Introduction

This article introduces some of the advantages of sharing a cohosted server machine, and specifically sharing it with User−Mode Linux (?) as a compartmentalization tool.

The intended audience are people generally comfortable with the Linux and Unix concepts of files, filesystems, and basic system administration who might be interested in providing safe root access to multiple users or want to break their services up into virtual machines (?) for security. In particular, ISP's that provide shell accounts or cohosted systems should find a new method of giving out more access to their clients in a more safe fashion.

I'll do my best to introduce the basic UML (?) concepts here, but do provide pointers to the main UML site for more information.

As we're still in the process of setting up this system, this is a work−in−progress. Expect updates from time to time.

History

I had worked in the mid 90's with Patrick Haller and Clem Yonkers at Websense.Net, a firm in northeastern Vermont that provided Internet access and a dial up masquerading Linux router/cache/firewall. When the company closed down, we went our separate ways; Clem and I stayed in the area, but Pat went to Pennsylvania to pursue his masters degree and work at an ISP there − PA.net.

He offered to host a computer there at a good price if Clem and I wanted one. We decided to take him up on his offer. Even though I've had perfectly good Internet connections at home and work, it's nice to have a place to serve up email, web sites, dns info, and free shell accounts for friends.

Clem and I started setting up the box in March of 1999. At that point it had Redhat 5.2 on a Pentium 75 (149.50 BogoMIPS!) with 64M of memory, a 13G hard drive, and a 10mbps Tulip Ethernet card.

The computer was named Slartibartfast, after the Douglas Adams character in the Hitchhiker's Guide to the Galaxy series. Both seemed old and slow, so the name fit. :−) I've long since learned that it's a pain to spell that out to people over the phone. Short names are better.

Goals

Slartibartfast has served its purpose well; it has fed out web pages, dns info, and shell accounts quite well for over 3 years. Not bad, considering that I'm pretty sure Windows NT would have refused to allow itself to even be installed on something so underpowered.

I must admit, though, that I'd like to able to run more intensive tasks on it. Freenet and the Linux Team Fortress server come to mind; both have noticeable CPU requirements. I'd like to be able to build UML root filesystems on it as well, but that takes a boatload of disk space, and a non−trivial amount of CPU and RAM; a full build of all the filesystems can take the better part of a day. I could peg Slart with any one of those three, and the UML builds alone could probably push a week or more.

None of the users on that system use it all that heavily − most use it as an email platform − but a few use it as a jumping off point for managing their own networks. Checking that a firewall lets in a certain kind of connection pretty much requires you to ssh out to some machine on the internet and then try to come back in to that port. Slart makes a good point to do that, as well as running nmap and other vulnerability checkers. *With permission*, mind you.

A few of the users had mentioned upgrading it at some point. The idea started to make more and more sense as I saw the system steadily loaded with even simple monitoring tools.

Here are the goals for the new box:

- At least a gigabyte of memory; I expect this will be running virtual machines that like lots of memory (see below)
- Large amounts of hard drive space; ideally, we'd all be able to use this to store anything we'd like, up to and possibly including backing up our home systems to it. Two of us suggested mirrored storage for at least part of the available drive space to allow us to recover if a drive dies.
- A fast processor, although I have a personal leaning that we avoid the fastest available because of the cost. Something in the 75−80% of fastest would still have plenty of horsepower. Originally, nobody had requested multiple CPU's, so we were going to buy a single processor system − but read on.
- A rack mount system so Pat − our gracious cohosting host − is happy. :−)
- The ability to handle lots of domains, email accounts, web serving, shell accounts, and any other servers we'd like. A few of us asked for a machine that could be a backup web server in case of failure in the primary; that's no problem!

New Hardware, first pass

I started to look around at replacements in the summer of 2002. Pat had been kind enough to allow us to cohost a machine in a desktop case, but non−rackmount systems make his job harder so we wanted to get a rackmount. Penguin computing, Dell, and IBM all had possibilities, but their prices were significantly higher than Eracks.com. We originally considered:

- Eracks.com/SERVE:
- Lockable 2U chassis
- AMD Athlon XP 1800, 1.53ghz
- 1.5G ram
- 3x 120G drive, with 2 software raid'ed together for 240G storage.
- Single Intel nic
- No keyboard, mouse, or monitor
- Redhat Linux 7.3

This looked beefy enough to handle the small number of people and jobs we had planned for it. Now, about *paying* for a $2640 machine...

Finances

I went back to the people that had had free acounts on Slart 1 for a while and some other friends that I thought might be interested. I asked them whether they'd be interested in splitting the cost on a replacement server, making sure it was completely clear that they could continue to use the server whether they took part in the replacement costs or not. I figured if I could get 4 others, we could split a $2640 server for $530 each.

Little did I know.

