Getting the best from your server with User-Mode Linux

Matthew Bloch <matthew@bytemark.co.uk>
Bytemark Hosting

9th February 2004

Abstract

A straightforward guide to User-Mode Linux, a maturing virtualisation technology, and a set of recommendations on "best practice" Linux system administration for managing large numbers of virtualised systems. This paper is based on the technical experiences arising from running a hosting business based on User-Mode Linux technology.

1 Virtualisation

1.1 Overview

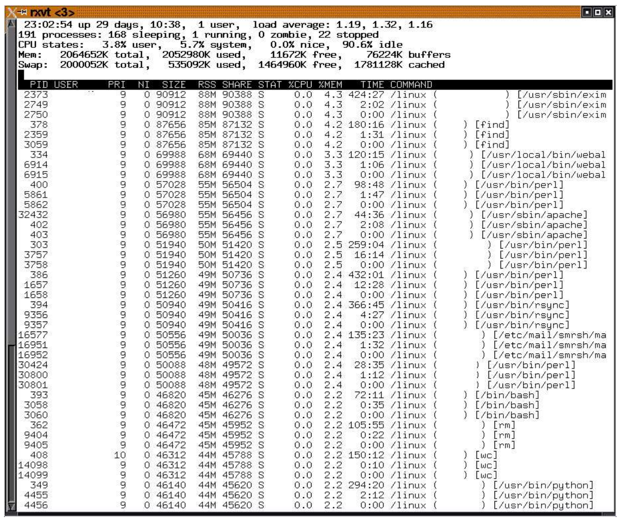
Virtualisation is the term for encapsulating the facilities of a complete running computer system inside another piece of software. We tend to talk of *host* and *guest* systems where the host is the more powerful system performing the virtualisation, and the guest is the system which is fooled into running in an environment which it was not explicitly designed for. They could even be identical computer systems, but one plays host to the other and has full control over its operating environment.

Just as a pilot inside a flight simulator has everything he needs to exercise his flying skills to the full without fearing the consequences, a guest system inside a virtualiser can execute any program with the same impunity; the virtualiser provides a near enough match to the expected environment that the guest does need to make any adjustments to how it would normally operate. The guest system can be used to run risky or untested code without fearing the consequences, since it can be quickly reset through the host system.

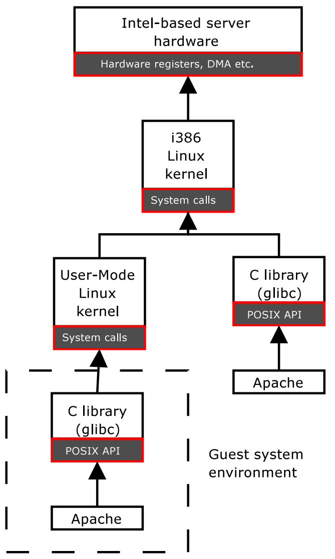
1.2 A different port of Linux

User-Mode Linux[1] is a virtualiser, but also a bona-fide new port of the Linux kernel. The port is not to a new hardware platform, but to a software platform: the ported kernel runs as a process inside a normal i386 Linux kernel with user privileges. So instead of drivers accessing real hardware, its drivers access files on the host instead. Internally it is a full Linux kernel environment which is capable of running unmodified Linux software. Externally it appears as four processes running on a host Linux system. Here is the output from "top" on a host machine running 25 different guest kernels, and a diagram of the layers of software compatibility that make it work:

[1] http://user-mode-linux.sf.net/


```
rxvt <3>
 23:02:54 up 29 days, 10:38,  1 user,  load average: 1.19, 1.32, 1.16
191 processes: 168 sleeping, 1 running, 0 zombie, 22 stopped
CPU states:   3.8% user,   5.7% system,   0.0% nice,  90.6% idle
Mem:   2064652K total,  2052980K used,    11672K free,    76224K buffers
Swap:  2000052K total,   535092K used,  1464960K free,  1781128K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 2373             9   0 90912  88M 90388 S     0.0  4.3 424:27 /linux (          ) [/usr/sbin/exim
 2749             9   0 90912  88M 90388 S     0.0  4.3   2:02 /linux (          ) [/usr/sbin/exim
 2750             9   0 90912  88M 90388 S     0.0  4.3   0:00 /linux (          ) [/usr/sbin/exim
  378             9   0 87656  85M 87132 S     0.0  4.2 180:16 /linux (    ) [find]
 2359             9   0 87656  85M 87132 S     0.0  4.2   1:31 /linux (    ) [find]
 3059             9   0 87656  85M 87132 S     0.0  4.2   0:00 /linux (    ) [find]
  334             9   0 69988  68M 69440 S     0.0  3.3 120:15 /linux (     ) [/usr/local/bin/webal
 6914             9   0 69988  68M 69440 S     0.0  3.3   1:06 /linux (     ) [/usr/local/bin/webal
 6915             9   0 69988  68M 69440 S     0.0  3.3   0:00 /linux (     ) [/usr/local/bin/webal
  400             9   0 57028  55M 56504 S     0.0  2.7  98:48 /linux (    ) [/usr/bin/perl]
 5861             9   0 57028  55M 56504 S     0.0  2.7   1:47 /linux (    ) [/usr/bin/perl]
 5862             9   0 57028  55M 56504 S     0.0  2.7   0:00 /linux (    ) [/usr/bin/perl]
32432             9   0 56980  55M 56456 S     0.0  2.7  44:36 /linux (     ) [/usr/sbin/apache]
  402             9   0 56980  55M 56456 S     0.0  2.7   2:08 /linux (     ) [/usr/sbin/apache]
  403             9   0 56980  55M 56456 S     0.0  2.7   0:00 /linux (     ) [/usr/sbin/apache]
  303             9   0 51940  50M 51420 S     0.0  2.5 259:04 /linux (        ) [/usr/bin/perl]
 3757             9   0 51940  50M 51420 S     0.0  2.5  16:14 /linux (        ) [/usr/bin/perl]
 3758             9   0 51940  50M 51420 S     0.0  2.5   0:00 /linux (        ) [/usr/bin/perl]
  386             9   0 51260  49M 50736 S     0.0  2.4 432:01 /linux (    ) [/usr/bin/perl]
 1657             9   0 51260  49M 50736 S     0.0  2.4  12:28 /linux (    ) [/usr/bin/perl]
 1658             9   0 51260  49M 50736 S     0.0  2.4   0:00 /linux (    ) [/usr/bin/perl]
  394             9   0 50940  49M 50416 S     0.0  2.4 366:45 /linux (    ) [/usr/bin/rsync]
 9356             9   0 50940  49M 50416 S     0.0  2.4   4:27 /linux (    ) [/usr/bin/rsync]
 9357             9   0 50940  49M 50416 S     0.0  2.4   0:00 /linux (    ) [/usr/bin/rsync]
16577             9   0 50556  49M 50036 S     0.0  2.4 135:23 /linux (       ) [/etc/mail/smrsh/ma
16951             9   0 50556  49M 50036 S     0.0  2.4   1:32 /linux (       ) [/etc/mail/smrsh/ma
16952             9   0 50556  49M 50036 S     0.0  2.4   0:00 /linux (       ) [/etc/mail/smrsh/ma
30424             9   0 50088  48M 49572 S     0.0  2.4  28:35 /linux (      ) [/usr/bin/perl]
30800             9   0 50088  48M 49572 S     0.0  2.4   1:12 /linux (      ) [/usr/bin/perl]
30801             9   0 50088  48M 49572 S     0.0  2.4   0:00 /linux (      ) [/usr/bin/perl]
  393             9   0 46820  45M 46276 S     0.0  2.2  72:11 /linux (    ) [/bin/bash]
 3058             9   0 46820  45M 46276 S     0.0  2.2   0:35 /linux (    ) [/bin/bash]
 3060             9   0 46820  45M 46276 S     0.0  2.2   0:00 /linux (    ) [/bin/bash]
  362             9   0 46472  45M 45952 S     0.0  2.2 105:55 /linux (       ) [rm]
 9404             9   0 46472  45M 45952 S     0.0  2.2   0:22 /linux (       ) [rm]
 9405             9   0 46472  45M 45952 S     0.0  2.2   0:00 /linux (       ) [rm]
  408            10   0 46312  44M 45788 S     0.0  2.2 150:12 /linux (    ) [wc]
14098             9   0 46312  44M 45788 S     0.0  2.2   0:10 /linux (    ) [wc]
14099             9   0 46312  44M 45788 S     0.0  2.2   0:00 /linux (    ) [wc]
  349             9   0 46140  44M 45620 S     0.0  2.2 294:20 /linux (       ) [/usr/bin/python]
 4455             9   0 46140  44M 45620 S     0.0  2.2   2:12 /linux (       ) [/usr/bin/python]
 4456             9   0 46140  44M 45620 S     0.0  2.2   0:00 /linux (       ) [/usr/bin/python]
```



