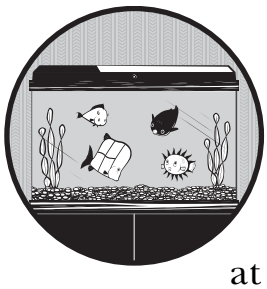
7

HOSTING UNTRUSTED USERS UNDER XEN: LESSONS FROM THE TRENCHES



Now that we've gone over the basics of Xen administration—storage, networking, provisioning, and management—let's look at applying these basics in practice. This chapter is mostly a case study of our VPS hosting firm, prgmr.com, and the lessons we've learned from renting Xen instances to the public.

The most important lesson of public Xen hosting is that the users can't be trusted to cooperate with you or each other. Some people will always try to seize as much as they can. Our focus will be on preventing this tragedy of the commons.

Advantages for the Users

There's exactly one basic reason that a user would want to use a Xen VPS rather than paying to colocate a box in your data center: it's cheap, especially for someone who's just interested in some basic services, rather than massive raw performance.

GRID COMPUTING AND VIRTUALIZATION

One term that you hear fairly often in connection with Xen is *grid computing*. The basic idea behind grid computing is that you can quickly and automatically provision and destroy nodes. Amazon's EC2 service is a good example of a grid computing platform that allows you to rent Linux servers by the hour.

Grid computing doesn't require virtualization, but the two concepts are fairly closely linked. One could design a system using physical machines and PXEboot for fast, easy, automated provisioning without using Xen, but a virtualization system would make the setup more lightweight, agile, and efficient.

There are several open source projects that are attempting to create a standard and open interface to provision "grid computing" resources. One such project is Eucalyptus (*http://www.eucalyptus.com/*). We feel that standard frameworks like this—that allow you to easily switch between grid computing providers—are essential if "the grid" is to survive.

Xen also gives users nearly all the advantages they'd get from colocating a box: their own publicly routed network interface, their own disk, root access, and so forth. With a 128MB VM, they can run DNS, light mail service, a web server, IRC, SSH, and so on. For lightweight services like these, the power of the box is much less important than its basic existence—just having something available and publicly accessible makes life more convenient.

You also have the basic advantages of virtualization, namely, that hosting one server with 32GB of RAM is a whole lot cheaper than hosting 32 servers with 1GB of RAM each (or even 4 servers with 8GB RAM each). In fact, the price of RAM being what it is, I would argue that it's difficult to even economically justify hosting a general-purpose server with less than 32GB of RAM.

The last important feature of Xen is that, relative to other virtualization systems, it's got a good combination of light weight, strong partitioning, and robust resource controls. Unlike some other virtualization options, it's consistent—a user can rely on getting exactly the amount of memory, disk space, and network bandwidth that he's signed up for and approximately as much CPU and disk bandwidth.

Shared Resources and Protecting Them from the Users

> Xen's design is congruent to good security.
>
> —Tavis Ormandy, *http://taviso.decsystem.org/virtsec.pdf*

It's a ringing endorsement, by security-boffin standards. By and large, with Xen, we're not worried about keeping people from breaking out of their virtual machines—Xen itself is supposed to provide an appropriate level of isolation. In paravirtualized mode, Xen doesn't expose hardware drivers to

domUs, which eliminates one major attack vector.[1] For the most part, securing a dom0 is exactly like securing any other server, except in one area.

That area of possible concern is in the access controls for shared resources, which are not entirely foolproof. The primary worry is that malicious users could gain more resources than they're entitled to, or in extreme cases cause denial-of-service attacks by exploiting flaws in Xen's accounting. In other words, we are in the business of enforcing performance isolation, rather than specifically trying to protect the dom0 from attacks via the domUs.

Most of the resource controls that we present here are aimed at users who aren't necessarily malicious—just, perhaps, exuberant.

Tuning CPU Usage

The first shared resource of interest is the CPU. While memory and disk size are easy to tune—you can just specify memory in the config file, while disk size is determined by the size of the backing device—fine-grained CPU allocation requires you to adjust the scheduler.

Scheduler Basics

The Xen scheduler acts as a referee between the running domains. In some ways it's a lot like the Linux scheduler: It can preempt processes as needed, it tries its best to ensure fair allocation, and it ensures that the CPU wastes as few cycles as possible. As the name suggests, Xen's scheduler schedules domains to run on the physical CPU. These domains, in turn, schedule and run processes from their internal run queues.

Because the dom0 is just another domain as far as Xen's concerned, it's subject to the same scheduling algorithm as the domUs. This can lead to trouble if it's not assigned a high enough weight because the dom0 has to be able to respond to I/O requests. We'll go into more detail on that topic a bit later, after we describe the general procedures for adjusting domain weights.

