 11858, block 0, offset 0: directory corrupted
Salvage? yes

Missing '.' in directory inode 11858.
Fix? yes

Missing '..' in directory inode 11858.
Fix? yes

Untested (4127) [100fe44c]: trap_kern.c line 31
```

I need to get the signal thread to detach from pid 4127 so that I can attach to it with gdb. This is done by sending it a SIGUSR1, which is caught by the signal thread, which detaches the process:

```
kill -USR1 4127
```

Now I can run gdb on it:

```
~/linux/2.3.26/um 1034: gdb linux
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) att 4127
Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 4127
0x10075891 in __libc_nanosleep ()
```

The backtrace shows that it was in a write and that the fault address
(address in frame 3) is 0x50000800, which is right in the middle of
the signal thread's stack page:

```
     (gdb) bt
     #0  0x10075891 in __libc_nanosleep ()
     #1  0x1007584d in __sleep (seconds=1000000)
         at ../sysdeps/unix/sysv/linux/sleep.c:78
     #2  0x1006ce9a in stop () at user_util.c:191
     #3  0x1006bf88 in segv (address=1342179328, is_write=2) at trap_kern.c:31
     #4  0x1006c628 in segv_handler (sc=0x5006eaf8) at trap_user.c:174
     #5  0x1006c63c in kern_segv_handler (sig=11) at trap_user.c:182
     #6  <signal handler called>
     #7  0xc0fd in ?? ()
     #8  0x10016647 in sys_write (fd=3, buf=0x80b8800 "R.", count=1024)
         at read_write.c:159
     #9  0x1006d603 in execute_syscall (syscall=4, args=0x5006ef08)
         at syscall_kern.c:254
     #10 0x1006af87 in really_do_syscall (sig=12) at syscall_user.c:35
     #11 <signal handler called>
     #12 0x400dc8b0 in ?? ()
     #13 <signal handler called>
     #14 0x400dc8b0 in ?? ()
     #15 0x80545fd in ?? ()
     #16 0x804daae in ?? ()
     #17 0x8054334 in ?? ()
     #18 0x804d23e in ?? ()
     #19 0x8049632 in ?? ()
     #20 0x80491d2 in ?? ()
     #21 0x80596b5 in ?? ()
     (gdb) p (void *)1342179328
     $3 = (void *) 0x50000800
```

Going up the stack to the segv_handler frame and looking at where in the code the access happened shows that it happened near line 110 of block_dev.c:

```
(gdb) up
#1  0x1007584d in __sleep (seconds=1000000)
    at ../sysdeps/unix/sysv/linux/sleep.c:78
../sysdeps/unix/sysv/linux/sleep.c:78: No such file or directory.
(gdb)
#2  0x1006ce9a in stop () at user_util.c:191
191       while(1) sleep(1000000);
(gdb)
#3  0x1006bf88 in segv (address=1342179328, is_write=2) at trap_kern.c:31
31          KERN_UNTESTED();
(gdb)
#4  0x1006c628 in segv_handler (sc=0x5006eaf8) at trap_user.c:174
174       segv(sc->cr2, sc->err & 2);
(gdb) p *sc
$1 = {gs = 0, __gsh = 0, fs = 0, __fsh = 0, es = 43, __esh = 0, ds = 43,
  __dsh = 0, edi = 1342179328, esi = 134973440, ebp = 1342631484,
  esp = 1342630864, ebx = 256, edx = 0, ecx = 256, eax = 1024, trapno = 14,
  err = 6, eip = 268550834, cs = 35, __csh = 0, eflags = 66070,
  esp_at_signal = 1342630864, ss = 43, __ssh = 0, fpstate = 0x0, oldmask = 0,
  cr2 = 1342179328}
(gdb) p (void *)268550834
$2 = (void *) 0x1001c2b2
(gdb) i sym $2
block_write + 1090 in section .text
(gdb) i line *$2
Line 209 of "/home/dike/linux/2.3.26/um/include/asm/arch/string.h"
   starts at address 0x1001c2a1 <block_write+1073>
   and ends at 0x1001c2bf <block_write+1103>.
(gdb) i line *0x1001c2c0
Line 110 of "block_dev.c" starts at address 0x1001c2bf <block_write+1103>
   and ends at 0x1001c2e3 <block_write+1139>.
```

Looking at the source shows that the fault happened during a call to copy_to_user to copy the data into the kernel:

```
     107             count -= chars;
     108             copy_from_user(p,buf,chars);
     109             p += chars;
     110             buf += chars;
```

p is the pointer which must contain 0x50000800, since buf contains 0x80b8800 (frame 8 above). It is defined as:

```
                     p = offset + bh->b_data;
```

I need to figure out what bh is, and it just so happens that bh is passed as an argument to mark_buffer_uptodate and mark_buffer_dirty a few lines later, so I do a little disassembly:

```
(gdb) disas 0x1001c2bf 0x1001c2e0
Dump of assembler code from 0x1001c2bf to 0x1001c2d0:
0x1001c2bf <block_write+1103>:  addl   %eax,0xc(%ebp)
0x1001c2c2 <block_write+1106>:  movl   0xffffdd4(%ebp),%edx
0x1001c2c8 <block_write+1112>:  btsl   $0x0,0x18(%edx)
0x1001c2cd <block_write+1117>:  btsl   $0x1,0x18(%edx)
0x1001c2d2 <block_write+1122>:  sbbl   %ecx,%ecx
0x1001c2d4 <block_write+1124>:  testl  %ecx,%ecx
0x1001c2d6 <block_write+1126>:  jne    0x1001c2e3 <block_write+1139>
0x1001c2d8 <block_write+1128>:  pushl  $0x0
0x1001c2da <block_write+1130>:  pushl  %edx
0x1001c2db <block_write+1131>:  call   0x1001819c <__mark_buffer_dirty>
End of assembler dump.
```

At that point, bh is in %edx (address 0x1001c2da), which is calculated at 0x1001c2c2 as %ebp + 0xffffdd4, so I figure exactly what that is, taking %ebp from the sigcontext_struct above:

```
     (gdb) p (void *)1342631484
     $5 = (void *) 0x5006ee3c
     (gdb) p 0x5006ee3c+0xffffdd4
     $6 = 1342630928
     (gdb) p (void *)$6
     $7 = (void *) 0x5006ec10
```

```
      (gdb) p *((void **)$7)
      $8 = (void *) 0x50100200
```

Now, I look at the structure to see what's in it, and particularly, what its b_data field contains:

```
      (gdb) p *((struct buffer_head *)0x50100200)
      $13 = {b_next = 0x50289380, b_blocknr = 49405, b_size = 1024, b_list = 0,
        b_dev = 15872, b_count = {counter = 1}, b_rdev = 15872, b_state = 24,
        b_flushtime = 0, b_next_free = 0x501001a0, b_prev_free = 0x50100260,
        b_this_page = 0x501001a0, b_reqnext = 0x0, b_pprev = 0x507fcf58,
        b_data = 0x50000800 "", b_page = 0x50004000,
        b_end_io = 0x10017f60 <end_buffer_io_sync>, b_dev_id = 0x0,
        b_rsector = 98810, b_wait = {lock = <optimized out or zero length>,
          task_list = {next = 0x50100248, prev = 0x50100248}, __magic =
1343226448,
          __creator = 0}, b_kiobuf = 0x0}
```

The b_data field is indeed 0x50000800, so the question becomes how that happened. The rest of the structure looks fine, so this probably is not a case of data corruption. It happened on purpose somehow.

The b_page field is a pointer to the page_struct representing the 0x50000000 page. Looking at it shows the kernel's idea of the state of that page:

```
  (gdb) p *$13.b_page
  $17 = {list = {next = 0x50004a5c, prev = 0x100c5174}, mapping = 0x0,
    index = 0, next_hash = 0x0, count = {counter = 1}, flags = 132, lru = {
      next = 0x50008460, prev = 0x50019350}, wait = {
      lock = <optimized out or zero length>, task_list = {next = 0x50004024,
        prev = 0x50004024}, __magic = 1342193708, __creator = 0},
    pprev_hash = 0x0, buffers = 0x501002c0, virtual = 1342177280,
    zone = 0x100c5160}
```

Some sanity-checking: the virtual field shows the "virtual" address of this page, which in this kernel is the same as its "physical" address, and the page_struct itself should be mem_map[0], since it represents the first page of memory:

```
(gdb) p (void *)1342177280
$18 = (void *) 0x50000000
(gdb) p mem_map
$19 = (mem_map_t *) 0x50004000
```

These check out fine.

Now to check out the page_struct itself. In particular, the flags field shows whether the page is considered free or not:

```
(gdb) p (void *)132
$21 = (void *) 0x84
```

The "reserved" bit is the high bit, which is definitely not set, so the kernel considers the signal stack page to be free and available to be used.

At this point, I jump to conclusions and start looking at my early boot code, because that's where that page is supposed to be reserved.

In my setup_arch procedure, I have the following code which looks just fine:

```
bootmap_size = init_bootmem(start_pfn, end_pfn - start_pfn);
free_bootmem(__pa(low_physmem) + bootmap_size, high_physmem -
low_physmem);
```

Two stack pages have already been allocated, and low_physmem points to the third page, which is the beginning of free memory.
The init_bootmem call declares the entire memory to the boot memory manager, which marks it all reserved. The free_bootmem call frees up all of it, except for the first two pages. This looks correct to me.

So, I decide to see init_bootmem run and make sure that it is marking those first two pages as reserved. I never get that far.

Stepping into init_bootmem, and looking at bootmem_map before looking at what it contains shows the following:

```
(gdb) p bootmem_map
$3 = (void *) 0x50000000
```

Aha! The light dawns. That first page is doing double duty as a stack and as the boot memory map. The last thing that the boot memory manager does is to free the pages used by its memory map, so this page is getting freed even its marked as reserved.

The fix was to initialize the boot memory manager before allocating those two stack pages, and then allocate them through the boot memory manager. After doing this, and fixing a couple of subsequent buglets, the stack corruption problem disappeared.

1 13 3. . W Wh ha at t t to o d do o w wh he en n U UM ML L
d do oe es sn n' 't t w wo or rk k

1 13 3. .1 1. . S St tr ra an ng ge e c co om mp pi il la at ti io on n
e er rr ro or rs s w wh he en n y yo ou u b bu ui il ld d f fr ro om m
s so ou ur rc ce e

As of test11, it is necessary to have "ARCH=um" in the environment or on the make command line for all steps in building UML, including clean, distclean, or mrproper, config, menuconfig, or xconfig, dep, and linux. If you forget for any of them, the i386 build seems to contaminate the UML build. If this happens, start from scratch with

```
host%
make mrproper ARCH=um
```

and repeat the build process with ARCH=um on all the steps.

See ``Compiling the kernel and modules'' for more details.

Another cause of strange compilation errors is building UML in /usr/src/linux. If you do this, the first thing you need to do is clean up the mess you made. The /usr/src/linux/asm link will now point to /usr/src/linux/asm-um. Make it point back to /usr/src/linux/asm-i386. Then, move your UML pool someplace else and build it there. Also see below, where a more specific set of symptoms is described.

1 13 3. .3 3. . A A v va ar ri ie et ty y o of f p pa an ni ic cs s a an nd d h ha an ng gs s w wi it th h / /t tm mp p o on n a a r re ei is se er rf fs s f fi il le es sy ys s- -
t te em m

I saw this on reiserfs 3.5.21 and it seems to be fixed in 3.5.27. Panics preceded by