This type of "porting" is not a new idea, and Linux has been ported as a user-space application in the past (to the IBM OS/390 mainframe architecture) but UML achieves this impressive goal on commodity hardware through a relatively small set of modifications to the core Linux code.

As a brief aside, a dizzying number of other user-mode porting projects are in full swing: Xen[2] is an open source *Virtual Machine Monitor* developed at the University of Cambridge which runs another port of Linux; not a user-mode port, but it does not run at the full privilege level usually reserved for the OS on an Intel computer. coLinux[3] is another port of Linux to Windows' user-mode environment and system call interface. Faumachine[4] started as a fork of User-Mode Linux with an emphasis towards testing and fault diagnosis, and plex86[5] is a project to implement a lightweight virtualiser as a kernel module.

Despite all the choice, Jeff Dike's User-Mode Linux project is the simplest to deploy Linux virtualisation technology, and one with the most overtly practical emphasis.

1.3 Economic incentives

Since its first running version, UML has been a useful development tool for kernel developers wanting to test new kernel code without constant reboots. However since November 2002 its performance was increased by an order of magnitude; enough to make it viable as an efficient virtualiser for ordinary server tasks. This table gives a rough summary of the number of 64MB Linux servers that can be run efficiently on a modern P4-based server with redundant hard drives:

	1U server	**2U server**
CPU	2.6GHz Pentium 4	2 x 2.4GHz Xeon
Memory	2048MB	4096MB
Guest kernels	25	50
Price (approx.)	£1000	£2000
Cost per 64MB server	£40	£40

It was this grimy economic incentive which drew Bytemark (and many other companies) to commercial hosting with UML. Space in a well-connected data centre

[2]http://www.cl.cam.ac.uk/Research/SRG/netos/xen/
[3]http://www.colinux.org/
[4]http://www.faumachine.org/
[5]http://www.plex86.org/

is very expensive, but that most general purpose Linux mail & web hosts are idle most of the time, hence the space is often sitting idle. UML allows such "high-end" data centre space to be sold in much smaller increments than a rack unit while providing all the same facilities.

2 Setting up a User-Mode Linux host system

Virtualisers in general make difficult demands on a host machine, especially if many are running at the same time. In a multi-user environment, the administrator of such a host system will be under pressure to ensure the virtualised systems behave externally *exactly* as a non-virtualised system. On one hand he can optimise their performance for common system loads at the risk of creating a worst case scenario where all virtualised systems suddenly make resource demands at once and slow down. On the other he can try to make the systems behave fairly, trying to make sure each guest can make a certain level of demand at any time; this is usually at the expense of overall performance, but is the option we prefer.

The former configuration means configuring the virtualised systems to share the host resources as much as possible to trade off each others' ebbs in demand, whereas the latter means configuring them to act as independently and selfishly as possible to minimise the variance of demand on the host.

Contrary to most people's initial assumptions, the main point of contention in a User-Mode Linux system is not the CPU: when User-Mode Linux processes are vying for its time, they will receive it pretty much equally. The real problem is contention for disc access; the less RAM a guest system has, the more it must rely on its swap, and it's the disc access which suffers under load and which is not always fairly distributed between guests.

Here we will discuss the basics of setting up a User-Mode Linux host system, and the tools needed to divide up the host between guest operating systems.

