User Mode Linux Installation Guide

Giorgos Kappes geokapp@gmail.com

Last Update: 11 September 2011

1. Introduction

User Mode Linux (UML) is a virtual Linux machine that runs on Linux. Actually, UML is not a virtual machine where you can install your preferred operating system. Instead it's a virtual operating system that can be run above Linux. More specifically, UML is a port of Linux to Linux.

The UML is running as a process on the host kernel. Like all other processes on the host, UML makes system calls to the host kernel in order to do its work. Unlike the other host processes, UML has its own system call interface for its processes to use. Thus, To the UML processes, the UML instance is a kernel (Figure 1).

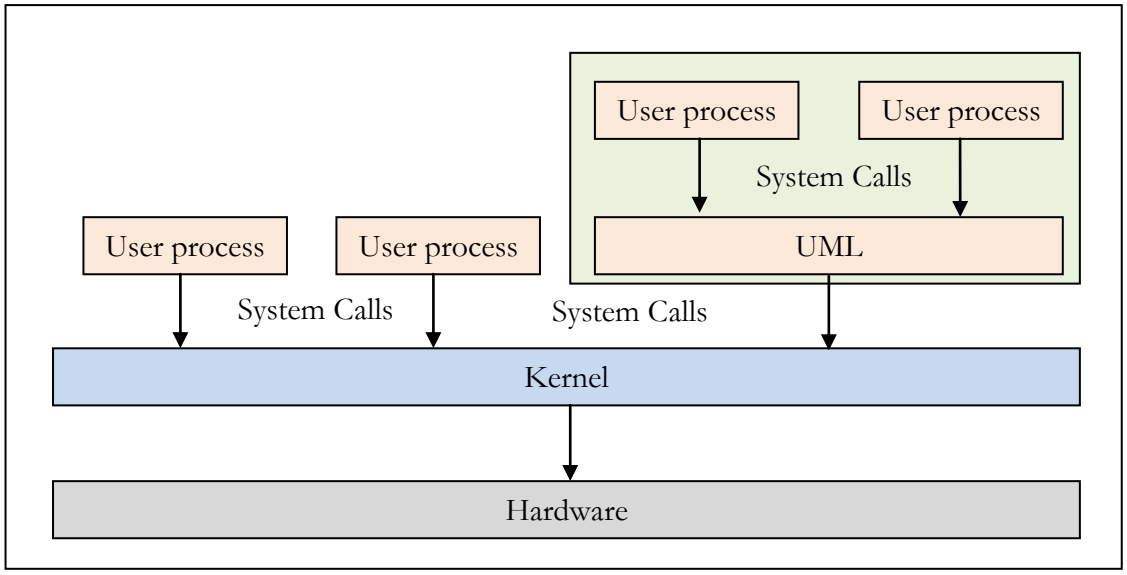

Figure 1 – UML is running as a process on the host kernel

As each UML guest is just a normal application running as a process in user space, this approach provides the user with a way of running multiple virtual Linux machines on a single piece of hardware.

2. Preparation

In order to successfully build and run user-mode Linux we need to download and install the following tools and utilities:

- GCC
- libstdc++ (In Debian/ Ubuntu: libstdc++6-dev)
- make
- ncurses (In Debian/ Ubuntu: libncurses5-dev)
- uml-utilities
- debootstrap

After installing the above tools, we need to download the kernel source tree from www.kernel.org. For this guide we are going with the latest stable Linux kernel which is version 3.0. We can download and extract a Linux kernel source tree with the following commands:

```
host$ mkdir /home/giorgos/uml
host$ mkdir /home/giorgos/uml/linux-src
host$ cd /home/giorgos/uml/linux-src
host$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.0.tar.gz
host$ tar xvf linux-3.0.tar.gz
```

3. Kernel Configuration and build

We are now ready to configure our new kernel. We start with the UML default configuration, noting that it is important to define the kernel architecture as 'um' (user-mode Linux):

```
host$ cd linux-3.0
host$ make mrproper ARCH=um
host$ make defconfig ARCH=um
```

Now, we can select additional kernel options by running:

```
host$ make menuconfig ARCH=um
```