```
Detaching pid nnnn
```

are diagnostic of this problem. This is a reiserfs bug which causes a thread to occasionally read stale data from a mmapped page shared with another thread. The fix is to upgrade the filesystem or to have /tmp be an ext2 filesystem.

1 13 3. .4 4. . T Th he e c co om mp pi il le e f fa ai il ls s w wi it th h e er rr ro or rs s a ab bo ou ut t c co on nf fl li ic ct ti in ng g t ty yp pe es s f fo or r
’ ’o op pe en n’ ’, , ’ ’d du up p’ ’, , a an nd d ’ ’w wa ai it tp pi id d’ ’

This happens when you build in /usr/src/linux. The UML build makes the include/asm link point to include/asm-um. /usr/include/asm points to /usr/src/linux/include/asm, so when that link gets moved, files which need to include the asm-i386 versions of headers get the incompatible asm-um versions. The fix is to move the include/asm link back to include/asm-i386 and to do UML builds someplace else.

1 13 3. .5 5. . U UM ML L d do oe es sn n’ ’t t w wo or rk k w wh he en n / /t tm mp p i is s a an n N NF FS S f fi il le es sy ys st te em m

This seems to be a similar situation with the ReiserFS problem above. Some versions of NFS seems not to handle mmap correctly, which UML depends on. The workaround is have /tmp be a non-NFS directory.

1 13 3. .6 6. . U UM ML L h ha an ng gs s o on n b bo oo ot t w wh he en n c co om mp pi il le ed d w wi it th h g gp pr ro of f s su up pp po or rt t

If you build UML with gprof support and, early in the boot, it does this

```
kernel BUG at page_alloc.c:100!
```

you have a buggy gcc. You can work around the problem by removing UM_FASTCALL from CFLAGS in arch/um/Makefile-i386. This will open up another bug, but that one is fairly hard to reproduce.

1 13 3. .7 7. . s sy ys sl lo og gd d d di ie es s w wi it th h a a
S SI IG GT TE ER RM M o on n s st ta ar rt tu up p

The exact boot error depends on the distribution that you're booting, but Debian produces this:

```
/etc/rc2.d/S10sysklogd: line 49:    93 Terminated
start-stop-daemon --start --quiet --exec /sbin/syslogd -- $SYSLOGD
```

This is a syslogd bug. There's a race between a parent process installing a signal handler and its child sending the signal. See this uml-devel post <http://www.geocrawler.com/lists/3/Source-Forge/709/0/6612801> for the details.

1 13 3. .8 8. . T TU UN N/ /T TA AP P n ne et tw wo or rk ki in ng g
d do oe es sn n' 't t w wo or rk k o on n a a 2 2. .4 4 h ho os st t

There are a couple of problems which were <http://www.geocrawler.com/lists/3/SourceForge/597/0/> name="pointed out"> by Tim Robinson <timro at trkr dot net>

+ o It doesn't work on hosts running 2.4.7 (or thereabouts) or earlier. The fix is to upgrade to something more recent and then read the next item.

+ o If you see

```
File descriptor in bad state
```

when you bring up the device inside UML, you have a header mismatch between the original kernel and the upgraded one. Make /usr/src/linux point at the new headers. This will only be a problem if you build uml_net yourself.

1 13 3. .9 9. . Y Yo ou u c ca an n n ne et tw wo or rk k t to o t th he e
h ho os st t b bu ut t n no ot t t to o o ot th he er r m ma ac ch hi in ne es s
o on n t th he e
n ne et t

If you can connect to the host, and the host can connect to UML, but you cannot connect to any other machines, then you may need to enable IP Masquerading on the host. Usually this is only experienced when using private IP addresses (192.168.x.x or 10.x.x.x) for host/UML networking, rather than the public address space that your host is connected to. UML does not enable IP Masquerading, so you will need to create a static rule to enable it:

```
host%
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Replace eth0 with the interface that you use to talk to the rest of the world.

Documentation on IP Masquerading, and SNAT, can be found at www.netfilter.org <http://www.netfilter.org> .

If you can reach the local net, but not the outside Internet, then that is usually a routing problem. The UML needs a default route:

```
UML#
route add default gw gateway IP
```

The gateway IP can be any machine on the local net that knows how to reach the outside world. Usually, this is the host or the local network's gateway.

Occasionally, we hear from someone who can reach some machines, but not others on the same net, or who can reach some ports on other machines, but not others. These are usually caused by strange firewalling somewhere between the UML and the other box. You track this down by running tcpdump on every interface the packets travel over and see where they disappear. When you find a machine that takes the packets in, but does not send them onward, that's the culprit.