2.1 Building and running User-Mode Linux

The latest patches to the 2.4 and 2.6 vanilla Linux kernels are found at the User-Mode Linux web site[6]. The bulk of its code can be found the arch/um directory alongside i386 and other system-specific "port" code. After applying the patch, the user can configure and build the kernel as a user process in exactly the same way as a normal cross-architecture compilation:

```
make ARCH=um menuconfig
make ARCH=um linux
```

This will compile Linux as a normal ELF binary which can be run by any user. A pre-built kernel can be found compiled for Debian systems (*apt-get install user-mode-linux*) systems, and an RPM for Redhat systems is downloadable from the User-Mode Linux web site. However there are currently various security and stability compromises that the system administrator needs to make which ensure there is no easy agreement on what a sensible "standard" user-mode kernel should be. Thus administrators need to be as familiar with building UML kernels as they are with building regular kernels. Here are some build-time decisions the administrator needs to make:

- Does the kernel need to run on a host with more 2GB physical memory? This requires a slightly different memory mapping inside the guest kernel, and is an explicit build-time option.

[6]http://user-mode-linux.sf.net/

- Does the guest system administrator need access to the host filesystem?

In addition there are some more developer-oriented options, including an option to simulate an SMP system even on a host with a single processor, support sound output through the host and even allow direct hardware access for driver writers. Also many non hardware-specific patches will be applicable to a UML kernel: so we compile in FreeS/WAN for IPSec support among other patches.

Although the Linux binary does not require superuser privileges to run, it will require superuser intervention if the user running the guest kernel wishes to use networking or allocate large amounts of memory to their kernel. The recommended way of managing such privileges is through a *run script* which is set up by the host administrator to perform any super-user actions in a systematic and secure fashion. Although users may run UML kernels without any such superuser setup, it makes networking to the outside world all but impossible.

2.2 Setting up the environment

The system administrator divides up the host system among the guests using several tools and command-line options.

2.2.1 RAM

Guest kernels will need a certain amount of RAM to function properly. Currently this is specified simply by telling the guest how much to claim:

```
./linux mem=128MB ...
```

Currently it is possible to ask the kernel to claim far more physical RAM than is available to the host, since not all of the RAM is claimed immediately. This is an example of how to create a "worst case" scenario for the benefit of some gain under normal conditions: the guest kernels can in total claim more RAM than is available on the host, but it will only be allocated when it is needed. In practice this really doesn't work for very long.

Even if you don't overallocate memory, it is necessary to appreciate how the host kernel allocates RAM to its processes. Unless some action is taken to ensure to the contrary, the guest kernel may end up accessing physical RAM which has been swapped to disc. This situation is always a performance disaster for the guest system, and is discussed in 5.3 onwards.

2.2.2 Disc files for block devices

Every Linux system needing to organise its own filesystem on local storage needs a block device on which to write it. Just as the host system will have its hard discs or RAID array, the guest system will use large files created on the host's filesystem to act as block devices.

Disc files are created simply by creating a large file, e.g. to create a 3GB disc on the host:

```
dd if=/dev/zero of=/path/to/ubd2 bs=1024 count=$((3*1024*1024))
mkfs.ext3 /path/to/ubd2
```

and to map it to a block device in a User-Mode Linux process:

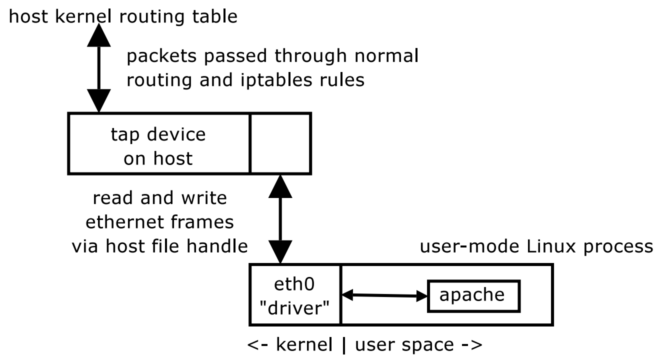
```
./linux ubd2=/path/to/ubd2 ...
```

User-Mode Linux's block devices are called /dev/ubd0 to 7 and have their own distinct device numbers. You can configure each of these to map to a separate file on the host which can be used in the same way as a block device on a normal Linux system. So you can use eahc one as a swap partition, backing for a filesystem or component in a RAID or LVM array. The block device driver does not simulate partitions on each block device, since there is no practical need to. User-Mode Linux can also support booting and running with an NFS root filesystem with the usual kernel options if this is desirable.

2.2.3 TUN interfaces to manage networking

The eth0 device on a guest kernel will be mapped to a TUN network device on the host. There are several other "driver" options for eth0 (e.g. one of the utilities accompanying UML is a virtual switch daemon which can be used instead of a TUN device) but we will concentrate on manually configured TUN devices since they are the most flexible.

TUN devices act like miniature ethernet segments on the host, but with a process responsible for reading and writing ethernet frames. The process can thus force packets through the kernel's routing table, and have packets delivered back to it in a secure fashion. The host system administrator can control what packets the guest kernels see through normal routing and firewalling techniques.


host kernel routing table
packets passed through normal routing and iptables rules
tap device on host
read and write ethernet frames via host file handle
user-mode Linux process
eth0 "driver"
apache
<- kernel | user space ->

Creating a TUN device on the host machine can be accomplished with the tunctl tool found with the User-Mode Linux tools package:

```
tunctl -u matthew -t tap_matthew
ifconfig tap_matthew hw ether fe:00:00:00:00:01
```

This creates an interface which can only be used by UNIX user "matthew" with a fictional, but fixed, hardware address for the host system. The hardware address should be set consistently for each guest system (e.g. you could assign them automatically by Linux user ID). To then use this TUN interface as an ethernet interface on in the guest system, we can pass these arguments to a User-Mode Linux process.

```
./linux eth0=tuntap,tap_matthew,fe:00:00:00:00:02 ...
```

Again you need a unique hardware address for the guest system, different to that which you assigned to the host system. The guest system administrator can then communicate directly with the host and forward packets through it.

We would recommend always fixing the hardware address at each end of the TUN interface, otherwise it is set randomly. It does makes sense to uniquely fix the MAC address for each guest system in a cluster, otherwise IPv6 autoconfiguration will operates counter-intuitively, allocating the guest kernel a random IP address each time the system boots.