Below, is a list of options needed by UML guest in order to work properly. Of course, you can adjust the kernel configuration to your own needs. It's important to mention that we have selected all the options below as build in, not as modules.

```
UML-specific options →
        Host processor type and features →
                Processor family → [choose your processor family]
                Generic x86 support – disabled

Block devices →
        Virtual block device – enabled
        Loopback device support – enabled
        Network block device support – enabled

Character devices →
        stderr console – enabled
        virtual serial line – enabled
        port channel support – enabled
        pty channel support – enabled
        tty channel support – enabled
        xterm channel support – enabled
        (xterm) default console channel initialization
        (pts) defaults serial line channel initialization

Networking support →
        UML Network devices →
                TUN/TAP transport – enabled

Network device support →
        Universal TAN/TUP device driver support – enabled
```

In this guide we chose some additional kernel options as build-in. These options are the following:

```
Networking support →
        Networking options →
                IP: kernel level autoconfiguration – enabled
                IP: DHCP support – enabled
                IP: BOOTP support – enabled
                IP: RARP support – enabled

File Systems →
        The extended 4 (ext4) filesystem – enabled
                Ext4 POSIX Access Control Lists – enabled
        Kernel automounter version 4 support – enabled
        Network File Systems →
                NFS client support – enabled
                NFS client support for NFS version 3 – enabled
                NFS client support for the NFSv3 ACL protocol
                expansion – enabed
                NFS client support for NFS version 4 – enabled
                Root filesystem on NFS – enabled
                Use the new idmapper upcall routine – enabled
                NFS server support – enabled
                NFS server support for NFS version 3 – enabled
                NFS server support for the NFSv3 ACL protocol
                expansion – enabed
                NFS server support for NFS version 4 – enabled
```

The most of the above options provide NFS client/server support and you can omit them if you don't want to use NFS.

It's time to build our brand new kernel by running:

```
host$ make ARCH=um
host$ cp linux /home/giorgos/uml/.
```

During the build we will be asked to specify some additional options for the UML kernel. In this guide we choose the default values, but you can adjust the kernel configuration to your own needs.

When the build finishes, we will have a UML binary called "linux" which we eventually copy in the '/home/giorgos/uml' directory. UML is both a Linux kernel and a Linux process. As a Linux process, it can be run just like any other executable on the system.

4. File system creation

In order for our new UML kernel to be functional we have to supply it with a root file system. We will create a disk image which will contain the root file system for our guest UML. This image will act as a hard disk for our guest system. In order to create an empty disk image we will use the 'dd' (man 1 dd) utility:

```
host$ cd /home/giorgos/uml
host$ dd if=/dev/zero of=/home/giorgos/uml/root-fs bs=1024K
count=1024
```

With the above command we created an empty disk image with 1GB free space. Now, we must initialize the file system inside of the image file. We will use ext4 as our root file system:

```
host$ mkfs.ext4 /home/giorgos/uml/root-fs
```

At this point we have an empty disk image formatted with ext4 file system. The next step is to populate the file system with a copy of the host's root or with a base system of our favorite Linux distribution. In this guide we choose to install a Debian Squeeze base system. To achieve this, we first mount the disk image to a new directory on the host and then we use the 'debootstrap' (man 8 debootstrap) utility to populate the image with a Debian Squeeze base system:

```
host$ su
```

```
host$ mkdir /mnt/root-fs
host$ mount –o loop /home/giorgos/uml/root-fs /mnt/root-fs
host$ debootstrap --arch i386 squeeze /mnt/root-fs
http://ftp.us.debian.org/debian
```

In the above example we chose i386 as the architecture of our new system. We can specify a different architecture by changing the argument after 'arch' parameter. However, it's important to select the same architecture with the UML kernel we built earlier.

5. Guest system configuration

After populating the disk image with a base Debian system we need to edit some files in order to have our guest system in a valid configuration:

a) /etc/fstab

The '/etc/fstab' file is a system configuration file that lists all available disks and disk partitions and indicates how they are to be initialized or otherwise integrated into the overall system's file system. In this guide we mount the virtual disk device '/dev/ubd0' on the root specifying 'ext4' as the file system. Also, we mount the proc file system on '/proc' directory:

```
/dev/ubd0   /      ext4   defaults      0 1
proc        /proc  proc   defaults      0 0
```

b) /etc/hostname

The hostname file contains the host name of the guest system. We add the name:

```
uml1
```

c) /etc/network/interfaces

This file contains network interface configuration information for the ifup(8) and ifdown(8) commands. This is where you configure how your system is connected to the network. At this point we will add the loopback interface to this file. Later on this guide we will add some more network interfaces to provide a fully functional networking on the UML guest. So, we open this file with an editor and we add the following values:

```
# This file describes the network interfaces available on your
system
# and how to activate them. For more information, see
interfaces(5).
```

```
#
# The loopback network interface
#
auto lo
iface lo inet loopback
```