Xen can use a variety of scheduling algorithms, ranging from the simple to the baroque. Although Xen has shipped with a number of schedulers in the past, we're going to concentrate on the *credit scheduler*; it's the current default and recommended choice and the only one that the Xen team has indicated any interest in keeping.

The `xm dmesg` command will tell you, among other things, what scheduler Xen is using.

```
# xm dmesg | grep scheduler
(XEN) Using scheduler: SMP Credit Scheduler (credit)
```

If you want to change the scheduler, you can set it as a boot parameter—to change to the SEDF scheduler, for example, append `sched=sedf` to the kernel line in GRUB. (That's the Xen kernel, not the dom0 Linux kernel loaded by the first `module` line.)

[1] In HVM mode, the emulated QEMU devices are something of a risk, which is part of why we don't offer HVM domains.

VCPUs and Physical CPUs

For convenience, we consider each Xen domain to have one or more virtual CPUs (VCPUs), which periodically run on the physical CPUs. These are the entities that consume credits when run. To examine VCPUs, use `xm vcpu-list <domain>`:

```
# xm vcpu-list horatio
Name                              ID VCPUs   CPU State   Time(s) CPU Affinity
horatio                           16     0     0   ---  140005.6 any cpu
horatio                           16     1     2   r--  139968.3 any cpu
```

In this case, the domain has two VCPUs, 0 and 1. VCPU 1 is in the *running* state on (physical) CPU 1. Note that Xen will try to spread VCPUs across CPUs as much as possible. Unless you've pinned them manually, VCPUs can occasionally switch CPUs, depending on which physical CPUs are available.

To specify the number of VCPUs for a domain, specify the `vcpus=` directive in the config file. You can also change the number of VCPUs while a domain is running using `xm vcpu-set`. However, note that you can decrease the number of VCPUs this way, but you can't increase the number of VCPUs beyond the initial count.

To set the CPU affinity, use `xm vcpu-pin <domain> <vcpu> <pcpu>`. For example, to switch the CPU assignment in the domain *horatio*, so that VCPU0 runs on CPU2 and VCPU1 runs on CPU0:

```
# xm vcpu-pin horatio 0 2
# xm vcpu-pin horatio 1 0
```

Equivalently, you can pin VCPUs in the domain config file (*/etc/xen/horatio*, if you're using our standard naming convention) like this:

```
vcpus=2
cpus=[0,2]
```

This gives the domain two VCPUs, pins the first VCPU to the first physical CPU, and pins the second VCPU to the third physical CPU.

Credit Scheduler

The Xen team designed the credit scheduler to minimize wasted CPU time. This makes it a *work-conserving* scheduler, in that it tries to ensure that the CPU will always be working whenever there is work for it to do.

As a consequence, if there is more real CPU available than the domUs are demanding, all domUs get all the CPU they want. When there is contention—that is, when the domUs in aggregate want more CPU than actually exists—then the scheduler arbitrates fairly between the domains that want CPU.

Xen does its best to do a fair division, but the scheduling isn't perfect by any stretch of the imagination. In particular, cycles spent servicing I/O by domain 0 are not charged to the responsible domain, leading to situations where I/O-intensive clients get a disproportionate share of CPU usage.

Nonetheless, you can get pretty good allocation in nonpathological cases. (Also, in our experience, the CPU sits idle most of the time anyway.)

The credit scheduler assigns each domain a *weight* and, optionally, a *cap*. The weight indicates the relative CPU allocation of a domain—if the CPU is scarce, a domain with a weight of 512 will receive twice as much CPU time as a domain with a weight of 256 (the default). The cap sets an absolute limit on the amount of CPU time a domain can use, expressed in hundredths of a CPU. Note that the CPU cap can exceed 100 on multiprocessor hosts.

The scheduler transforms the weight into a *credit* allocation for each VCPU, using a separate accounting thread. As a VCPU runs, it consumes credits. If a VCPU runs out of credits, it only runs when other, more thrifty VCPUs have finished executing, as shown in Figure 7-1. Periodically, the accounting thread goes through and gives everybody more credits.

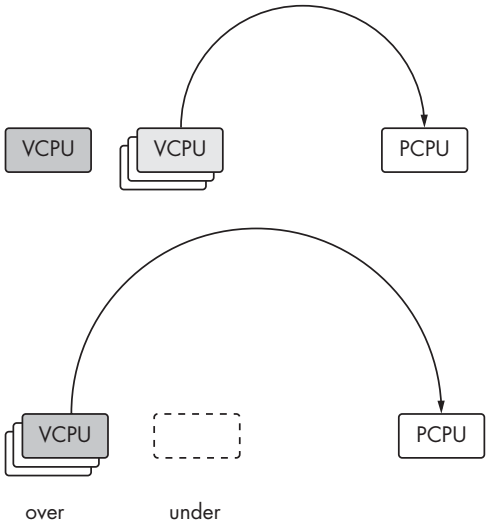


Figure 7-1: VCPUs wait in two queues: one for VCPUs with credits and the other for those that are over their allotment. Once the first queue is exhausted, the CPU will pull from the second.