2.3 Terminal devices

The guest kernel can be assigned host terminal devices to which it should map its console (/dev/console), terminal (/dev/tty*) and serial (/dev/ttyS*) devices. The simplest to set up is to allocate the guest kernel a physical terminal on the host which you can change to with the usual Ctrl+Alt+Fkey. For instance to set up the console on physical terminal 9:

```
./linux con0=tty:/dev/tty9
```

More generally useful is to allocate a pseudo-tty:

```
./linux con0=pty
```

This technique allows the user to attach a terminal program to the pty once it is allocated, e.g. GNU screen. The administrator or other users of the guest kernel can then log into the kernel as if at a keyboard or serial terminal, which is useful for emergency access when the guest kernel's network setup or secure shell daemon is broken.

2.4 Preparing a guest filesystem

Since filesystem building is something that is done relatively infrequently, and since the UML's "hardware" configuration never changes significantly from the inside, there is relatively little motivation to try to do a completely thorough job of it from scratch each time unless a highly programmatic approach to generating a base system is needed. If systematisation is needed, a tool such as System Imager[7] may do a better job of dynamically reconfiguring a known-working filesystem image than trying to generate one from the distribution media each time.

Arguably the simplest and most practical way is simply to lift a clean installation of the relevant Linux distribution from a normal i386 install and copy all the files into a loopback filesystem. You can then use this giant disc file as a template which is manually or automatically edited for each new user. A minimal set of changes needed to get a Debian system up and running from such a state would be:

- **/etc/fstab** : change /dev/hd[abcd] reference to /dev/ubd[0123] ;
- **/etc/network/interfaces** : alter to suit the configuration of the TAP interface;
- **/etc/ssh/ssh_host_key*** : regenerate with ssh-keygen otherwise every new host will have the same host keys;
- **/etc/hosts** : rewrite to reflect external IP address of host;
- **/var/logs/*, /root/.bash_history etc.** : remove completely;
- **/etc/shadow** : assign a random or requested root password for the system's administrator.

The more software you leave in your base image, the harder it will be to configure consistently, so keeping it simple is obviously a virtue. However the more software that is pre-configured and known that the user will not need to reinstall or reconfigure significantly, the more efficiently Copy-on-Write files will work (see 5.1).

A good final step before handing over a guest filesystem to a user is to update its packages to the latest patched versions; under Debian *apt-get update; apt-get dist-upgrade* will make sure you are not giving the guest administrator any long-fixed security holes, and will ensure the network configuration is correct as well.

[7] http://www.systemimager.org/

2.5 Booting and kernel updates

Probably the major difference between working inside a User-Mode Linux system and a conventional i386 Linux system is that the kernel resides *outside* of the filesystem since the host system itself is the boot loader. Currently there is no boot loader which will pick a kernel from inside a disc file and run it directly in the same way that LILO or grub do; hence users will be unable to recompile their own kernels easily without co-operation from the administrator. Such co-operation is usually desirable in all but the most trusted of environments for security reasons. Regular user-driven kernel updates in a UML environment is not generally appropriate because:

- The administrator must take care of both host and guest security updates in a timely fashion: compromises of guest systems could be escalated into compromises of host systems, which could compromise *all* guest systems on a host.
- There is no variety of hardware for a user-mode kernel to take into account, hence guest system administrators do not need to compile in differing hardware drivers. It is the host administrator's decision as to what kernel build will work best on the hardware.

Since Bytemark opened its doors, only three of our customers have had a genuine need for a separate kernel– for other users we have rolled in all the patches people requested into a standard kernel which everyone is happy with.

2.6 Remote control over a running kernel

The simplest form of control the host administrator has over a UML process is to kill it. This is equivalent to pulling the plug on a conventional Linux server with much the same consequences, and is only a sensible option when potential data loss is not relevant, or when the prospect of a clean reboot seems remote.

A helpful feature of the user-mode kernel is its "management console" facility. When the kernel starts up it creates a local socket which can be used to pass commands to the kernel. The administrator can use this facility through the supplied *mconsole* tool to do any of the following:

- disaster recovery: send a Ctrl+Alt+Del or system request to the kernel to cause kneejerk responses (unmount, reboot, synchronise discs etc.);
- dynamic resource allocation: add or remove network interfaces and block devices;
- monitoring and remote troubleshooting : read files from the filesystem including those in /proc;
- read what host resources are currently allocated (espeically useful to find out which pty to open with *screen* on the host when you've configured a serial line to use a pseudo-terminal device as in 2.3).

So far this is the state of the art in setup and remote control of User-Mode Linux processes. It naturally follows to work out how to put all these facilities into a set of sensible administration practices.

3 UML administration concepts

A different set of skills are needed for the efficient management of a host running User-Mode Linux systems compared to a single server running a typical mix of "office" type services.

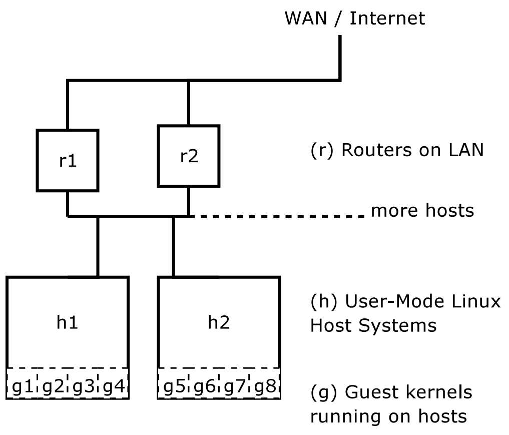
Typical UNIX practice is to draw a line between the capabilities and privileges of the sysadmin and his users. In a situation where a UML host is running tens of independent Linux systems administered by different people, you are in a situation where your users are also systems administrators, and those administrators may have responsibility for users themselves. As the administrator of the host system, it is much easier to break, corrupt or cut off your users' systems than it would be if these were physical hosts.

While it is simple to start your first UML system running, I propose certain best practices for effective large-scale management of many guest systems. These have stemmed from the practices we developed at Bytemark to manage our customers systems with minimal disruption while being able to add many advantages over traditional co-location packages.

I'll describe some design principles drawn from experience (and from the presentable half of our own management scripts) which should help the administrator implement his own set of scripts to easily manage a large-scale UML installation.