Note that the lines beginning with # are comments.

d) /etc/securetty

The '/etc/securetty' file allows you to specify which TTY devices the root user is allowed to login on. The '/etc/securetty' file is read by the login program usually '/bin/login'. Its format is a list of the tty devices' names allowed, and for all others that are commented out or do not appear in this file, root login is disallowed. We append this file with the following values:

```
tty0
ttys/0
```

e) /dev/

The '/dev' directory contains the special device files for all the available devices. If the virtual block device 'ubd0' does not exist there we have to create it manually. We run the following commands as root:

```
host$ chroot /mnt/root-fs
host$ mknod --mode=660 ubd0 b 98 0
host$ chown root:disk ubd0
host$ exit
```

f) /etc/inittab

The '/etc/inittab' file describes which processes are started at boot up and during normal operation. To login immediately after the boot messages we commend out the following lines:

```
# 2:23:respawn:/sbin/getty 38400 tty2
# 3:23:respawn:/sbin/getty 38400 tty3
# 4:23:respawn:/sbin/getty 38400 tty4
# 5:23:respawn:/sbin/getty 38400 tty5
# 6:23:respawn:/sbin/getty 38400 tty6
```

And we modify 'tty1' to say 'tty0' in the line beginning with '1:'

```
1:2345:respawn:/sbin/getty 38400 tty0
```

Now, it's time to add a new user to the UML guest. To do this we run the following commands as root:

```
host$ chroot /mnt/root-fs
host$ adduser uml
host$ exit
```

Instead of using the tool 'adduser', we can add a new user by hand, adding an entry to '/etc/passwd' and '/etc/shadow' files. After adding the new user we can specify a password for the root user. We run the following commands as root:

```
host$ chroot /mnt/root-fs
host$ passwd
host$ exit
```

After completing the above steps, we can boot our new UML guest. However, it's important to unmount the guest's root file system from the host before booting it, otherwise it will be corrupted.

```
host$ umount /mnt/root-fs
host$ cd /home./giorgos/uml
host$ ./linux ubd0=rootf-fs mem=128MB
```

In the above example we assume that we use a virtual disk image named 'root-fs' as the UML guest root file system. With the 'mem=128MB' option we specify the amount of RAM that the UML guest will believe it has. Actually this command doesn't allocate 128MB of physical RAM on the host. Instead, it creates a 128MB sparse file on the host. Being sparse, this file will occupy very little disk space until data starts being written to it. As the UML instance uses its memory, it will start putting data in the memory backed by this file. As that happens, the host will start allocating memory to hold that data. Since the file is fixed in size, the UML instance is limited to that amount of memory.

6. Networking configuration

In this section we will take a look at networking with UML. There are a variety of ways to provide networking on the UML guests including SLIP, TUN/TAP and Ethertap. In this guide we will use TUN/TAP. TUN and TAP are virtual network kernel devices and they supported entirely in software. TAP simulates an Ethernet device and it operates with layer 2 packets such as Ethernet frames. TUN simulates a network layer device and it operates with layer 3 packets such as IP packets. Packets sent by an operating system via a TUN/TAP device are delivered to a user-space program that attaches itself to the device. A user-space program may also pass packets into a TUN/TAP device. In this case TUN/TAP device delivers these packets to the operating system network stack thus emulating their reception from an external source.

6.1. Configuring TUN/TAP device

In order to use TUN/TAP device make sure that the TUN/TAP module is included in your kernel. If you use a recent kernel then the needed TUN/TAP module is already built. Otherwise, recompile your kernel including the configuration option: *CONFIG_TUN (Network device support -> Universal TUN/TAP device driver support)*. To see if the TUN/TAP module is running run as root:

```
host$ lsmod | grep tun
```

If the above command gives an output then the module is running. Otherwise, you have to manually load the TUN/TAP module. To do so, run as root:

```
host$ modprobe tun
```

or add a 'tun' entry on '/etc/modules' if you want the module to run automatically when the system boots.