In this case, the details are probably less important than the practical application. Using the `xm sched-credit` commands, we can adjust CPU allocation on a per-domain basis. For example, here we'll increase a domain's CPU allocation. First, to list the weight and cap for the domain horatio:

```
# xm sched-credit -d horatio
{'cap': 0, 'weight': 256}
```

Then, to modify the scheduler's parameters:

```
# xm sched-credit -d horatio -w 512
# xm sched-credit -d horatio
{'cap': 0, 'weight': 512}
```

Of course, the value "512" only has meaning relative to the other domains that are running on the machine. Make sure to set all the domains' weights appropriately.

To set the cap for a domain:

```
# xm sched-credit -d domain -c cap
```

Scheduling for Providers

We decided to divide the CPU along the same lines as the available RAM—it stands to reason that a user paying for half the RAM in a box will want more CPU than someone with a 64MB domain. Thus, in our setup, a customer with 25 percent of the RAM also has a minimum share of 25 percent of the CPU cycles.

The simple way to do this is to assign each CPU a weight equal to the number of megabytes of memory it has and leave the cap empty. The scheduler will then handle converting that into fair proportions. For example, our aforementioned user with half the RAM will get about as much CPU time as the rest of the users put together.

Of course, that's the worst case; that is what the user will get in an environment of constant struggle for the CPU. Idle domains will automatically yield the CPU. If all domains but one are idle, that one can have the entire CPU to itself.

NOTE *It's essential to make sure that the dom0 has sufficient CPU to service I/O requests. You can handle this by dedicating a CPU to the dom0 or by giving the dom0 a very high weight—high enough to ensure that it never runs out of credits. At prgmr.com, we handle the problem by weighting each domU with its RAM amount and weighting the dom0 at 6000.*

This simple weight = memory formula becomes a bit more complex when dealing with multiprocessor systems because independent systems of CPU allocation come into play. A good rule would be to allocate VCPUs in proportion to memory (and therefore in proportion to weight). For example, a domain with half the RAM on a box with four cores (and hyperthreading turned off) should have at least two VCPUs. Another solution would be to give all domains as many VCPUs as physical processors in the box—this would allow all domains to burst to the full CPU capacity of the physical machine but might lead to increased overhead from context swaps.

Controlling Network Resources

Network resource controls are, frankly, essential to any kind of shared hosting operation. Among the many lessons that we've learned from Xen hosting has been that if you provide free bandwidth, some users will exploit it for all it's worth. This isn't a Xen-specific observation, but it's especially noticeable with the sort of cheap VPS hosting Xen lends itself to.

We prefer to use `network-bridge`, since that's the default. For a more thorough look at `network-bridge`, take a look at Chapter 5.

Monitoring Network Usage

Given that some users will consume as much bandwidth as possible, it's vital to have some way to monitor network traffic.[2]

To monitor network usage, we use BandwidthD on a physical SPAN port. It's a simple tool that counts bytes going through a switch—nothing Xen-specific here. We feel comfortable doing this because our provider doesn't allow anything but IP packets in or out, and our antispoof rules are good enough to protect us from users spoofing their IP on outgoing packets.

A similar approach would be to extend the *dom0 is a switch* analogy and use SNMP monitoring software. As mentioned in Chapter 5, it's important to specify a `vifname` for each domain if you're doing this. In any case, we'll leave the particulars of bandwidth monitoring up to you.

ARP CACHE POISONING

If you use the default `network-bridge` setup, you are vulnerable to ARP cache poisoning, just as on any layer 2 switch.

The idea is that the interface counters on a layer 2 switch—such as the virtual switch used by `network-bridge`—watch traffic as it passes through a particular port. Every time a switch sees an Ethernet frame or ARP is-at, it keeps track of what port and MAC it came from. If it gets a frame destined for a MAC address in its cache, it sends that frame down the proper port (and only the proper port). If the bridge sees a frame destined for a MAC that is not in the cache, it sends that frame to all ports.*

Clever, no? In most cases this means that you almost never see Ethernet frames destined for other MAC addresses (other than broadcasts, etc.). However, this feature is designed purely as an optimization, not a security measure. As those of you with cable providers who do MAC address verification know quite well, it is fairly trivial to fake a MAC address. This means that a malicious user can fill the (limited in size) ARP cache with bogus MAC addresses, drive out the good data, and force all packets to go down all interfaces. At this point the switch becomes basically a hub, and the counters on all ports will show all traffic for any port.

There are two ways we have worked around the problem. You could use Xen's `network-route` networking model, which doesn't use a virtual bridge. The other approach is to ignore the interface counters and use something like BandwidthD, which bases its accounting on IP packets.