3.1 Properties of User-Mode Linux clusters

The following setup is an appropriately simple but scalable architecture for deploying clusters of User-Mode Linux servers: we need one or more border routers for each cluster, and a host machine for roughly every 25 guest systems. It is certainly possible to live without your own router in this situation, but movement of guest systems between hosts becomes much more difficult without the routers' co-operation.



An idea that has helped us think more clearly about planning deployments of UMLs is that *host machines are better thought of as routers.* They contain inbuilt networks of independent machines, but in a lot of ways they can be treated as living on an invisible internal network. So half the problems that you face as an administrator are those you'd face with any network of routers.

The other half of your problems will stem from certain expectations of your administration that need to be upheld from the principle that *your host machines must be as invisible as possible:*

1. guest systems must be mobile between hosts in such a cluster without needing their filesystems modified;
2. guest systems must be rebooted immediately if they crash;
3. the administrator must be able to shut down a guest system cleanly without warning and without risking damage;
4. the administrator must be able to add and remove resources from a guest system without demanding it is restarted;
5. guest systems must not be able to interfere with each other, or the host.

In short, guest system administrators should not need to know the details of the system that they are hosted on, and should be able to treat their guest system as if it were a normal physical machine. The host administrator needs to maintain this illusion as much as possible.

3.2 Creating a structure for guest systems

Host machines need to run very little software: we use a standard Debian installation running sshd and Dan Bernstein's process supervisor suite in place of init scripts.

In order to isolate guest systems' execution from each other, we assign each one its own Linux account and arrange them in a distinct directory structure.

File	Owner	Description
/machines/...	root	All machines on this host
/machines/*user*/...	root	Directory for user
/machines/*user*/run	root	Run script for guest kernel (see below)
/machines/*user*/jail/...	*user*	"jail" directory for guest kernel
/machines/*user*/jail/ubd2	*user*	Block device file
/machines/*user*/jail/linux	*user*	Linux kernel image
/machines/*user*/settings/...	root	Settings directory

On our systems, these user accounts are login accounts but with a specially restricted shell to allow users to perform "out of band" operations on their guest system such as forcing them up or down, booting them at the console and so on. Since a system which is primarily running UML kernels will find its time largely taken up with that task, there does not seem a practical use for giving users full shell logins to the host.

We use Dan Bernstein's daemontools[8] to denote an *activated* from a *deactivated* guest system. Briefly, the daemontools suite of software is a simple replacement for the usual init script arrangement. It manages a set of single-process services in the /service directory. To start a service, you create a run script which you place in /service/*yourservice*/run and this will be kept running, being restarted if necessary, until you remove the directory from /service. Along with its other features, this takes some tedium out of maintaining init scripts for custom services such as keeping guest kernels running.

Thus activated systems are marked out with a symlink from /machines/user to /service/uml-user. When the symlink is not there, the guest system is considered deactivated and will not restart automatically.

The settings directories contain the kernel's boot-time parameters, and they usually imply some setup on the host: so we have assigned IPv4 addresses, IPv6 prefixes, memory size, swap size and any other host resources which need some setup before the Linux binary itself is run.

[8]http://cr.yp.to/daemontools.html

3.3 Copying guest systems between hosts

Since each guest system's state (kernel, filesystem and allocated host resources) is completely contained in its own directory structure, the relevant user directory can be copied wholesale to another host in a cluster and activated immediately.

This implies some cleanup work when deactivating on the old host machine. For one, the old host must release any claims on that guest's IP address allocations, usually by explicit communication with the router. It must also ensure that the kernel is not running before allowing a copy to take place.

3.4 The run script

This is a crucial part of a functioning User-Mode Linux host machine. The run script is executed as root, and what it needs to do is of course to run the guest kernel, making up suitable command-line parameters from the available settings.

Most of its functionality should be concerned with checking that running the kernel is safe, i.e.

1. that the guest system's directory structure is intact: disc files, kernels and so on should be present to ensure the kernel does not abort unexpectedly when booting,and swap files created to the correct size if not present;

2. that the kernel is not already running on this host machine (we use a lock file to avoid filesystem corruption);

3. that the IP addresses it is about to claim do not respond to pings (which would imply the same guest system is already running on another host);

4. that the host kernel has the necessary support for SKAS mode (see 5.2).

The run script should stop in its tracks if it suspects any of these problems, but otherwise needs to set set up the network interface and routing for the guest system's configured IP addresses, both IPv4 and IPv6 before dropping to user privileges and running the linux binary itself.

On Bytemark's sytems we also perform a chroot to the jail directory as a security measure just before dropping to user privilege, so that the UML process only has access to its own disc files. The jail directory also needs the following files set up:

- special file /dev/net/tun (character device 10, minor 200) should be created with mknod; this device is used by the guest kernel to read and write network packets through its TUN device;

- special file /dev/ptmx (charcter device 5, minor 2) should be created for the guest kernel to open pseudo-terminals on the host;

- the devpts filesystem should be mouned on /dev/pts for the same reason;

- /proc/mm needs to be available if the host is running the SKAS3 patch (see 5.2 for details); the `mount --bind` command can be used to remap this from the host's /proc filesystem without giving the guest kernel access to the whole host /proc filesystem.

We do not pretend that this jail setup is as secure as it can possibly be, but it provides some common sense protection against malicious parties who manage to break out of a guest kernel.

Finally the run script has three very helpful stop-gap measures to help manage troublesome host systems:

- it will not start any guest systems nor allow any user logins when the file /machines-nostart has been touched by the host administrator;
- it will not start any guest systems until the host's uptime is more than 120 seconds;
- before starting any guest system when the above conditions have been previously true, it will wait a random amount between 1 and 600 seconds before starting the kernel.