To access the TUN/TAP device we will use the '/dev/net/tun' device node. If this file doesn't exists in your system you have to create it manually with the following commands as root:

```
host$ mkdir /dev/net (If it doesn't exist already)
host$ mknod /dev/net/tun c 10 200
```

The next step is to provide the appropriate access rights to '/dev/net/tun' device node. Generally, we don't like unprivileged users to use the TUN/TAP device. In this context, we will create a group named 'uml-net' and we will give read/write permissions to this file. Then, we will add the users that we would like to have access on TUN/TAP device in the newly created group. To create the new group we add a line in '/etc/group':

```
uml-net:x:110:giorgos
```

In the above example, 110 is the group id of the new group. This number must be unique among the other groups' ids, so be careful while you choosing this. "giorgos" is the username of the user we would like to include in this group. Now, we are ready to specify TUN/TAP device node permissions. We run the following commands as root:

```
host$ chmod 660 /dev/net/tun
host$ chgrp uml-net /dev/net/tun
```

6.2. Environment parameters and bridging

Having configured the TUN/TAP device it's time to specify our network parameters. At first we have to make some modifications on the host network parameters. In order to provide networking on the UML guest, we will create a virtual bridge on the host. Then, we will connect there all networking interfaces. A bridge is a way to connect two Ethernet segments together in a protocol independent way. Packets are forwarded based on Ethernet address, rather than IP address (like a router). Since forwarding is done at Layer 2, all protocols can go transparently through a bridge. Of course there are many other ways to provide networking on the UML guest without bridging, but in this guide we prefer to use bridging. Let's begin by viewing our initial network configuration running the command 'ifconfig' as root:

```
eth0    Link encap:Ethernet  HWaddr 00:00:e8:7d:f5:25
        inet addr:192.168.1.200  Bcast:192.168.1.255
        Mask:255.255.255.0
        inet6 addr: fe80::200:e8ff:fe7d:f525/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500
        Metric:1
        RX packets:10108 errors:0 dropped:0 overruns:0 frame:0
        TX packets:330 errors:0 dropped:0 overruns:0 carrier:0
        collisions:1 txqueuelen:1000
        RX bytes:1853908 (1.7 MiB)  TX bytes:35865 (35.0 KiB)
        Interrupt:10 Base address:0xec00

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:89 errors:0 dropped:0 overruns:0 frame:0
        TX packets:89 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:8124 (7.9 KiB)  TX bytes:8124 (7.9 KiB)
```

As we see in the output of 'ifconfig' our host has one network interface named 'eth0' configured with IP address 192.168.1.200. At this point we have to collect some additional information about our network. Thus, the gateway of our home network is 192.168.1.1, the network address is 192.168.1.0, the netmask is 255.255.255.0 and the broadcasting address is 192.168.1.255.

The easiest way to connect our UML guests on the network is to directly connect them in our home network using a bridge. In this approach, the UML guests will need some free IPs from our home network. If we have no free IPs left, or if we are not allowed to use other IPs of the network we are connected, we can create a virtual network in our host system. In this second approach we will use a new network for our UML, guests

different from the physical one (e.g. 192.168.2.0/24). Then, we will assign an IP of the new network to the bridge (e.g. 192.168.2.1). Thus, the bridge will act as a gateway for our UML guests which can take any IP from the new network (192.168.2.0/24). However, for the UML guests to be able to communicate with other machines on our physical network, we must configure routing on both the host and the guests. We will see this approach later on subsection 6.2.

A step that has to be done on both cases is to configure the UML guest's 'resolv.conf' file. The 'resolv.conf' file is the resolver configuration file. It is being used to configure client side access to the Internet Domain Name System (DNS). This file defines which name servers to use. Here, we simply copy the host 'resolv.conf' to the UML guest.

6.3. Connecting UML guests on the physical network

In this approach we will connect the UML guests in our physical network using a bridge. Thus, if our home network is 192.168.1.0/24, the UML guests will take IPs from the range 192.168.1.0/24. The first step is to create a network bridge. By creating a network bridge a new virtual network interface is generated. This interface must be configured with the options that the 'eth0' interface currently has and then any other interface (like 'eth0') can be connected directly in this bridge (figure 2).