The response was completely unexpected. Almost everyone I wrote to sent back a "Yes!". Some of those I'd written also suggested other people who might be interested. Friends of friends wrote in asking if they could take part. By the time the dust settled, I had 21 partners in the new system.

Wow.

I was thrilled, but also a little worried about horsepower. Some of those users and tasks might use the system quite heavily. I sent out another proposal asking if we could make the system a little beefier. As nobody complained...

New Hardware, second pass

- Eracks.com/TDA
- Tyan Thunder K7 Dual AMD 760MP IDE
- Dual AMD Athlon MP 1900 1.60 Ghz CPU's
- 2GB DDR ECC Registered Memory
- CD−Rom drive
- 3x 120GB 7200 RPM IDE HD

  There's an additional bay for a hard drive; I installed one more 120G. By using the raid 5 provided by the 3ware card, we end up with 360G of usable storage.

  While Eracks uses Western Digital and IBM drives, we requested that they not use IBM drives in our system. One of the contributors pointed out that his personal experience had been an almost 100% failure rate in IBM drives. That, and the fact that IBM is getting out of the drive business, encouraged us to go with WD.
- 2 Ethernet ports on the motherboard
- 4 port 3ware IDE raid card
- No keyboard, mouse, or monitor
- Redhat Linux 7.3

The new system came out to $4670 base price. I figured that by the time we shipped it twice we'd be around $250 per person. Since Clem and I were splitting the monthly hosting costs without asking for any contributions to that part, if there was some left over we could apply the remainder to the hosting costs.

I ordered the system in early August 2002. When I sent out the notification, I sent along a real request for money. Realizing that people's financial situations do change, I reminded everyone that contributions were completly optional, and that any amount of contribution − including $0.00 − was enough to guarantee continued access to the old and new boxes. :−)

Remember that I'm asking for money from a pool of close friends and family. All through the process, I'm giving people easy ways to say "I'm not comfortable contributing.", and letting them know they don't have to give a reason why. I'd much rather lose them as financial contributors and keep them as friends.

There's a certain amount of risk in this. The machine has been charged to my credit card, and if large numbers of the contributors back out, I'm going to have a very expensive and very idle box on my hands. *grin* I'm not too worried, though. I know most of these people very well.

Two of the other team members were amazingly kind in offering to split any shortfall with me, and a third made an extra contribution to cover another contributor that wasn't able to do a full share. They deserve special appreciation. Many thanks, guys.

Shots of the system

The system was held up because it was initially overheating. Eracks kept it a bit longer and ordered some extra cooling for it. In the end it took about 3 weeks from initial order to when it actually shipped.

Matt Fearnow was kind enough to take some pictures of the machine. Click on a thumbnail below to get the full image.

6 muffin fans to left of silver vertical divider, 2 blue fans on cpu, and two more muffin fans at upper left behind power supply.

- Slanted chips are each 512M ECC ram. At lower left is the 3ware raid card.
- Blue fans are on top of the processors.
- 4 120G hard drives at right below the floppy and cd−rom drive.

- Fans are now covered with some special vents that force more air over the processors.
- Back panel; from left to right, ps/2 keyboard and mouse, usb, serial, parallel, vga, 2 fast ethernet and pci slot.
- Locking front cover in place.
- Front cover open.
- Would you trust this guy to sell you a used car?

Cohosting

Cohosting wasn't too much of a problem. Pat had been giving us a good price per month for hosting the original Slart, and was willing to continue to do so.

IP addresses did turn out to be a bit of an issue, however. I wanted to allow users to set up their own networking and be able to run servers. While this could have all been done with a single IP address, it would have taken quite a bit of work on the host (?) to do the masquerading and port forwarding. SMTP mail, for example, would have been a pain as there's not an easy way to specify a port other than 25 for an SMTP mail server. We'd be stuck with one mail server for everyone.

In the end, I decided to suck it up and go for an extra 32 addresses, even though that was an extra $40/month out of my pocket.

Security issues − Root for everyone!

As others needed to be able to change the system and server configuration on the original Slartibartfast from time to time (to add new dns info, virtual web sites, email domains, etc.), I had been giving out sudo access for a while to those that needed it. For a small number of accounts given to trusted friends, this isn't a problem, but we're looking at 21 people to start with, and probably more in the future. How do you give out the ability to edit system files, start new daemons, install new applications, etc., while avoiding the security and privacy problems inherent in giving out the root password or even sudo access?

User Mode Linux to the rescue.

There's a variation on the standard Linux kernel that's exactly what we need. User−Mode Linux allows one to start a second (or third, fourth...21st) linux kernel on a system. Each new kernel gets its own root filesystem (each stored as a file on the host hard drive). In this way, each UML instance is a complete virtual machine (?) that's all but indistinguishable from a real computer. All of them run as a normal user on the host, but still give root−level access, the ability to start daemons, the ability to run text and graphical applications, full networking, and almost all of the other capabilites of a stock Linux system. The only exception is that one can't directly address hardware inside UML, so the UML environment provides virtual drives, virtual network adapters, and virtual X windows displays. In practice, almost no applications (other than /sbin/hwclock and some scsi tools) access devices directly, so very few changes are needed to the root filesystem to make it work correctly under UML.