* We are using the words *port* and *interface* interchangeably here. This is a reasonable simplification in the context of interface counters on an SNMP-capable switch.

Once you can examine traffic quickly, the next step is to shape the users. The principles for network traffic shaping and policing are the same as for standalone boxes, except that you can also implement policies on the Xen host. Let's look at how to limit both incoming and outgoing traffic for a particular interface—as if, say, you have a customer who's going over his bandwidth allotment.

[2] In this case, we're talking about bandwidth monitoring. You should also run some sort of IDS, such as Snort, to watch for outgoing abuse (we do) but there's nothing Xen-specific about that.

Network Shaping Principles

The first thing to know about shaping is that it only works on outgoing traffic. Although it is possible to *police* incoming traffic, it isn't as effective. Fortunately, both directions look like outgoing traffic at some point in their passage through the dom0, as shown in Figure 7-2. (When we refer to outgoing and incoming traffic in the following description, we mean from the perspective of the domU.)

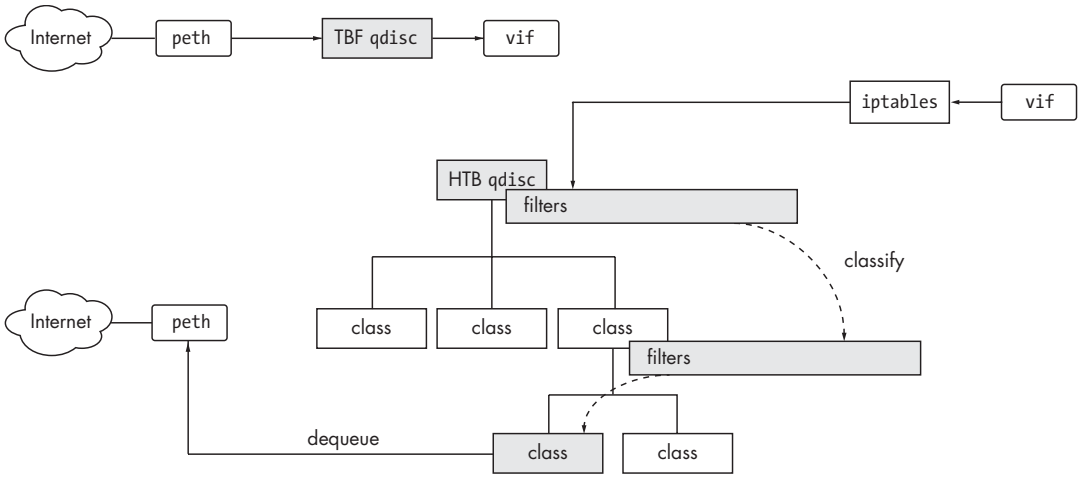


Figure 7-2: Incoming traffic comes from the Internet, goes through the virtual bridge, and gets shaped by a simple nonhierarchical filter. Outgoing traffic, on the other hand, needs to go through a system of filters that assign packets to classes in a hierarchical queuing discipline.

Shaping Incoming Traffic

We'll start with incoming traffic because it's much simpler to limit than outgoing traffic. The easiest way to shape incoming traffic is probably the *token bucket filter* queuing discipline, which is a simple, effective, and lightweight way to slow down an interface.

The token bucket filter, or TBF, takes its name from the metaphor of a bucket of tokens. Tokens stream into the bucket at a defined and constant rate. Each byte of data sent takes one token from the bucket and goes out immediately—when the bucket's empty, data can only go as tokens come in. The bucket itself has a limited capacity, which guarantees that only a reasonable amount of data will be sent out at once. To use the TBF, we add a `qdisc` (*queuing discipline*) to perform the actual work of traffic limiting. To limit the virtual interface `osric` to 1 megabit per second, with bursts up to 2 megabits and maximum allowable latency of 50 milliseconds:

```
# tc qdisc add dev osric root tbf rate 1mbit latency 50ms peakrate 2mbit maxburst 40MB
```

This adds a `qdisc` to the device `osric`. The next arguments specify where to add it (`root`) and what sort of `qdisc` it is (`tbf`). Finally, we specify the rate,

latency, burst rate, and amount that can go at burst rate. These parameters correspond to the token flow, amount of latency the packets are allowed to have (before the driver signals the operating system that its buffers are full), maximum rate at which the bucket can empty, and the size of the bucket.

Shaping Outgoing Traffic

Having shaped incoming traffic, we can focus on limiting outgoing traffic. This is a bit more complex because the outgoing traffic for all domains goes through a single interface, so a single token bucket won't work. The policing filters might work, but they handle the problem by dropping packets, which is . . . bad. Instead, we're going to apply traffic shaping to the outgoing physical Ethernet device, peth0, with a *Hierarchical Token Bucket*, or HTB `qdisc`.

The HTB discipline acts like the simple token bucket, but with a hierarchy of buckets, each with its own rate, and a system of filters to assign packets to buckets. Here's how to set it up.