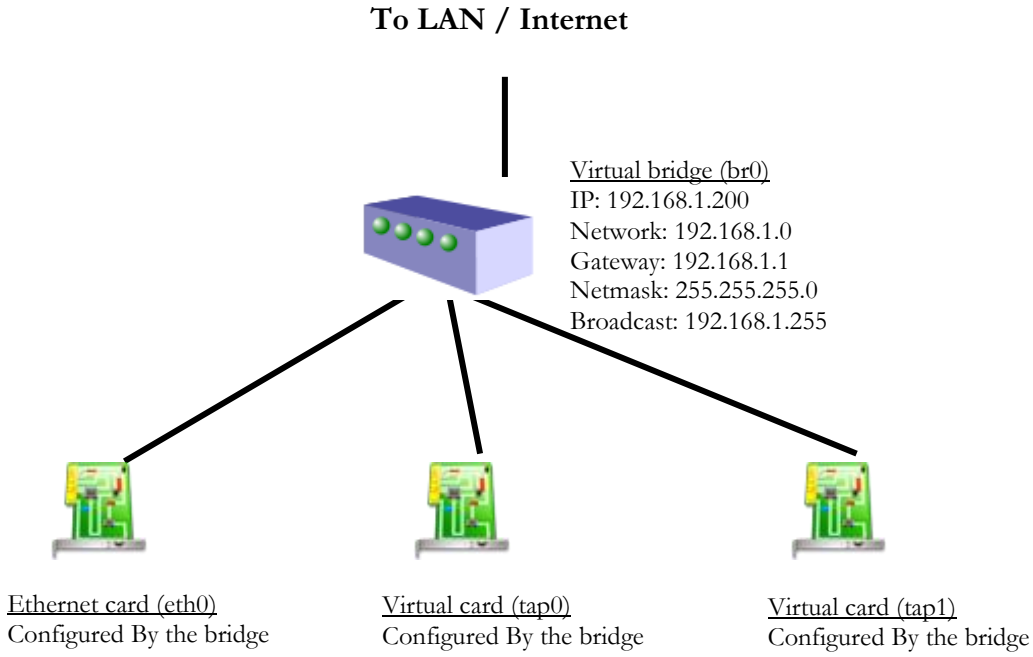

Figure 2 – Connecting all the network interfaces on a virtual network bridge

To achieve the configuration shown in figure 2 we have to install the bridging utilities by running as root:

```
host$ apt-get install bridge-utils
```

Then, we must modify the '/etc/network/interfaces' configuration file. An example is given below:

```
# This file describes the network interfaces available on your
system
# and how to activate them. For more information, see
interfaces(5).

#
# The loopback network interface
#
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet manual

#
# The tap0 interface
#
auto tap0
iface tap0 inet manual
        tunctl_user giorgos

#
# The bridge interface
#
auto br0
iface br0 inet static
        address 192.168.1.200
        network 192.168.1.0
        netmask 255.255.255.0
        broadcast 192.168.1.255
        gateway 192.168.1.1
        bridge_ports eth0 tap0
        bridge_fd 9
        bridge_hello 2
        bridge_maxage 12
        bridge_stp off
```

As shown in the above example, we configure 'eth0' as 'inet manual'. This option will allow the bridge to manually setup the 'eth0' interfaces avoiding conflicts with other configuration utilities like 'network-manager'. Also, we specify a new interface named 'tap0'. This is the virtual TUN/TAP interface that the UML guest will connect on. Similarly with 'eth0' we configure 'tap0' as 'inet manual'. Generally, when we use a network bridge we mustn't specify IP addresses on the other interfaces that will be

connected on the bridge. Instead, the bridge has the responsibility to configure the connected interfaces. With the 'tunctl_user' option we specify which users can start and use the 'tap0' interface. The last entry of the network configuration file is 'br0'. The 'br0' is our virtual network bridge and we configure it with the options that the 'eth0' interface previously had. Furthermore, we specify the network interfaces that we want to connect on the bridge with the option: 'bridge_ports eth0 tap0'. Here, we connect 'eth0' and 'tap0'. Finally we specify some additional options for our bridge. It's important to mention here that in this guide we are using static IP configuration but this is not restrictive. You can easily use DHCP instead.

At this point, the needed configuration on the host is completed. Now it's time to make some configuration on the UML guest:

```
host$ mount –o loop /home/giorgos/uml/root-fs /mnt/root-fs
host$ chroot /mnt/root-fs
host$ vi /etc/network/interfaces
host$ exit
host$ cp /etc/resolv.conf /mnt/root-fs/etc/resolv.conf
host$ umount /mnt/root-fs
```

We add in the '/etc/network/interfaces' file the following configurations:

```
auto eth0
iface eth0 inet static
        address 192.168.1.21
        netmask 255.255.255.0
        broadcast 192.168.1.255
        gateway 192.168.1.1
```

As we can see, we specify a network interface named 'eth0' and we have configured it with a static IP. Again, you can use DHCP if you prefer it. If you specify a static IP on 'eth0' then you must choose an available IP address of your network. Here we choose 192.168.1.21. Also, we specify the correct network, netmask, broadcast and gateway values.