1 13 3. .1 10 0. . I I h ha av ve e n no o r ro oo ot t a an nd d I I w wa an nt t t to o s sc cr re ea am m

Thanks to Birgit Wahlich for telling me about this strange one. It turns out that there's a limit of six environment variables on the kernel command line. When that limit is reached or exceeded, argument processing stops, which means that the 'root=' argument that UML usually adds is not seen. So, the filesystem has no idea what the root device is, so it panics.

The fix is to put less stuff on the command line. Glomming all your setup variables into one is probably the best way to go.

1 13 3. .1 11 1. . U UM ML L b bu ui il ld d c co on nf fl li ic ct t b be et tw we ee en n p pt tr ra ac ce e. .h h a an nd d u uc co on nt te ex xt t. .h h

On some older systems, /usr/include/asm/ptrace.h and /usr/include/sys/ucontext.h define the same names. So, when they're included together, the defines from one completely mess up the parsing of the other, producing errors like:

```
     /usr/include/sys/ucontext.h:47: parse error before
     `10'
```

plus a pile of warnings.

This is a libc botch, which has since been fixed, and I don't see any way around it besides upgrading.

1 13 3. .1 12 2. . T Th he e U UM ML L B Bo og go oM Mi ip ps s i is s e ex xa ac ct tl ly y h ha al lf f t th he e h ho os st t' 's s B Bo og go oM Mi ip ps s

On i386 kernels, there are two ways of running the loop that is used to calculate the BogoMips rating, using the TSC if it's there or using a one-instruction loop. The TSC produces twice the BogoMips as the loop. UML uses the loop, since it has nothing resembling a TSC, and will get almost exactly the same BogoMips as a host using the loop. However, on a host with a TSC, its BogoMips will be double the loop BogoMips, and therefore double the UML BogoMips.

1 13 3. .1 13 3. . W Wh he en n y yo ou u r ru un n U UM ML L, , i it t i im mm me ed di ia at te el ly y s se eg gf fa au ul lt ts s

If the host is configured with the 2G/2G address space split, that's why. See ``UML on 2G/2G hosts'' for the details on getting UML to run on your host.

1 13 3. .1 14 4. . x xt te er rm ms s a ap pp pe ea ar r, , t th he en n i im mm me ed di ia at te el ly y d di is sa ap pp pe ea ar r

If you're running an up to date kernel with an old release of uml_utilities, the port-helper program will not work properly, so xterms will exit straight after they appear. The solution is to upgrade to the latest release of uml_utilities. Usually this problem occurs when you have installed a packaged release of UML then compiled your own development kernel without upgrading the uml_utilities from the source distribution.

1 13 3. .1 15 5. . A An ny y o ot th he er r p pa an ni ic c, , h ha an ng g, , o or r s st tr ra an ng ge e b be eh ha av vi io or r

If you're seeing truly strange behavior, such as hangs or panics that happen in random places, or you try running the debugger to see what's happening and it acts strangely, then it could be a problem in the host kernel. If you're not running a stock Linus or -ac kernel, then try that. An early version of the preemption patch and a 2.4.10 SuSE kernel have caused very strange problems in UML.

Otherwise, let me know about it. Send a message to one of the UML mailing lists - either the developer list - user-mode-linux-devel at lists dot sourceforge dot net (subscription info) or the user list - user-mode-linux-user at lists dot sourceforge do net (subscription info), whichever you prefer. Don't assume that everyone knows about it and that a fix is imminent.

If you want to be super-helpful, read ``Diagnosing Problems'' and follow the instructions contained therein.

1 14 4. . D Di ia ag gn no os si in ng g P Pr ro ob bl le em ms s

If you get UML to crash, hang, or otherwise misbehave, you should report this on one of the project mailing lists, either the developer list - user-mode-linux-devel at lists dot sourceforge dot net (subscription info) or the user list - user-mode-linux-user at lists dot sourceforge dot net (subscription info). When you do, it is likely that I will want more information. So, it would be helpful to read the stuff below, do whatever is applicable in your case, and report the results to the list.