First, we have to make sure that the packets on Xen's virtual bridge traverse `iptables`:

```
# echo 1 > /proc/sys/net/bridge/bridge-nf-call-iptables
```

This is so that we can mark packets according to which domU emitted them. There are other reasons, but that's the important one in terms of our traffic-shaping setup. Next, for each domU, we add a rule to mark packets from the corresponding network interface:

```
# iptables -t mangle -A FORWARD -m physdev --physdev-in baldr -j MARK --set-mark 5
```

Here the number 5 is an arbitrary mark—it's not important what the number is, as long as there's a useful mapping between number and domain. We're using the domain ID. We could also use `tc` filters directly that match on source IP address, but it feels more elegant to have everything keyed to the domain's physical network device. Note that we're using `physdev-in`—traffic that goes out from the domU comes in to the dom0, as Figure 7-3 shows.

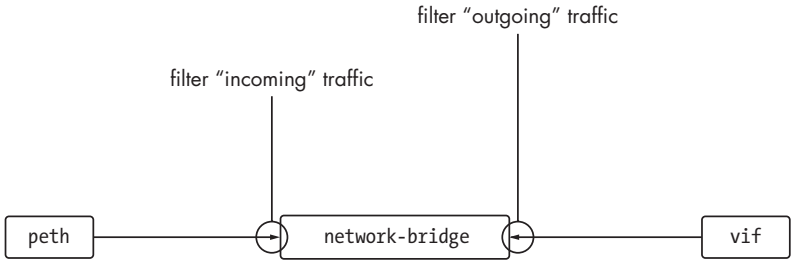


Figure 7-3: We shape traffic coming into the domU as it comes into the dom0 from the physical device, and shape traffic leaving the domU as it enters the dom0 on the virtual device.

Next we create a HTB `qdisc`. We won't go over the HTB options in too much detail—see the documentation at *http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm* for more details:

```
# tc qdisc add dev peth0 root handle 1: htb default 12
```

Then we make some classes to put traffic into. Each class will get traffic from one domU. (As the HTB docs explain, we're also making a parent class so that they can share surplus bandwidth.)

```
# tc class add dev peth0 parent 1: classid 1:1 htb rate 100mbit
# tc class add dev peth0 parent 1:1 classid 1:2 htb rate 1mbit
```

Now that we have a class for our domU's traffic, we need a filter that will assign packets to it.

```
# tc filter add dev peth0 protocol ip parent 1:0 prio 1 handle 5 fw flowid 1:2
```

Note that we're matching on the "handle" that we set earlier using `iptables`. This assigns the packet to the 1:2 class, which we've previously limited to 1 megabit per second.

At this point traffic to and from the target domU is essentially shaped, as demonstrated by Figure 7-4. You can easily add commands like these to the end of your `vif` script, be it `vif-bridge`, `vif-route`, or a wrapper. We would also like to emphasize that this is only an example and that the Linux Advanced Routing and Traffic Control how-to at *http://lartc.org/* is an excellent place to look for further documentation. The `tc` man page is also informative.

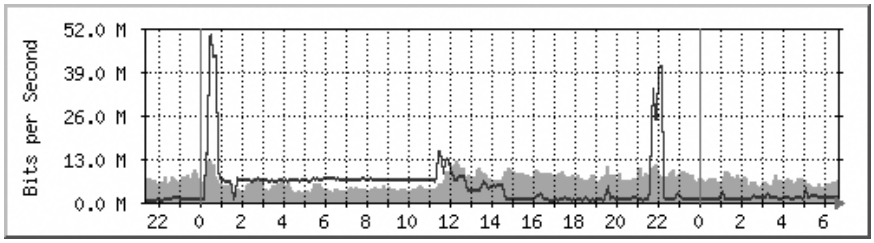


Figure 7-4: The effect of the shaping filters

Storage in a Shared Hosting Environment

As with so much else in system administration, a bit of planning can save a lot of trouble. Figure out beforehand where you're going to store pristine filesystem images, where configuration files go, and where customer data will live.

For pristine images, there are a lot of conventions—some people use */diskimages*, some use */opt/xen*, */var/xen* or similar, some use a subdirectory of */home*. Pick one and stick with it.

Configuration files should, without exception, go in */etc/xen*. If you don't give `xm create` a full path, it'll look for the file in */etc/xen*. Don't disappoint it.

As for customer data, we recommend that serious hosting providers use LVM. This allows greater flexibility and manageability than blktap-mapped files while maintaining good performance. Chapter 4 covers the details of working with LVM (or at least enough to get started), as well as many other available storage options and their advantages. Here we're confining ourselves to lessons that we've learned from our adventures in shared hosting.