After completing the above steps, the UML guest will have a configured network connection. To check this we run the UML guest again:

```
host$ cd /home./giorgos/uml
host$ ./linux ubd0=root-fs mem=128M eth0=tuntap,tap0,,
```

At this time, we provide a network device to the virtual machine. The general format is

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

For example, a virtual ethernet device may be attached to a host TUN/TAP device as follows:

eth0=tuntap,tap0,fe:fd:0:0:0:1,192.168.1.12

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. If we leave the ethernet and the IP address fields blank the system will assign them automatically. It's important to mention that in the case where we are using a network bridge we must not assign an IP address to the TUN/TAP device.

To check that everything is working we can try to ping our network gateway, and some internet hosts from our UML guest. Also we can try to ping the UML guest from the host.

6.4. Connecting UML guests on a new network

If we cannot use IP addresses of the physical network that our host system is connected, the ideal solution is to create a new network for our UML guests. Thus, let's say that our host system is connected on 192.168.1.0/24 network via the interface 'eth0'. The main idea behind this approach is to create a new virtual bridge and assign it an IP address of a different network. Then all the UML guests will connect to this newly created network having the host's bridge as their gateway. However, in order for the UML guests to reach the physical network 192.168.1.0/24 where the host is connected in, some routing must be done on the host between the virtual bridge and its primary network interface. Figure 3 summarizes this setup.

Let's assume that we want to use the network 192.168.2.0/24 to connect our UML guests. We will assign to the bridge the IP address 192.168.2.1 as shown in figure 3. To create the new bridge and assign it the new IP we run as root the following commands on the host:

```
host$ brctl addbr br0
host$ ifconfig br0 192.168.2.1 netmask 255.255.255.0 up
host$ brctl stp br0 off
host$ brctl setfd br0 1
host$ brctl sethello br0 1
```