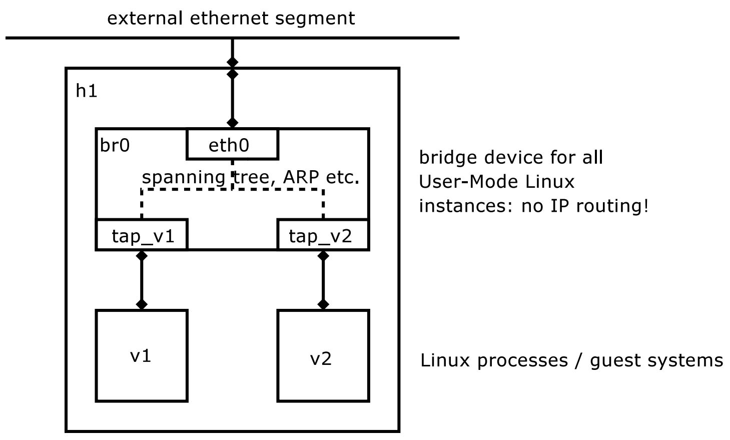
Thus if a host machine has just been rebooted, it will not come under any heavy load until the administrator has had a chance to log in and prevent it if necessary. It will also make sure that when the guest systems come up, they come up in a staggered fashion so that each guest's boot process runs relatively promptly rather than being in competition with every other kernel booting at the same time.

4 Arranging the network

A TUN interface is simply a virtual "crossover cable" between a host and a process on that host, and are the basis of networking with User-Mode Linux. Understanding the possibilities for connecting TUN devices to the outside world is crucial to deploying UML; here we will describe the most common setups with reference to a the following example network setup:

Host system IP address	1.2.3.10
Guest system IP address	1.2.3.20
Netmask for external network	255.255.255.0
Router on external network	1.2.3.1
Guest system's TUN interface name	tun_guest
Host system's external interface	eth0

4.1 Bridge-based networking


external ethernet segment
h1
br0
eth0
spanning tree, ARP etc.
tap_v1
tap_v2
v1
v2
bridge device for all User-Mode Linux instances: no IP routing!
Linux processes / guest systems

This routing is the simplest to configure: the kernel bridge device allows several network devices to be clumped together as if they were joined by a single ethernet switch with the `brctl` tool. Every device in the bridge has equal privilege, and thus UML instances can interact with the external ethernet segment exactly as the host machine can. This means they can claim IP addresses simply by replying to ARP requests which they will see in exactly the same fashion as a host connected to the external ethernet.

This also means that UML instances can use ARP replies or other ethernet-level spoofing techniques to claim any IP address they like, including those which might be used by the host or other critical devices on the ethernet segment. In environments where guest kernels are not trusted, this configuration must be secured through use of the ebtables patch to hardwire each guest system a "physical" MAC address and limit which IPs can come from those MAC addresses, or additionally through MAC filtering on the switch.

To secure a bridged setup is probably a more difficult job than the alternatives in an untrusted environment, but for a trusted environment, it is the quickest and simplest solution. Also it is perfectly secure if the guest kernels only need to talk to each other rather than to the outside world; simply leave the external interface out of the bridge configuration.

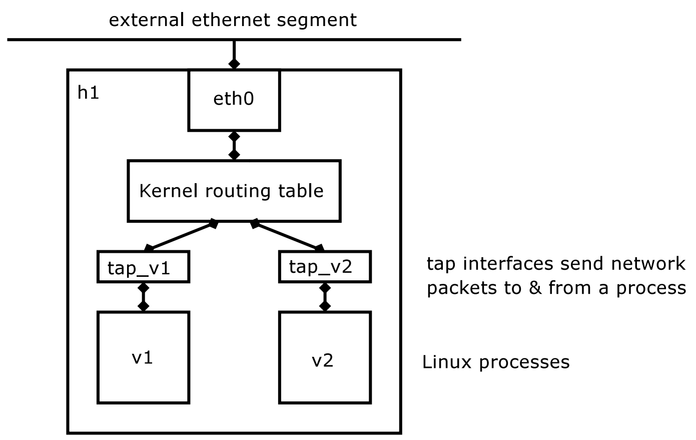
Example host-side configuration

```
brctl addbr bigbridge
brctl addif bigbridge eth0
brctl addif bigbridge tun_guest
```

Example guest-side configuration

```
ifconfig eth0 1.2.3.20 netmask 255.255.255.0 gw 1.2.3.1
up
```

4.2 IP-based routing ("level 3")


external ethernet segment
h1
eth0
Kernel routing table
tap_v1
tap_v2
tap interfaces send network packets to & from a process
v1
v2
Linux processes

In this configuration, guests are limited to communication with the host machine and not automatically allowed access to ethernet frames on the external network segment. This means that all routes need to be set up explicitly in this fashion by the run script:

ip route add 1.2.3.4 dev tap_guest1

This ensures that UMLs will see only packets destined for their assigned IP addresses. However they will not see any traffic for their IP address unless traffic for that IP ends up in the host machine's routing table. Here are three possibly strategies to make the routing work:

1. Enable proxy arp on the host kernel, and manually add the interface mapping for the allocated IPs into the host's arp cache. This instructs the host

to answer ARP requests on behalf of all its guest systems without the need to permit the dangerous privilege of a direct ARP reply. This is simple to setup but has a complication if a guest machine is moved between hosts in a cluster: the host system which used to host the moved guest must be instructed to forget about its old ARP entry to avoid claiming it in perpetuity, and the new host system will need to issue a gratuitous ARP to effect the move *in that order*. Not getting this right results in the guest system becoming wholly or sporadically unavailable after the move.

2. Run a suitably configured routing daemon on the host machines (h0-h1) and internal routers (r0-r1) to propogate the routing changes automatically through RIP or OSPF. Unfortunately Linux is not well catered for in this area. The most mature routing suite we found was Quagga[9] but it's not simple to set up.

3. Statically route IP blocks to a particular host machine from the internal routers. This is the simplest and most secure solution, but removes the ability for a particular guest system to run on any of the hosts in the event of a host failure which breaks our first principle that the guest system should not need to know any details of the host it is running on.

Though this is a relatively secure configuration, the network is still vulnerable to guest machines spoofing their source IP address (though they will never see incoming traffic to any spoofed address). A simple iptables rule can be used to stop spoofed source IP addresses coming from "the wrong TAP interface", and insertion of such a rule should be part of the run script (see 3.4).