The solution is relatively straightforward, then. Everybody gets their own UML root filesystem and gets root access to it. While everybody gets an account on the host to keep file ownership straight, only the administrator (I, at the moment) actually gets to log into the host. While people inside UML's can't see what tasks are running on other UML's, the administrator can see all of them. While people can't see each other's root filesystems, the administrator can. From a trust standpoint, it does mean that the users have to trust the host administrator to not invade their privacy. For example, there's a patch to the UML kernel that allows the administrator to monitor every keystroke going into a terminal session in a UML (even if ssh is used!) and see all screen output coming back on that terminal. As the administrator, I'm promising not to use that patch on users' kernels. I will use it on a few honeypots, however.

In practice, the trust issue here is the same as if 21 people, all with their own computers, decided to cohost at an ISP. As the ISP has physical access to the boxes and the network cables, those 21 people need to trust that ISP not to break into their machines or sniff their traffic.

Performance issues

Because User−Mode Linux intercepts system calls that would normally have gone to the host kernel and may need to modify the original system call going to the host or the results coming back, there's some overhead in running UML. That's one of the reasons I wanted to get extra memory and processor power for this box; one way to compensate for this slowdown is to get better hardware on which to run it.

I don't have hard numbers for this slowdown. As there's a relatively fixed overhead for each system call, programs that call out to the kernel a lot will be hit harder than ones that do a lot of work internally and only rarely need to access the disk, network, or screen.

Jeff Dike, the primary developer of the UML project, is actively working on performance questions. I have no doubt that UML will steadily progress to the point where the performance difference will be less than a few

percent − if it's not there already. :−)

Even with the beefy box, it may still be the case that some applications may have performance or latency problems inside UML; we won't know until the box goes in and we start using it. For that reason, I'm still holding open the possibility that we'll end up running those sensitive apps right on the host OS rather than inside a UML instance.

Application issues

As a general rule, applications work under UML in exactly the same way they would under a host kernel. Command line applications, X windows applications and servers all do as you'd expect, with only a very few exceptions. Here are some notes about compatibility issues.

NTP
: The NTPD time server works just fine inside uml; you can run it in there with no problem at all. However, it turns out you don't need to. By running a single copy of NTPD on the host, all of the other UML's get their time information right from the host clock, so there's no need to run ntpd inside any virtual machine if you have it running on the host.

Half−Life
: This Linux server allows multiple (up to 32) client machines to connect; each players movements in this online game are retransmitted to all the other clients. While the individual packets tend to be small (under 100 bytes), there are a lot of them and the game tends to be very sensitive to even small latency problems. Early tests show that this setup is even more responsive than the physical machine we used to use as a server. In fact, we're running two servers; Return to Castle Wolfenstein (66.59.111.169) and Team Fortress (66.59.109.137). We're even using Network Address Translation on the host for every packet going to the Team Fortress server.

Heavy Use servers
: We're lucky at the moment in that none of our services are used all that heavily. Mail, Web, and DNS are all low load services. If any of them were heavy load servers, I'd consider moving them out to the host as well, remembering that each service I move to the host is going a bit faster, but is less secure.

Programs that access hardware directly
: There are remarkably few applications that need to talk directly to the physical hardware on a system; here's what to do about a few of them:

Application	How to handle
/sbin/clock, /sbin/hwclock	Delete and run ntpd on the host
Scanner applications	Run on the host and transfer files into a uml that needs the scan
XFree86 server (display)	Use the Xnest server and send the display to the host screen
Sound applications	Use the virtual sound device and send the audio to the host sound card

Host hardware and OS

The raid card is being used to provide redundancy for all the drives. Of the 4 120 gigabyte drives, 360GB is available storage space. Without getting too deep into the details of RAID 5, you can picture it as the fourth drive backing up the other three. This means that even the root filesystem can easily be mirrored.

One important point about RAID 5 is though it's pretty efficient (all but one of the drives is available for user storage) and reads are generally faster (up to 3X single drive speed) than reading from a single drive, it's quite a bit slower (down to 1/4 single drive speed) on disk writes. Keep that in mind if your system use includes heavy writes.

I did consider using software raid, but that has some cpu overhead and I don't yet know how to mirror the root filesystem in such a way that the system will come back up seamlessly in case of primary drive failure.

I did a very bare install of Linux on the host. The only publicly visible server on the host will be OpenSSH. All other services that need to run will be run in UML instances.