For any diagnosis, you're going to need to build a debugging kernel. The binaries from this site aren't debuggable. If you haven't done this before, read about ``Compiling the kernel and modules''  and ``Kernel debugging'' UML first.

1 14 4. .1 1. . C Ca as se e 1 1 : : N No or rm ma al l k ke er rn ne el l p pa an ni ic cs s

The most common case is for a normal thread to panic. To debug this, you will need to run it under the debugger (add 'debug' to the command line). An xterm will start up with gdb running inside it. Continue it when it stops in start_kernel and make it crash. Now ^C gdb and

If the panic was a "Kernel mode fault", then there will be a segv frame on the stack and I'm going to want some more information. The stack might look something like this:

```
(UML gdb)  backtrace
#0  0x1009bf76 in __sigprocmask (how=1, set=0x5f347940, oset=0x0)
    at ../sysdeps/unix/sysv/linux/sigprocmask.c:49
#1  0x10091411 in change_sig (signal=10, on=1) at process.c:218
#2  0x10094785 in timer_handler (sig=26) at time_kern.c:32
#3  0x1009bf38 in __restore ()
    at ../sysdeps/unix/sysv/linux/i386/sigaction.c:125
#4  0x1009534c in segv (address=8, ip=268849158, is_write=2, is_user=0)
    at trap_kern.c:66
#5  0x10095c04 in segv_handler (sig=11) at trap_user.c:285
#6  0x1009bf38 in __restore ()
```

I'm going to want to see the symbol and line information for the value of ip in the segv frame. In this case, you would do the following:

```
(UML gdb)  i sym 268849158
```

and

```
(UML gdb)  i line *268849158
```

The reason for this is the __restore frame right above the segv_handler frame is hiding the frame that actually segfaulted. So, I have to get that information from the faulting ip.

1 14 4. .2 2. . C Ca as se e 2 2 : : T Tr ra ac ci in ng g t th hr re ea ad d

p pa an ni ic cs s

The less common and more painful case is when the tracing thread panics. In this case, the kernel debugger will be useless because it needs a healthy tracing thread in order to work. The first thing to do is get a backtrace from the tracing thread. This is done by figuring out what its pid is, firing up gdb, and attaching it to that pid. You can figure out the tracing thread pid by looking at the first line of the console output, which will look like this:

```
tracing thread pid = 15851
```

or by running ps on the host and finding the line that looks like this:

```
jdike 15851 4.5 0.4 132568 1104 pts/0 S 21:34 0:05 ./linux [(tracing thread)]
```

If the panic was 'segfault in signals', then follow the instructions above for collecting information about the location of the seg fault.

If the tracing thread flaked out all by itself, then send that backtrace in and wait for our crack debugging team to fix the problem.

1 14 4. .3 3. . C Ca as se e 3 3 : : T Tr ra ac ci in ng g t th hr re ea ad d p pa an ni ic cs s c ca au us se ed d b by y o ot th he er r t th hr re ea ad ds s

However, there are cases where the misbehavior of another thread caused the problem. The most common panic of this type is:

```
wait_for_stop failed to wait for <pid> to stop with <signal number>
```

In this case, you'll need to get a backtrace from the process mentioned in the panic, which is complicated by the fact that the kernel debugger is defunct and without some fancy footwork, another gdb can't attach to it. So, this is how the fancy footwork goes:

In a shell:

```
host% kill -STOP pid
```

Run gdb on the tracing thread as described in case 2 and do:

```
(host gdb)  call detach(pid)
```

If you get a segfault, do it again. It always works the second time.