Regulating Disk Access with ionice

One common problem with VPS hosting is that customers—or your own housekeeping processes, like backups—will use enough I/O bandwidth to slow down everyone on the machine. Furthermore, I/O isn't really affected by the scheduler tweaks discussed earlier. A domain can request data, hand off the CPU, and save its credits until it's notified of the data's arrival.

Although you can't set hard limits on disk access rates as you can with the network QoS, you can use the `ionice` command to prioritize the different domains into subclasses, with a syntax like:

```
# ionice -p <PID> -c <class> -n <priority within class>
```

Here `-n` is the knob you'll ordinarily want to twiddle. It can range from 0 to 7, with lower numbers taking precedence.

We recommend always specifying 2 for the class. Other classes exist—3 is idle and 1 is realtime—but idle is extremely conservative, while realtime is so aggressive as to have a good chance of locking up the system. The within-class priority is aimed at proportional allocation, and is thus much more likely to be what you want.

Let's look at `ionice` in action. Here we'll test `ionice` with two different domains, one with the highest normal priority, the other with the lowest.

First, `ionice` only works with the CFQ I/O scheduler. To check that you're using the CFQ scheduler, run this command in the dom0:

```
# cat /sys/block/[sh]d[a-z]*/queue/scheduler
noop anticipatory deadline [cfq]
noop anticipatory deadline [cfq]
```

The word in brackets is the selected scheduler. If it's not `[cfq]`, reboot with the parameter `elevator = cfq`.

Next we find the processes we want to `ionice`. Because we're using `tap:aio` devices in this example, the dom0 process is `tapdisk`. If we were using `phy:` devices, it'd be `[xvd <domain id> <device specifier>]`.

```
# ps aux | grep tapdisk
root      1054   0.5   0.0   13588   556   ?   Sl   05:45   0:10   tapdisk
/dev/xen/tapctrlwrite1 /dev/xen/tapctrlread1
root      1172   0.6   0.0   13592   560   ?   Sl   05:45   0:10   tapdisk
/dev/xen/tapctrlwrite2 /dev/xen/tapctrlread2
```

Now we can `ionice` our domains. Note that the numbers of the `tapctrl` devices correspond to the order the domains were started in, not the domain ID.

```
# ionice -p 1054 -c 2 -n 7
# ionice -p 1172 -c 2 -n 0
```

To test `ionice`, let's run a couple of Bonnie++ processes and time them. (After Bonnie++ finishes, we `dd` a load file, just to make sure that conditions for the other domain remain unchanged.)

```
prio 7 domU tmp # /usr/bin/time -v  bonnie++  -u 1 && dd if=/dev/urandom of=load
prio 0 domU tmp # /usr/bin/time -v  bonnie++  -u 1 && dd if=/dev/urandom of=load
```

In the end, according to the wall clock, the domU with priority 0 took 3:32.33 to finish, while the priority 7 domU needed 5:07.98. As you can see, the `ionice` priorities provide an effective way to do proportional I/O allocation.

The best way to apply `ionice` is probably to look at CPU allocations and convert them into priority classes. Domains with the highest CPU allocation get priority 1, next highest priority 2, and so on. Processes in the dom0 should be ioniced as appropriate. This will ensure a reasonable priority, but not allow big domUs to take over the entirety of the I/O bandwidth.

Backing Up DomUs

As a service provider, one rapidly learns that customers don't do their own backups. When a disk fails (not *if*—*when*), customers will expect you to have complete backups of their data, and they'll be very sad if you don't. So let's talk about backups.

Of course, you already have a good idea how to back up physical machines. There are two aspects to backing up Xen domains: First, there's the domain's virtual disk, which we want to back up just as we would a real machine's disk. Second, there's the domain's running state, which can be saved and restored from the dom0. Ordinarily, our use of *backup* refers purely to the disk, as it would with physical machines, but with the advantage that we can use domain snapshots to pause the domain long enough to get a clean disk image.

We use `xm save` and LVM snapshots to back up both the domain's storage and running state. LVM snapshots aren't a good way of implementing full copy-on-write because they handle the "out of snapshot space" case poorly, but they're excellent if you want to preserve a filesystem state long enough to make a consistent backup.

Our implementation copies the entire disk image using either a plain `cp` (in the case of file-backed domUs) or `dd` (for `phy:` devices). This is because we very much want to avoid mounting a possibly unclean filesystem in the dom0, which can cause the entire machine to panic. Besides, if we do a raw device backup, domU administrators will be able to use filesystems (such as ZFS on an OpenSolaris domU) that the dom0 cannot read.