As part of the setup, I was asked how much swap space to allocate. Because the host ram+swap has to be larger than the total amount of memory allocated to all UML's, I settled on 6G of swap. That required 3 separate 2G swap partitions.

Once the host came up, I modified /etc/fstab to include the following line:

```
tmpfs   /tmp    tmpfs   defaults,size=7168M 0 0
```

which took effect on the next reboot. From that point on, all files stored in the /tmp directory actually get stored in memory, not on disk (a ramdisk, essentially, but one that can grow and shrink and strangely enough, get swapped out istself). Because the UML kernels put their entire ram images in /tmp on the host, this avoids a terrible bottleneck where every write into UML memory needs a corresponding write out to host physical disk, one which would be even worse on this raid 5 system.

If this doesn't make sense, don't worry. Just add that line to /etc/fstab on the host (putting in a number in place of 7168 that's both larger than the combined amount of UML memory and smaller than the amount of host ram+swap) and reboot the host.

Stability

For the first month or so, the host wasn't all that stable − it would crash every 2 to 12 days. We replaced Redhat's latest kernel with a custom compiled 2.4.20−pre7−ac2 kernel (see Buildkernel if you need help). That helped quite a bit, bringing us to 12 days of uptime before hitting a bug in the kernel. Based on a discussion about possible problems with AMD and SMP, I turned off the apic with the "noapic" kernel option, asked our ISP to plug in a PS/2 mouse, and had them change the bios setting to MP 1.1. We shut down Linux for a night so we could run memtest86 on it − no memory problems at all in 7 complete passes of the memory stress tester. Still, the host would crash with distressing regularity.

The clue came when one of the pa.net techs took a look in the bios setup for the machine while rebooting it. The bios has a live display of the inside temperature of the machine − it was reading 80 degrees celcius, or 176 degrees farenheit. :−(

I spoke to Eracks about it and they agreed that was too hot. The CPU's might be rated to handle that much, but the memory was not. After taking a look at a copy of this article on a mirror site, they realized that the fans in zaphod were ones they were no longer using. They mailed me out a new pair of cpu fans that night which I installed in mid−October.

That left a problem, though − how did we keep the system cool until then? The answer was quite simple − I added "nosmp" to the kernel command line in /boot/grub/grub.conf (add it to an "append" line in /etc/lilo.conf if you boot with lilo) and rebooted the box. Even though it came up in a kernel that knows how to initialize and use 2 processors, the nosmp parameter tells it to only use the first and ignore the second. My best understanding is that the second will put out little or no heat because it's not actively running programs. It'll run slower, but it kept the system running cool enough to stay stable until I could get new fans in.

With the new fans in and both CPU's enabled, the system is running at 110 degrees fahrenheit − much better. I also brought the system up to kernel 2.4.20−pre11; David Coulson and I think that kernels 2.4.20−pre9 and below have a lockup bug that appears to be fixed in 2.4.20−pre10 and higher.

I plan to have a dedicated mail server UML for those that don't want to run their own. There will also be a DNS UML, a Web UML, a Half−Life UML, etc. While this might seem like overkill, I want to compartmentalize services. If someone comes up with an exploit for the Bind name server, for example, they may be able to break into the DNS UML, maybe even getting root level access, *but only to that UML instance*. The attacker can't access the host or any of the other UML's.

There's a *possible* quirk in using an SMP machine in the host. Since signals are used to indicate that a disk block or network packet is ready to be processed, there's a potential race condition where a signal is sent to a process, but never delivered. The end result is that the UML hangs.

Jeff Dike is aware of the problem and is considering a real fix, but in the meantime there are some good workarounds. If we see this problem at all − and we hope we don't − one can nudge the UML awake again by providing it with another signal to replace the lost one. That's as simple as pinging the UML. If we see the problem at all, I plan to run a job in the background on the host that does nothing but ping each of the UML's every 5 seconds. That's a very small amount of CPU to work around a potential problem.

A more drastic workaround would be to simply disable the second CPU entirely by running a Uniprocessor kernel on the host. This would be a last gasp measure if the UMLs freeze even with the constant ping going on.

In practice, we've found that none of these workarounds has been necessary. While I did see one UML pause, to be re−awakened with a keystroke on its console, that has not been repeated.

UML − intro to the kernel

In an effort to make the process understandable, I'm going to leave out a number of technical details that aren't relevant to this process. If the low level details interest you, head over to Jeff's UML site.

Installing UML means pulling down two things; a UML kernel and a root filesystem. Both can be found at the UML site.

The kernel is run just like a normal program on the host. Instead of typing "pine" to start up my mail reader or "netscape" to start up my web browser, I type "linux" to start up a new virtual machine.

UML, Intro to the root filesystems

As the UML kernel starts it looks in the current directory for a file called "root_fs". This is the root filesystem we pulled down from Jeff's site and unpacked. If it's named something else, we can tell the UML kernel with a command line parameter:

```
linux ubd0=/home/someuser/root_fs.md82
```

On the host, this looks like a *very* large file − often in the hundreds of megabytes. What's in it?