Detach from the tracing thread and attach to that other thread:

```
(host gdb)  detach
```

```
(host gdb)  attach pid
```

If gdb hangs when attaching to that process, go back to a shell and do:

```
host%
kill -CONT pid
```

And then get the backtrace:

```
(host gdb)  backtrace
```

1 14 4. .4 4. . C Ca as se e 4 4 : : H Ha an ng gs s

Hangs seem to be fairly rare, but they sometimes happen. When a hang happens, we need a backtrace from the offending process. Run the kernel debugger as described in case 1 and get a backtrace. If the current process is not the idle thread, then send in the backtrace. You can tell that it's the idle thread if the stack looks like this:

```
#0  0x100b1401 in __libc_nanosleep ()
```

```
#1  0x100a2885 in idle_sleep (secs=10) at time.c:122
#2  0x100a546f in do_idle () at process_kern.c:445
#3  0x100a5508 in cpu_idle () at process_kern.c:471
#4  0x100ec18f in start_kernel () at init/main.c:592
#5  0x100a3e10 in start_kernel_proc (unused=0x0) at um_arch.c:71
#6  0x100a383f in signal_tramp (arg=0x100a3dd8) at trap_user.c:50
```

If this is the case, then some other process is at fault, and went to sleep when it shouldn't have. Run ps on the host and figure out which process should not have gone to sleep and stayed asleep. Then attach to it with gdb and get a backtrace as described in case 3.

1 15 5. . T Th ha an nk ks s

A number of people have helped this project in various ways, and this page gives recognition where recognition is due.

If you're listed here and you would prefer a real link on your name, or no link at all, instead of the despammed email address pseudo-link, let me know.

If you're not listed here and you think maybe you should be, please let me know that as well. I try to get everyone, but sometimes my bookkeeping lapses and I forget about contributions.

1 15 5. .1 1. . C Co od de e a an nd d D Do oc cu um me en nt ta at ti io on n

Rusty Russell <rusty at linuxcare.com.au> -

+ o wrote the HOWTO <http://user-mode-linux.sourceforge.net/UserModeLinux-HOWTO.html>

+ o prodded me into making this project official and putting it on SourceForge

+ o came up with the way cool UML logo <http://user-mode-linux.sourceforge.net/uml-small.png>

+ o redid the config process

Peter Moulder <reiter at netspace.net.au> - Fixed my config and build processes, and added some useful code to the block driver

Bill Stearns <wstearns at pobox.com> -

+ o HOWTO updates

+ o lots of bug reports

+ o lots of testing

+ o dedicated a box (uml.ists.dartmouth.edu) to support UML development

+ o wrote the mkrootfs script, which allows bootable filesystems of RPM-based distributions to be cranked out

+ o cranked out a large number of filesystems with said script

Jim Leu <jleu at mindspring.com> - Wrote the virtual ethernet driver and associated usermode tools

Lars Brinkhoff <http://lars.nocrew.org/> - Contributed the ptrace proxy from his own project <http://a386.nocrew.org/> to allow easier kernel debugging

Andrea Arcangeli <andrea at suse.de> - Redid some of the early boot code so that it would work on machines with Large File Support

Chris Emerson <http://www.chiark.greenend.org.uk/~cemerson/> - Did the first UML port to Linux/ppc

Harald Welte <laforge at gnumonks.org> - Wrote the multicast transport for the network driver

Jorgen Cederlof - Added special file support to hostfs

Greg Lonnon <glonnon at ridgerun dot com> - Changed the ubd driver to allow it to layer a COW file on a shared read-only filesystem and wrote the iomem emulation support

Henrik Nordstrom <http://hem.passagen.se/hno/> - Provided a variety of patches, fixes, and clues

Lennert Buytenhek - Contributed various patches, a rewrite of the network driver, the first implementation of the mconsole driver, and did the bulk of the work needed to get SMP working again.

Yon Uriarte - Fixed the TUN/TAP network backend while I slept.

Adam Heath - Made a bunch of nice cleanups to the initialization code, plus various other small patches.

Matt Zimmerman - Matt volunteered to be the UML Debian maintainer and is doing a real nice job of it. He also noticed and fixed a number of actually and potentially exploitable security holes in uml_net. Plus the occasional patch. I like patches.

James McMechan - James seems to have taken over maintenance of the ubd driver and is doing a nice job of it.

Chandan Kudige - wrote the umlgdb script which automates the reloading of module symbols.