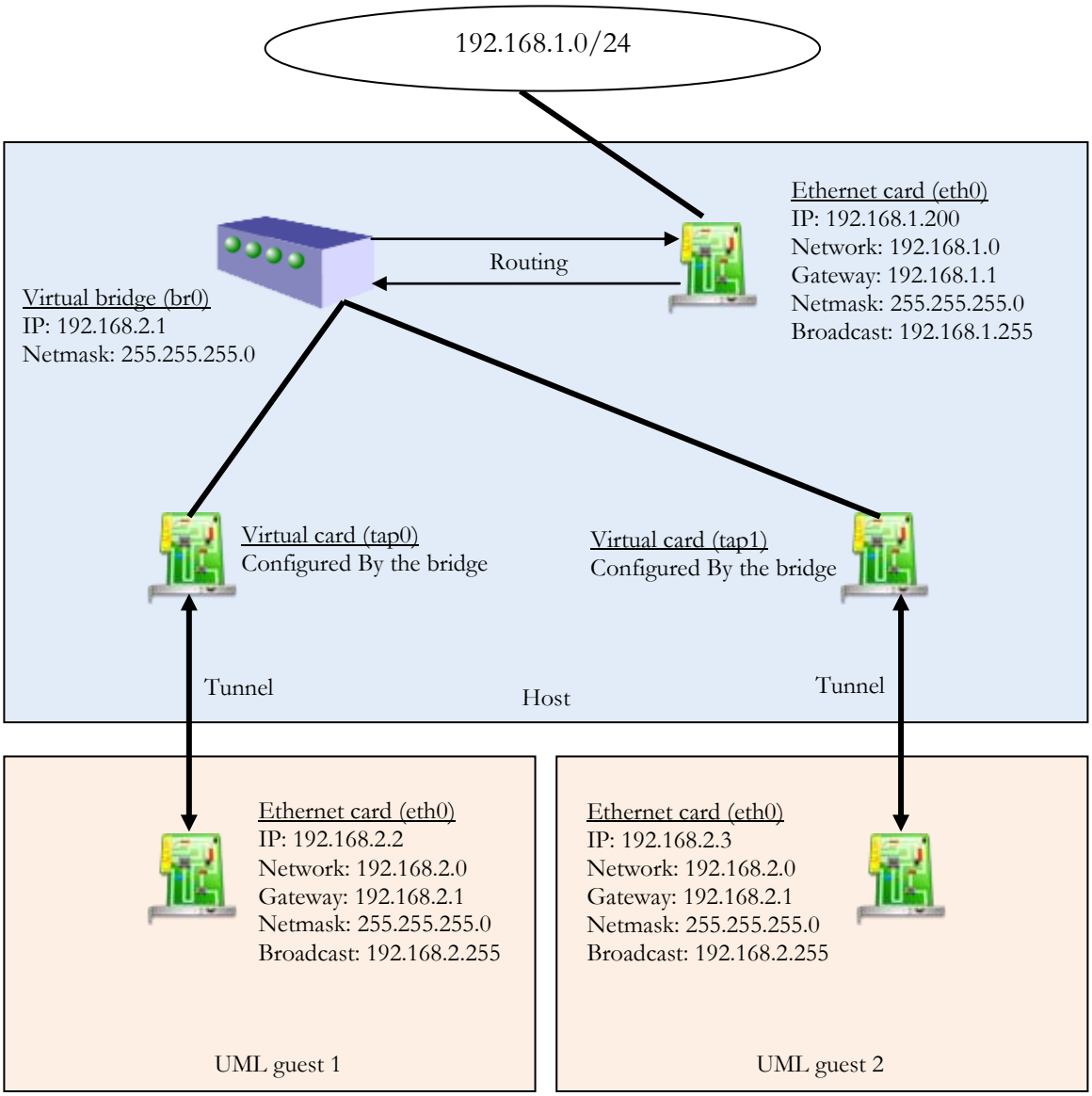

Figure 3 – Connecting the UML guests on a different network.

Of course 'bridge-utils' must be already installed on the system for the above to work. Now, 'ifconfig' gives us the following results:

```
br0     Link encap:Ethernet  HWaddr 56:f5:9d:41:8f:97
        inet addr:192.168.2.1  Bcast:192.168.2.255
        Mask:255.255.255.0
        UP BROADCAST MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0    Link encap:Ethernet  HWaddr 00:00:e8:7d:f5:25
```

```
          inet addr:192.168.1.200  Bcast:192.168.1.255
          Mask:255.255.255.0
          inet6 addr: fe80::200:e8ff:fe7d:f525/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500
          Metric:1
          RX packets:68 errors:0 dropped:0 overruns:0 frame:0
          TX packets:71 errors:0 dropped:0 overruns:0 carrier:0
          collisions:2 txqueuelen:1000
          RX bytes:7335 (7.1 KiB)  TX bytes:8917 (8.7 KiB)
          Interrupt:10 Base address:0xec00

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:86 errors:0 dropped:0 overruns:0 frame:0
          TX packets:86 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:7968 (7.7 KiB)  TX bytes:7968 (7.7 KiB)
```

Having created the bridge, it's time to create a new TUN/TAP interface named 'tap0'. We run as root the following commands on the host:

```
host$ tunctl -u giorgos -t tap0
host$ ifconfig tap0 0.0.0.0 up
```

Note that we have not assigned an IP address to the interface 'tap0' because we will connect it on the bridge which we created earlier. Also, note the '–u giorgos' parameter of the command 'tunctl'. With this parameter we specify to the system the owner of the new TUN/TAP device. Now we are ready to connect the new virtual interface to the bridge. To achieve this we run the following command as root on the host:

```
host$ brctl addif br0 tap0
```

The above steps must be done every time when the host boots. To make them permanent we can add some configuration on the '/etc/network/interfaces':

```
# This file describes the network interfaces available on your
system
# and how to activate them. For more information, see
interfaces(5).

#
# The loopback network interface
#
auto lo
iface lo inet loopback

#
# The primary network interface
```

```
#
allow-hotplug eth0
iface eth0 inet static
        address 192.168.1.200
        netmask 255.255.255.0
        network 192.168.1.0
        broadcast 192.168.1.255
        gateway 192.168.1.1

auto tap0
iface tap0 inet manual
        tunctl_user giorgos

auto br0
iface br0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        bridge_ports tap0
        bridge_fd 9
        bridge_hello 2
        bridge_maxage 12
        bridge_stp off
```