Lets say I work at a hardware manufacturer that wants to sell systems loaded with Linux. Asking a tech to sit down in front of each machine sold and walk through the entire install over and over again is too time

consuming. Instead, I'm going to install Linux *once* in a generic way. When I'm done, I'll temporarily hook up a second larger drive to that machine. With a command like:

```
dd if=/dev/hda1 of=/mnt/largerdrive/root_fs.generic
```

I can save a copy to a file. When the next machine rolls off the assembly line, I can boot up one of the single floppy linuxes like Toms Root/boot and copy that pristine image off a CD, network drive, or second hard drive onto the new machine's main hard drive. Any last minute changes for the customer, and the system's ready to ship.

The point of the interlude is to show that the file "root_fs.generic" is a large file that contains a root filesystem − everything needed to boot a UML except for the actual kernel! The prepackaged root filesystems are fully installed systems, just like our pristine system in the factory.

Installing the needed uml files for each user

I placed all the needed files in /home/shared/uml; the UML kernel binary, the root filesystem with a few minor changes, the swap file each system will use, and an 8GB partition that each user gets for storing their files. I wrote a setup−uml−user script that installs each of these in the right place, sets things like directory permissions, and sets up a parameter file for each user.

Here's a sample session of setting up jparker, a new user, with her own virtual machine:

```
adduser jparker
setup−uml−user jparker rh73     #jparker wants to use Redhat 7.0; "de30" would mean Debian 3.0
#Pick an IP address for her, say 1.2.3.160
#edit /home/jparker/.uml/params to reflect her IP address; use FE:FD:01:02:03:A0 to match the hex
uml_go jparker
screen −S jparker −R
#log in as root on console
#change root password, by default, it's empty or "root"
#On redhat, edit /etc/sysconfig/network−scripts/ifcfg−eth0 to have the following lines (with the

DEVICE=eth0
IPADDR=1.2.3.160
NETMASK=255.255.0.0
NETWORK=1.2.0.0
BROADCAST=1.2.255.255
GATEWAY=1.2.0.1
ONBOOT=yes

/etc/rc.d/init.d/network restart

#On debian, edit /etc/network/interfaces
#FIXME − get block from Debian system
                auto eth0
                network .160
                broadcast .191
        cd /etc/rc0.d
        cp −p S35networking ../rc2.d
        ./S35networking restart


ifconfig                #Check that the eth0 interface is up
ping www.yahoo.com      #To make sure she can get out to the real world
#On Debian:
        apt−get update
```

```
        apt-get install ssh
logout
Ctrl-a   d              #detach from the screen session, but leave UML running.
ping jparker's VM from the outside world
notify user
Add all hostnames to their domain.
```

Starting UMLs

Intro to "screen"

When the host system starts up, we need to start a bunch of UMLs. Since the UMLs need a console to write to, we're going to give them a fake one; the "screen" program.

screen creates a fake terminal to which a program (like "linux") can write and remembers what should be on the display. screen and linux both run in the background if we ask them to with the "detach" command line option: "−d −m".

Lets look at an example. Say I want to start up a UML virtual machine for user "linda". I'd like to start up "linux" as her and run it inside of the screen program:

```
screen −S linda −d −m su − linda −c "cd /home/linda ; linux"
```

From left to right; create a virtual screen (screen), name it linda (−S linda), detach it from the current terminal (−d −m), and in that screen run the quoted command as linda (su − linda −c), where the quoted command changes to her home directory and runs the linux binary ("cd /home/linda ; linux").

UML now starts and grabs its root filesystem from /home/linda/root_fs . At this point we can completely forget about it. However, say something goes wrong with the boot process. We'd like to see the console for linda's UML. To do so, we reattach to that screen terminal:

```
screen −S linda −R
```

Now we're looking at the main console for that UML. If it's stopped at "filesystem corrupted, please log in as root to fix it", we can do so now. If it's gotten all the way to "login:", we can log in normally. If there's some other problem, we can see that too.

Once we're done, we can detach from this uml again by pressing "Ctrl−a" , then "d" . UML will now run in the background again.

Starting multiple UML's at boot time

Now we need to put commands in /etc/rc.d/rc.local to start up the uml's. First, we'll start up the ethernet switch which will carry packets between the host and all the UML's:

```
/etc/rc.d/rc.uml−net
```

The rc.uml−net script ends up initializing the host network interface, assigning addresses, starting up the '/usr/bin/uml_switch −tap tap0" command, and putting in firewall rules.