Steve Schmidtke - wrote the UML slirp transport and hostaudio drivers, enabling UML processes to access audio devices on the host. He also submitted patches for the slip transport and lots of other things.

David Coulson <http://davidcoulson.net> -

+ o Set up the usermodelinux.org <http://usermodelinux.org> site, which is a great way of keeping the UML user community on top of UML goings-on.

+ o Site documentation and updates

+ o Nifty little UML management daemon UMLd <http://uml.openconsultancy.com/umld/>

+ o Lots of testing and bug reports

1 15 5. .2 2. . F Fl lu us sh hi in ng g o ou ut t b bu ug gs s

+ o Yuri Pudgorodsky

+ o Gerald Britton

+ o Ian Wehrman

+ o Gord Lamb

+ o Eugene Koontz

+ o John H. Hartman

+ o Anders Karlsson

+ o Daniel Phillips

+ o John Fremlin

+ o Rainer Burgstaller

+ o James Stevenson

+ o Matt Clay

+ o Cliff Jefferies

+ o Geoff Hoff

+ o Lennert Buytenhek

+ o Al Viro

+ o Frank Klingenhoefer

+ o Livio Baldini Soares

+ o Jon Burgess

+ o Petru Paler

+ o Paul

+ o Chris Reahard

+ o Sverker Nilsson

+ o Gong Su

+ o johan verrept

+ o Bjorn Eriksson

+ o Lorenzo Allegrucci

+ o Muli Ben-Yehuda

+ o David Mansfield

+ o Howard Goff

+ o Mike Anderson

+ o John Byrne

+ o Sapan J. Batia

+ o Iris Huang

+ o Jan Hudec

+ o Voluspa

1 15 5. .3 3. . B Bu ug gl le et ts s a an nd d c cl le ea an n- -u up ps s

+ o Dave Zarzycki

+ o Adam Lazur

+ o Boria Feigin

+ o Brian J. Murrell

+ o JS

+ o Roman Zippel

+ o Wil Cooley

+ o Ayelet Shemesh

+ o Will Dyson

+ o Sverker Nilsson

+ o dvorak

+ o v.naga srinivas

+ o Shlomi Fish

+ o Roger Binns

+ o johan verrept

+ o MrChuoi

+ o Peter Cleve

+ o Vincent Guffens

+ o Nathan Scott

+ o Patrick Caulfield

+ o jbearce

+ o Catalin Marinas

+ o Shane Spencer

+ o Zou Min

+ o Ryan Boder

+ o Lorenzo Colitti

+ o Gwendal Grignou

+ o Andre' Breiler

+ o Tsutomu Yasuda

1 15 5. .4 4. . C Ca as se e S St tu ud di ie es s

+ o Jon Wright

+ o William McEwan

+ o Michael Richardson

1 15 5. .5 5. . O Ot th he er r c co on nt tr ri ib bu ut ti io on ns s

Bill Carr <Bill.Carr at compaq.com> made the Red Hat mkrootfs script work with RH 6.2.

Michael Jennings <mikejen at hevanet.com> sent in some material which is now gracing the top of the index page <http://user-mode-linux.sourceforge.net/index.html> of this site.

SGI <http://www.sgi.com> (and more specifically Ralf Baechle <ralf at uni-koblenz.de>) gave me an account on oss.sgi.com <http://www.oss.sgi.com> . The bandwidth there made it possible to produce most of the filesystems available on the project download page.

Laurent Bonnaud <Laurent.Bonnaud at inpg.fr> took the old grotty Debian filesystem that I've been distributing and updated it to 2.2. It is now available by itself here.

Rik van Riel gave me some ftp space on ftp.nl.linux.org so I can make releases even when Sourceforge is broken.

Rodrigo de Castro looked at my broken pte code and told me what was wrong with it, letting me fix a long-standing (several weeks) and serious set of bugs.

Chris Reahard built a specialized root filesystem for running a DNS

server jailed inside UML. It's available from the download
<http://user-mode-linux.sourceforge.net/dl-sf.html> page in the Jail
Filesystems section.