The configuration on the host is nearly completed. It's time to do some configuration on our UML guest:

```
host$ mount –o loop /home/giorgos/uml/root-fs /mnt/root-fs
host$ chroot /mnt/root-fs
host$ vi /etc/network/interfaces
host$ exit
host$ umount /mnt/root-fs
```

In '/etc/network interfaces' we put the following lines:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 192.168.2.2
        netmask 255.255.255.0
        network 192.169.2.0
        broadcast 192.168.2.255
        gateway 192.168.2.1

up route add -net 192.168.2.0/24 gw 192.168.2.1
up route add default gw 192.168.2.1
```

As shown in the above example, we configure the primary network interface of the UML guest 'eth0' with the IP address 192.168.2.2. Also, we specify the netmask, the network which is 192.168.2.0, the broadcasting address and finally the gateway 192.168.2.1 which

is the host end of the tunnel which is connected on the bridge. The final two lines need some more explanation.

As we said earlier the host is connected on the physical network 192.168.1.0/24 whereas the UML guest is connected on the network 192.168.2.0/24. In order for the UML guest to be able to access the machines on the network 192.168.1.0/24 and vice versa we must perform some routing. With the line 'up route add -net 192.168.1.0/24 gw 192.168.2.1' we add a new route to the network 192.168.0/24 specifying that all the packets that are directed for that network must be send on the host end of the tunnel which is 192.168.2.1. Then, the host is responsible to properly route this packets. The final line 'up route add default gw 192.168.2.1' adds a default route with gateway the host end of the tunnel which has the address 192.168.2.1. This helps the UML guest to communicate with other hosts on the Internet.

Finally, for the above configuration to work, we must make some additional routing configurations on the host. At this point, the UML guest can communicate properly with the host and vice versa. However, in order for the UML guest to communicate with machines on 192.168.1.0/24 network or with the internet we must enable routing on the host system. To do this we run the following command as root:

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

In order to make this change permanent we can set the option 'net.ipv4.ip_forward=1' in the '/etc/sysctl.conf' file. A final step is to enable IP masquerading with the help of IP tables. We run the following command as root:

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

After completing the above steps the network configuration needed to provide network connectivity to the UML guest is ready. If we run:

```
host$ route –n
```

on the host we will get:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 br0
```

Note the final line in the above example. This line specifies the route to the newly created network 192.168.2.0/24. All the packets will be routed to the bridge 'br0' in

order to reach the UML guests. Then, the bridge will route each packet to the correct TUN/TAP interface which is connected to it.

Let's run the UML guest:

```
host$ cd /home./giorgos/uml
host$ ./linux ubd0=root-fs mem=128M eth0=tuntap,tap0,,
```

After the guest boots we can run:

```
guest$ ifconfig
```

to see if the primary network interface is correctly configured. The result must look like the below lines:

```
eth0      Link encap:Ethernet  HWaddr ce:15:d1:2d:86:f0
          inet addr:192.168.2.2  Bcast:192.168.2.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:57 errors:0 dropped:32 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3121 (3.0 KiB)  TX bytes:159 (159.0 B)
          Interrupt:5

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Also, in order to see if the routing is properly configured we can run

```
guest$ route –n
```

The result must look like the below lines:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
Use Iface
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0        0
eth0
192.168.1.0     192.168.2.1     255.255.255.0   UG    0      0        0
eth0
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0
eth0
```

Try to ping the host and some internet addresses to see if everything works properly. Also, you can try to ping the UML guest from the host.

We have now configured a UML guest with network access. We can easily configure more UML guests to have network access by assign them IP addresses from the range of 192.168.2.0/24. Each UML guest must be connected on a different TUN/TAP device on the host. Thus, if we need to provide network access on two UML guests we have to create two TUN/TAP devices (e.g 'tap0' and 'tap1') on the host and connect them on the virtual bridge. Then we can run the first UML guest with 'eth0=tuntap,tap0' parameter and the second UML guest with 'eth0=tuntap,tap1' parameter.

7. References

1 Jeff Dike: User Mode Linux. Prentice Hall, 2006

2 Linux BRIDGE-STP-HOWTO [http://tldp.org/HOWTO/BRIDGE-STP-HOWTO/index.html]

3 UML – Setting up the network tutorial [http://user-mode-linux.sourceforge.net/old/networking.html]

4 Bridging Network Connections [http://wiki.debian.org/BridgeNetworkConnections]

5 TUN/TAP [http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt]

6 The Linux Kernel Archives [http://www.kernel.org/]