Now we'll start each uml. The following is a simple example; I use a script called uml_go to actually start,

stop and give the status of the umls.

```
for Session in linda bob frank gparker ; do
        screen −S $Session −d −m su − $Session −c "cd /home/$Session ; /usr/bin/linux mem=128M ub
done
```

Each screen command (4, one for each user listed) starts and almost immediately falls into the background. This means that all 4 umls will be starting up simultaneously. While you might think this would be a heavy load on the processor and disk, we'll look at some ways to reduce that load later.

The "mem=128M" parameter, as you might expect, tells the UML to ask the host for memory when it needs it, but ask for no more than 128 megabytes. A UML instance with only a small number of applications running might very well never ask for more than 20−50M. An instance with a lot of running programs or steady disk activity might very well get all 128M from the host. We'll see in a few minutes how to give the UMLs swap space as well.

It's quite legal to tell the UML's that they can take more than their fair share of memory. In our new machine, we have 21 UMLs and 1 host OS, so there are 22 machines that need to share memory. With 2GB of ram to share, each one should get 93MB. Telling the UML's that they have 128MB each isn't a problem; the host kernel will swap out unused parts of the virtual machines just like it would for any other application running on the system. In short, give the UML's approximately their fair share of the ram plus a little more to round it up to some multiple of 16M. The host and UML kernels will some to a nice balance on memory. On the other hand, if you know that some of your UML's will almost certainly not use all the ram they're allocated, feel free to give more to the others.

Finally, the "ubd0=/home/$Session/root_fs" parameter tells UML that **u**sermode **b**lock **d**evice **0** (?) (the virtual drive that will be mounted as the root filesystem) should use the file /home/linda/root_fs . It's simply making it clear that we want the UML kernel to use that file as the root filesystem.

Other filesystems

Empty filesystems

I'd like to give each of my users 8G of space to store their files. I'll give them an empty partition to store their stuff; that will make it easier if they would ever like to try a different linux distribution. The mkemptyfs utility from http://www.stearns.org/mkrootfs/ will do that for us:

```
mkemptyfs ext3 8192
```

This creates a 8GB file on the host and formats it with the ext3 filesystem. Note that the file on the host is a *sparse* file; only the blocks that actually contain something other than all 0's are actually allocated. Because most of that file is 0's, it takes up less than 200M on the host. As the users start to fill it with things, that number will increase.

This means that I can give out more drive space than I actually have. :−)

I can copy the file emptyfs.ext3.8192 to my users home directories with the command:

```
cp −p −−sparse=always emptyfs.ext3.8192 /home/linda/storage.linda
```

By using the −−sparse=always switch, the copy command only writes out blocks in the destination file if they're not all 0's. This makes the copy sparse too, so we don't use as much drive space for people who won't fill their storage space.

Swap space

As the users may be running more applications than their 128M will hold, I'd like to give them some swap space as well. Once again, mkemptyfs to the rescue:

```
mkemptyfs swap 512
```

Now copy it to users home directories with:

```
cp −p −−sparse=always emptyfs.swap.512 /home/linda/swap
```

We'll look at how to tell the uml kernel to use the storage and swap spaces in a minute.

Copy On Write − COW

Think for a minute about the root filesystem. We've got 21 users, all of whom need their own copy of a 700M filesystem. Darn, we've already chewed up almost 15G of hard drive space before anyone's even logged in. *sigh*

Could we tell all the kernels to share a single root filesystem? Not quite. Each one believes it has exclusive access to the root_fs, storage, and swap files; if two kernels tried to access a single filesystem at once, it would be corrupted in seconds. *However...*

Imagine that someone gives you a painting that's unfinished and asks you to fill in the rest. They're concerned about it as it's very valuable and ask you to avoid making any changes to the original. You take a clear sheet of plastic and place it over the original and do your painting on that. No changes are made to the original, but one can see the entire drawing by looking at the original painting with the clear plastic layer on top. Any areas you haven't modified simply show the original.

This is the idea behind copy−on−write, or COW. We want to keep a pristine root filesystem, but still allow the users to make changes to it.

The uml kernels can be told this: take /home/shared/root_fs.pristine (the equivalent of the original painting that won't be modified) as the root filesystem. If you need to make any changes to that, don't write them to that file; instead, write them to /home/linda/root_fs.cow (the equivalent of the clear plastic layer on top).

/home/linda/root_fs.cow will start out as an empty file. Once the UML kernel mounts the root filesystem read−write, it will start writing new sectors out to that file, which will start to grow.

/home/shared/root_fs.pristine should be made read−only with:

```
chmod 444 /home/shared/root_fs.pristine
```

before you start using it in a copy−on−write environment. This approach depends on that file never changing. If you accidentally mounted it read−write at some point, it would no longer be usable as a base for the cow files.