An appropriate script to do as we've described might be:

```
#!/usr/bin/perl
my @disks,@stores,@files,@lvs;

$domain=$ARGV[0];

my $destdir="/var/backup/xen/${domain}/";
system "mkdir -p $destdir";

open (FILE, "/etc/xen/$domain") ;
while (<FILE>) {
        if(m/^disk/) {
                s/.*\[\s+([^\]]+)\s*\].*/\1/;
                @disks = split(/[,]/);

                # discard elements without a :, since they can't be
                # backing store specifiers
                while($disks[$n]) {
                        $disks[$n] =~ s/['"]//g;
                        push(@stores,"$disks[$n]") if("$disks[$n]"=~ m/:/);
                        $n++;
                }
                $n=0;

                # split on : and take only the last field if the first
                # is a recognized device specifier.
                while($stores[$n]) {
                        @tmp = split(/:/, $stores[$n]);
                        if(($tmp[0] =~ m/file/i) || ($tmp[0] =~ m/tap/i)) {
                                push(@files, $tmp[$#tmp]);
                        }
                        elsif($tmp[0] =~ m/phy/i) {
                                push(@lvs, $tmp[$#tmp]);
                        }
                        $n++;
                }
        }
}
close FILE;

print "xm save $domain $destdir/${domain}.xmsave\n";
system ("xm save $domain $destdir/${domain}.xmsave");

foreach(@files) {
    print "copying $_";
        system("cp $_ ${destdir}") ;
}

foreach $lv (@lvs) {
        system("lvcreate --size 1024m --snapshot --name ${lv}_snap $lv");
}

system ("xm restore $destdir/${domain}.xmsave && gzip $destdir/${domain}.xmsave");
```

```
foreach $lv (@lvs) {
    $lvfile=$lv;
    $lvfile=~s/\//_/g;
    print "backing up $lv";
        system("dd if=${lv}_snap | gzip -c > $destdir/${lvfile}.gz" ) ;
        system("lvremove ${lv}_snap" );
}
```

Save it as, say, */usr/sbin/backup_domains.sh* and tell `cron` to execute the script at appropriate intervals.

This script works by saving each domain, copying file-based storage, and snapshotting LVs. When that's accomplished, it restores the domain, backs up the save file, and backs up the snapshots via `dd`.

Note that users will see a brief hiccup in service while the domain is paused and snapshotted. We measured downtime of less than three minutes to get a consistent backup of a domain with a gigabyte of RAM—well within acceptable parameters for most applications. However, doing a bit-for-bit copy of an entire disk may also degrade performance somewhat.[3] We suggest doing backups at off-peak hours.

To view other scripts in use at prgmr.com, go to *http://book.xen.prgmr.com/*.

Remote Access to the DomU

The story on normal access for VPS users is deceptively simple: The Xen VM is exactly like a normal machine at the colocation facility. They can SSH into it (or, if you're providing Windows, `rdesktop`). However, when problems come up, the user is going to need some way of accessing the machine at a lower level, as if they were sitting at their VPS's console.

For that, we provide a console server that they can SSH into. The easiest thing to do is to use the dom0 as their console server and sharply limit their accounts.

NOTE *Analogously, we feel that any colocated machine should have a serial console attached to it.[4] We discuss our reasoning and the specifics of using Xen with a serial console in Chapter 14.*

An Emulated Serial Console

Xen already provides basic serial console functionality via `xm`. You can access a guest's console by typing `xm console <domain>` within the dom0. Issue commands, then type CTRL-] to exit from the serial console when you're done.

The problem with this approach is that `xm` has to run from the dom0 with effective UID 0. While this is reasonable enough in an environment with trusted domU administrators, it's not a great idea when you're giving an

[3] Humorous understatement.

[4] Our experience with other remote console tools has, overall, been unpleasant. Serial redirection systems work quite well. IP KVMs are barely preferable to toggling in the code on the front panel. On a good day.

account to anyone with $5. Dealing with untrusted domU admins, as in a VPS hosting situation, requires some additional work to limit access using `ssh` and `sudo`.

First, configure `sudo`. Edit `/etc/sudoers` and append, for each user:

```
<username> ALL=NOPASSWD:/usr/sbin/xm console <vm name>
```

Next, for each user, we create a *~/.ssh/authorized_keys* file like this:

```
no-agent-forwarding,no-X11-forwarding,no-port-forwarding,command="sudo xm
console <vm name>" ssh-rsa <key> [comment]
```

This line allows the user to log in with his key. Once he's logged in, `sshd` connects to the named domain console and automatically presents it to him, thus keeping domU administrators out of the dom0. Also, note the options that start with `no`. They're important. We're not in the business of providing shell accounts. This is purely a console server—we want people to use their domUs rather than the dom0 for standard SSH stuff. These settings will allow users to access their domains' consoles via SSH in a way that keeps their access to the dom0 at a minimum.

A Menu for the Users

Of course, letting each user access his console is really just the beginning. By changing the `command` field in `authorized_keys` to a custom script, we can provide a menu with a startling array of features!