Example host-side configuration (assuming option 1)

```
echo 1 >/proc/sys/net/config/all/proxy_arp
ip route replace to 1.2.3.20 dev tap_guest
arp -s 1.2.3.20 -i eth0 -D tap_guest pub
```

Example guest-side configuration

```
ifconfig eth0 1.2.3.20 up
route add default dev eth0
```

4.3 IPv6 considerations

IPv6 address space presents a more complicated routing setup than with IPv4. Guest system administrators will expect the kernel's IPv6 autoconfiguration to "just work", i.e. their system should see router advertisements appear on their system's ethernet interface, and be able to assign their own IPv6 address uniquely from within the advertised range.

4.3.1 Bridging

Preserving expectations of IPv6's autoconfiguration mechanism is a very good reason for fixing guest and host systems' hardware addresses on a TUN network interface (as discussed in 2.2.3). If the host is using the simple bridged network configuration, autoconfiguration work as expected: guest systems will see the

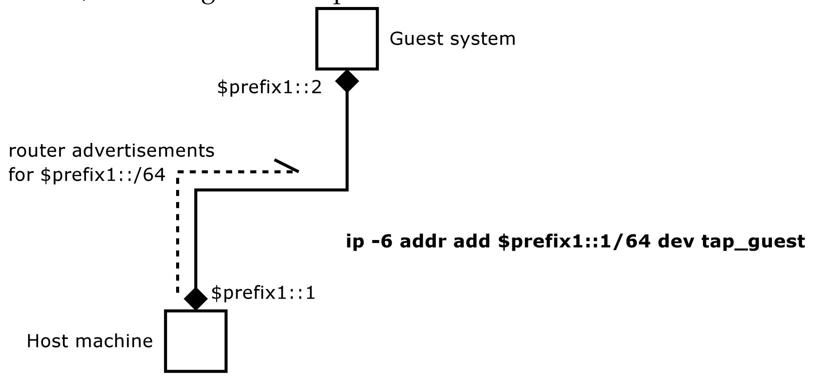
[9]http://www.quagga.net/

router advertisements coming from upstream and configure themselves accordingly. However they will be sharing the same IPv6 address space as other systems on the ethernet segment, and allocate themselves only a single address.

This carries with it the same security issues as for IPv4 regarding address spoofing; however bridging requires no extra setup effort and may be completely acceptable.

4.3.2 Full routing of IPv6 prefixes

Because IPv6 address space is not in short supply, guest administrators may expect a /64 or larger to be routed to their system for its exlusive use. For a single /64 allocation, we arrange this setup like this on the host:



In order to advertise routes to the guest systems, the host runs radvd[10]. Its configuration is regenerated periodically according to the current layout of its /machines directory (see 3.2). This causes it to advertise unique, global address prefixes to whichever virtual machines have them allocated by the host administrator shortly after he has altered a machine's IPv6 settings file.

We instruct the kernel to route the /64 allocation direct to the TAP interface, which means that the host can claim any address in its range that it likes. However we reserve ::1 which for the host machine itself. Also if the guest system administrator requests a second /64 or larger block to be allocated for tunnelling elsewhere, we instruct the host system to route further IPv6 subnets through ::2 (assumed to be claimed by the guest), allowing the guest system to act as a full router for those subnets. This behaviour is documented to guest system administrators so they can configure their systems to route accordingly without letting down those who expect autoconfiguration to work intuitively.

5 Performance considerations and future developments

User-Mode Linux is developing quickly and there are features which can be used to gain performance from guest kernels, although they can make life complicated for administrators which I've left until last for clarity.

For high-density hosting, I would only consider the SKAS patch and use of tmpfs to be compulsory, but it is worth knowing about the other options.

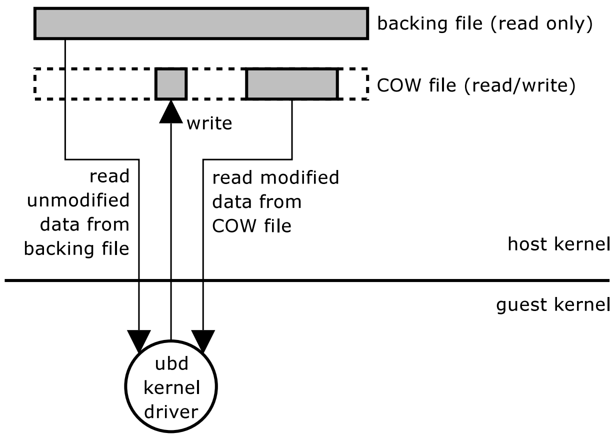
5.1 Copy-on-write (COW) disc files

On host systems running many guest kernels, you will probably develop a standard distribution or two which can be given out and customised for each user. For instance if you are running many Debian guest systems, you may make copies

[10]http://v6web.litech.org/radvd/

of the same root disc image for ten different users and alter only the networking configuration for each one.

However a large proportion of the each of these filesystem images will stay the same across the different installations, even after multiple upgrades to the system. This results in the host machine cacheing the same data many times over in its disc cache. An optimisation to get around this problem is to use a "copy on write" file. This is a file which contains only the changes to a particular disc image file. Where a particluar disc block has not been changed, it is read from the *backing file*; where it has, it is read from the *COW file:*



There are two advantages of COW files over making full copies of backing files:

- Chiefly, the host's disc cache will store the unmodified portions of a backing file only once, rather than several times for each guest system;
- COW files are created as sparse files on the host filesystem, so they only take up as much space as they contain modified data.

There are two disadvantages, which are areas of active development:

- COW files are a simple format which are currently specific to User-Mode Linux, so cannot be mounted over the host's loopback device (though there is a tool *uml-moo* to merge a COW and backing file);
- there is a long standing bug in the COW driver which can causes crashes when writing near the end of the disc.

Despite these inconveniences, the win on disc cacheing is large enough to make a noticeable difference on guests systems running a largely stable distribution.

5.2 The SKAS kernel patch

Since November 2002, UML will run in one of two modes depending on whether the this kernel patch has been applied.

The original is called *tracing thread* mode because it uses standard debugging features of the Linux kernel to allow the UML kernel to run effectively in user mode. It uses signals between processes to communicate system calls (very slow!) and forks a new host kernel process for every user-mode kernel process. It also maps the kernel into each guest process' address space; this is a security hole because of course any process running under a user-mode kernel can corrupt or rewrite the kernel by just writing to the top end of their address space.