Instead of the 14.7GB we needed before, we need 700M plus a small amount for each root_fs.cow file. As each user upgrades software, makes changes to files, or otherwise makes changes to their root filesystems, their cow files will grow. Our worst case is if everyone completely fills their root filesystems with completely different content; the combined storage would be about 21 x 700M for the cow files plus the original 700M pristine file; about 15.4GB. Remember this is worst case, though; most people won't do this.

In addition to 12GB − 14GB of space savings, there's one other major advantage to using COW − speed! As I mentioned, we'll have 21 UML's starting at almost exactly the same time. Because we're sharing a single linux kernel and they're doing most of their reads out of a single pristine root filesystem, one of those UMLs will read a given sector off the host hard drive, and the other 20 get it right out of cache. If we had given each user their own copy of the root filesystem, the boot process would take about 21 times as long because every sector that needs to be read needs to be read from 21 separate roots.

Shared/hostfs

While the 8GB /home should be plenty for each user, that storage is not very conducive to file sharing. I could see it being quite reasonable that users on the system want to make files easily accessible to each other.

UML offers a special filesystem called hostfs. Inside the UML, one can mount a directory on the host; any files in the host directory show up inside the UML. Anything I save inside the UML to a hostfs mount gets saved directly to the host's drive.

One quirk is that all requests to read or write any files out to the host drive are performed as the user that started the UML. For that reason, I gave everyone their own directories they own in the shared host directories.

Providing a UML kernel with hostfs support is somewhat of a security risk. A UML user can mount arbitrary directories on the host, allowing them to at least see a lot of the host drive. In some cases you may wish to use a UML kernel without hostfs support, or at least consider runnng UML chroot'ed.

Actually using cow, swap, and storage

The command line to start up the UMLs with these three is:

```
for Session in linda bob frank gparker ; do
        screen −S $Session −d −m su − $Session −c "cd /home/$Session ; /usr/bin/linux mem=128M ub
done
```

ubd0 (?) is always assumed to be the root filesystem. The prebuilt root filesystems at the main UML site all assume that if ubd1 is specified at all, it's a swap file and mount it as such (the line "/dev/ubd/1 none swap sw" has been added to /etc/fstab inside the UML filesystem). The storage partition is not automatically mounted, but the user can make that happen automatically by adding "/dev/ubd/2 /storage auto defaults 0 2" to /etc/fstab and typing "mount /storage".

Now what about that eth0=daemon switch?

Networking and addresses

Because we started the uml_switch program and told each of the UML's to connect their virtual ethernet cards to it (with the eth0=daemon parameter), the UML's are all ready to start talking to the host and each other over

a virtual ethernet switch. They just need to be assigned addresses.

To test out the networking, pull up one of the screen sessions with "screen −S linda −R". Log in as root (if this is your first login, the root password is, ahem, "root"; make sure you change this immediately, *before* you enable networking). Type:

```
ifconfig eth0 192.168.0.10 netmask 255.255.255.0 broadcast 192.168.0.255 up
```

Now ping the host with:

```
ping 192.168.0.254
```

To set that address permanently, pull up your favourite editor inside UML and edit /etc/sysconfig/network−scripts/ifcfg−eth0 . Modify the lines in it like so (remember that each UML needs its own unique IP address):

```
IPADDR=192.168.0.10
GATEWAY=192.168.0.254
NETMASK=255.255.255.0
NETWORK=192.168.0.0
BROADCAST=192.168.0.255
ONBOOT=yes
```

To test that this still works, type:

```
/etc/rc.d/init.d/network restart
```

and ping the host again.

Think of the UML's a machines on a lan, and the host computer as being their router out to the real world. Each machine has the host as its default gateway out.

Policies

As I generally know and trust the users on the system, and given the fact that they can work independantly in their own UML's with few, if any, privacy or security concerns, there won't be strict policies. I do ask them to respect the fact that this is still a shared physical computer. Long running or processor intensive jobs should be run inside "nice". The system shouldn't be used as launching point for spam or any kind of attack (except as part of authorized penetration testing).

The users need to be reminded that they're now the system administrators for their own linux systems. They need to perform their own regular upgrades, either by hand or with the help of tools like up2date or red−carpet.

While I can't possibly explain all the intricacies of UML, I've tried to highlight the practical and relevant points of UML (in particular, how their shared machine is set up) in a Users Guide for the VM users.

Snapshots

The entire contents of the users' virtual machines is stored in two files: ~/.uml/home and ~/.uml/root_fs.cow (the original root_fs is also required − dont ever let that get modified or deleted, and keep a backup or two). To make a snapshot of a given users system, tell them to halt their VM. Once it's been successfully shut down,

start it back up with:

```
uml_go username backup start
```

Just before the VM is restarted, the uml_go script makes a dated backup copy of home and root_fs.cow in ~/backup . This copy can take quite a while, especially on a raid 5 drive, so I don't recommend doing this at every boot. You might do this on request only.

Single purpose UMLs

X

Because UML doesn't have its own physical display to which it can write, we need to use a slightly different approach to running graphical applications inside a uml.

The easiest way is to make an ssh connection into the VM and run the X windows application; for example:

```
ssh my_virtual_machine xclock
```

The xlock application starts up inside the virtual machine, using the binary inside the VM and executing in the VMs memory. However, ssh carries the screen display back to the machine from which you launched the ssh connection and shows the xclock output on that display. It also carries your keystrokes and mouse movements through the encrypted ssh connection the the xclock running inside the VM.

The second approach involves putting an entire X desktop out to your display (as oppsed to the previous approach which only shows individual applications). Here are the steps for those using prebuilt root filesystems:

- Make sure you can ping the host computer from the UML.
- If you're using one of the prebuilt root filesystems from the UML download sites, make sure you have the db1 package installed − gdm needs it but doesn't list it in its dependencies.
- On the host, type:

  ```
  xhost +192.168.0.146
  ```

  (assuming your uml's IP address is 192.168.0.146). This tells the host X windows display to accept any incoming X windows from the UML.
- Type:

  ```
  telinit 5
  ```

  on the uml. A new desktop, 3/4 of the size of your physical display, should appear on the physical diplay. Log in and off you go!

Note that both of these require a working network connection to your vm.

Chroot

Bind mounts of shared space

Transition

Doing this yourself

There's no reason why you can't do the above yourself!

You'll need the following things to make it work:

Cohosting ISP
: Which one isn't terribly important, but you'll want to take into account price, bandwidth available (and whether there are bandwidth limits or surcharges), and location. While I'm comfortable managing a machine that's 7 hours away, that does mean that I occasionally have to ask the ISP to do things for me when I'm having trouble getting to a command prompt for whatever reason. Pat and his crew have been great at coming to my rescue, but I've worked up quite a Sushi debt to them. I suppose if that debt ever got too large, a Sumo wrestler would show up at my house to break my kneecaps, or at least my chopsticks. :−)

Physical computer
: As we've shown, you can do this with nothing more than a throwaway box. Machines that are no longer fast enough to run Windows are perfectly fine with Linux. I'll bet a used machine on EBay would get people up and running while you evaluate your disk, CPU, and memory needs. If your budget is limited, focus more on getting lots of memory and at least one moderately large disk in the system before looking to get the fastest CPU; a memory starved system is going to be effectively slower than one whose CPU is fully loaded.

A community of users
: I'm lucky enough to have friends that need a cohosted machine and are technically savvy enough to make use of it. If you're sharing a machine (with or without UML), that community will need to have at least a basic level of trust in each other.

At least one moderately capable Linux system administrator
: Setting up a system like this requires some background in how to set up networking, daemons, filesytems, and operating systems. If you go with the UML approach I've described above, someone in the project needs to be familiar with − or willing to learn about − the subtleties of UML setup. There's quite a bit of documentation at the User−Mode Linux web site, and pointers to mailing lists and IRC channels for where the documentation isn't complete.

Glossary

Host
: The physical computer on which all the operating systems and programs eventually run.

Host operating system
: The normal Linux distribution that starts up the machine.

Usermode Block Device, or ubd
: The UML kernel treats a few large files stored on the host as if they were partitions on a physical disk. Any writes to or reads from /dev/ubd/N are sent out to those large files. For example, a 512M /home/linda/.uml/swap file shows up as a 512M /dev/ubd/1, on which swapped out programs are stored.

User−Mode Linux, or UML

The specially compiled kernel that sits above the Host Operating system and provides virtual machines in which additional Operating systems can be started. While the implementations differ, this is similar to what VMWare does in allowing you to start a second or third operating system on a computer.

Virtual machine, or VM

The combination of UML kernel and operating system that appears to be a normal distribution running on a normal machine, when in fact the distribution is running on top of a Host operating system. The programs running in a virtual machine are isolated from programs in other VMs and the Host.

Thanks and credits

This project would not have gotten off the ground without the help of the following people. At the moment I'm not including the names of the financial contributors as I haven't gotten their permission to mention them. I sincerely appreciate their help and enthusiasm for the project.

Patrick Haller and Pa.net have been wonderful as our cohost ISP. Over the years I've asked Pat for help from time to time with system, hardware or network problems and he's always been willing to offer his help. Many thanks, Pat.

Clem and I came up with the component parts that made up the original Slartibartfast and have split the hosting costs. Thanks, Clem!

Finally, thanks to Jeff Dike, the primary UML developer, and all the other Linux developers that make such an excellent operating system. Jeff, in particular, has been fantastic about helping diagnose and fix all kinds of problems, both uml and not.

William is an Open−Source developer, enthusiast, and advocate from New Hampshire, USA. His day job at SANS pays him to work on network security and Linux projects.

Last edited: 11/6/2002

Best viewed with something that can show web pages... <grin>