The SKAS patch can be applied to the host kernel to provide a new device. In the widely-deployed skas3 patch, this appears as /proc/mm, though this is set to change in future releases. This device provides an extra hook into the host kernel which allows guest kernels more flexibility in managing pools of address space, and thus removes many practical limitations of tracing thread mode, most of all the speed and security issues. Processes running under the user-mode kernel in *SKAS mode* no longer have access to the kernel address space, and can communicate with the kernel much more quickly. This is much more efficient, to the point where a guest kernel can perform some tasks at a similar speed to the host itself. The host's scheduler is also not lumbered with one host process for every guest kernel process (each UML in SKAS mode appears as four processes on the host), which means that it can allocate CPU time more fairly.

5.3 Using tmpfs

User-mode kernels allocate memory by creating a memory-sized file in /tmp, then reading and writing from it. For example if you specify mem=64M on the linux command line, the kernel will create a 64MB file in /tmp.

The standard Linux kernel's disc cache (from the host's point of view) will be used up very quickly with memory accesses from multiple guest kernels. This will cause most memory to be read and written from disc, causing unnecessary extra contention on the disc cache.

It is a simple matter to mount the host's /tmp filesystem with the tmpfs filesystem. The tmpfs filesystem allocates space for files from the system's page cache rather than the disc cache. This makes them, of course, as temporary as the system's RAM, but this is a much better match for the behaviour we require for User-Mode Linux's "emulated" RAM.

5.4 Locking the guest kernel's memory (tmpfs continued)

tmpfs gives us much better performance for the "physical" RAM requested by guest kernels because it shifts the work of caching memory pages on the host from the disc cache to the page cache.

However Linux 2.4 does not behave particularly helpfully when it comes to swapping: it will start to swap memory from *any* process when the host RAM gets around 50-60% full so that it leaves enough for the disc cache. User-Mode Linux processes have their own page and disc cache, and may find that what it considers to be "physical" memory has in fact been swapped to disc, and require swapping back in before it can use it. Of course this is invisible from inside the guest kernel, but the administrator of the guest system may find that even though he has enough RAM available for all his applications, some of his "RAM" is in fact on disc.

A fix for this situation under 2.4 host kernels is a patch we developed from one submitted by Michel Pollet[11]. It tells the guest kernel to demand "locked" memory from the host, memory which the host guarantees not to swap out. If the host cannot make this guarantee, the kernel will fail to boot. Hence using this patch will guarantee you cannot squeeze more memory from your host than is physically available.

In a quick test we achieved fractionally faster kernel compilations when running 28 User-Mode Linux instances on a host machine with its guest kernels being allocated locked memory than without. However the main benefit was to remove the frustrating interactive delays caused by the guest kernels' memory being swapped out by the host at unexpected intervals.

[11] http://marc.theaimsgroup.com/?m=3D107090239400338

When running User-Mode Linux under a 2.6 host kernel, we can achieve much the same effect without patching the guest kernel by using the host's *swappiness* parameter. This is a control in the host's /proc filesystem which allows the administrator to set how hard the kernel should try to keep applications in RAM. For User-Mode Linux systems we should get the same effect as locked memory by settings the swappiness to 0.

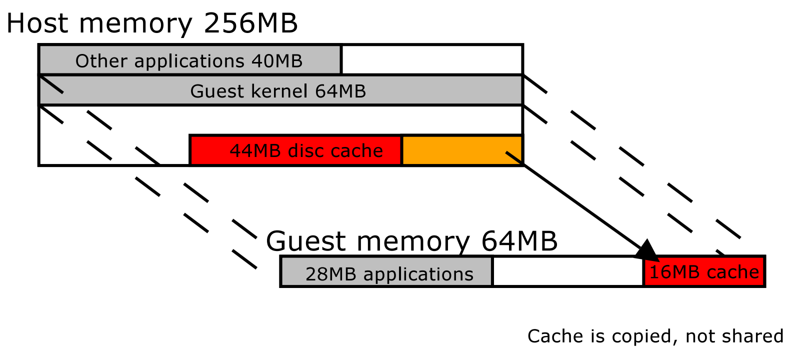
Currently we've not tested 2.6 as a host kernel, so go for the more conservative solution of requesting locked memory.

5.5 Squeezing on more kernels

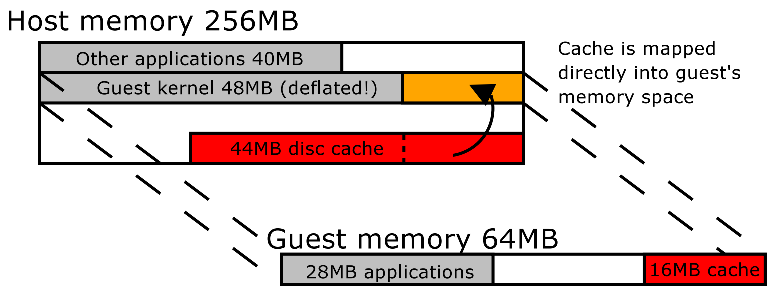
Currently at Bytemark our policy is to be "fair" to customers regarding memory usage. When they purchase a machine with 64MB RAM they expect that RAM to be real RAM and not RAM that might be swapped out; their kernels will have their own disc cache from this space and will manage as best they can.

However there is ongoing work on the User-Mode kernel to try to reduce memory usage on the host. For instance there is a relatively new host patch which will allow guest kernels to claim and free memory from their allocated range while running; in the first instance this is used to directly map pages from disc images on the host into the guest's disc cache. Once this mapping is done, the host can reclaim the memory that was previously allocated to the guest, at least while this disc mapping is in effect.

In the first instance, the guest kernel copies a read block from the host's disc cache to its own disc cache, using up the same memory twice on the host:



Using the recently-developed /dev/anon patch and the ubd=mmap option, the guest co-operates more with the host to map its cache directly into the guest's memory space, and to temporarily free the memory that had previously mapped to the guest:



While memory sharing schemes encourage overcommitting of host memory, it seems that this will be perfectly practical for guest systems running similar tasks.

The natural progression of this work is that as guest kernels share more memory with the host, they will soon not need an explicit amount of memory fixed at boot-time, but have memory mapped in and out by a daemon on the host in accordance with a particular policy or service level. This brings us completely up-to-date with the direction that User-Mode Linux development is taking.