Here's a sample script that we call *xencontrol*. Put it somewhere in the filesystem—say `/usr/bin/xencontrol`—and then set the line in `authorized_keys` to call `xencontrol` rather than `xm console`.

```
#!/bin/bash
DOM="$1"
cat << EOF
`sudo /usr/sbin/xm list $DOM`


Options for $DOM
1. console
2. create/start
3. shutdown
4. destroy/hard shutdown
5. reboot
6. exit
EOF
printf "> "
read X
case "$X" in
```

```
*1*) sudo /usr/sbin/xm console "$DOM" ;;
*2*) sudo /usr/sbin/xm create -c "$DOM" ;;
*3*) sudo /usr/sbin/xm shutdown "$DOM" ;;
*4*) sudo /usr/sbin/xm destroy "$DOM" ;;
*5*) sudo /usr/sbin/xm reboot "$DOM" ;;
esac
```

When the user logs in via SSH, the SSH daemon runs this script in place of the user's login shell (which we recommend setting to `/bin/false` or its equivalent on your platform). The script then echoes some status information, an informative message, and a list of options. When the user enters a number, it runs the appropriate command (which we've allowed the user to run by configuring `sudo`).

PyGRUB, a Bootloader for DomUs

Up until now, the configurations that we've described, by and large, have specified the domU's boot configuration in the config file, using the `kernel`, `ramdisk`, and `extra` lines. However, there is an alternative method, which specifies a `bootloader` line in the config file and in turn uses that to load a kernel from the domU's filesystem.

The bootloader most commonly used is PyGRUB, or Python GRUB. The best way to explain PyGRUB is probably to step back and examine the program it's based on, GRUB, the GRand Unified Bootloader. GRUB itself is a traditional bootloader—a program that sits in a location on the hard drive where the BIOS can load and execute it, which then itself loads and executes a kernel.

PyGRUB, therefore, is like GRUB for a domU. The Xen domain builder usually loads an OS kernel directly from the dom0 filesystem when the virtual machine is started (therefore acting like a bootloader itself). Instead, it can load PyGRUB, which then acts as a bootloader and loads the kernel from the domU filesystem.[5]

PyGRUB is useful because it allows a more perfect separation between the administrative duties of the dom0 and the domU. When virtualizing the data center, you want to hand off virtual hardware to the customer. PyGRUB more effectively virtualizes the hardware. In particular, this means the customer can change his own kernel without the intervention of the dom0 administrator.

NOTE *PyGRUB has been mentioned as a possible security risk because it reads an untrusted filesystem directly from the dom0. PV-GRUB (see "PV-GRUB: A Safer Alternative to PyGRUB?" on page 105), which loads a trusted paravirtualized kernel from the dom0 then uses that to load and jump to the domU kernel, should improve this situation.*

[5] This is an oversimplification. What actually happens is that PyGRUB copies a kernel from the domU filesystem, puts it in */tmp*, and then writes an appropriate domain config so that the domain builder can do its job. But the distinction is usually unimportant, so we've opted to approach PyGRUB as the bootloader it pretends to be.

PV-GRUB: A SAFER ALTERNATIVE TO PYGRUB?

PV-GRUB is an excellent reason to upgrade to Xen 3.3. The problem with PyGRUB is that while it's a good simulation of a bootloader, it has to mount the domU partition in the dom0, and it interacts with the domU filesystem. This has led to at least one remote-execution exploit. PV-GRUB avoids the problem by loading an executable that is, quite literally, a paravirtualized version of the GRUB bootloader, which then runs entirely within the domU.

This also has some other advantages. You can actually load the PV-GRUB binary from within the domU, meaning that you can load your first *menu.lst* from a read-only partition and have it fall through to a user partition, which then means that unlike my PyGRUB setup, users can never mess up their *menu.lst* to the point where they can't get into their rescue image.

Note that Xen creates a domain in either 32- or 64-bit mode, and it can't switch later on. This means that a 64-bit PV-GRUB can't load 32-bit Linux kernels, and vice versa.

Our PV-GRUB setup at prgmr.com starts with a normal `xm` config file, but with no bootloader and a `kernel=` line that points to PV-GRUB, instead of the domU kernel.

```
kernel = "/usr/lib/xen/boot/pv-grub-x86_64.gz"
extra = "(hd0,0)/boot/grub/menu.lst"
disk = ['phy:/dev/denmark/horatio,xvda,w','phy:/dev/denmark/rescue,xvde,r']
```

Note that we call the architecture-specific binary for PV-GRUB. The 32-bit (PAE) version is *pv-grub-x86_32*.

This is enough to load a regular *menu.lst*, but what about this indestructible rescue image of which I spoke? Here's how we do it on the new prgmr.com Xen 3.3 servers. In the `xm` config file:

```
kernel = "/usr/lib/xen/boot/pv-grub-x86_64.gz"
extra = "(hd1,0)/boot/grub/menu.lst"
disk = ['phy:/dev/denmark/horatio,xvda,w','phy:/dev/denmark/rescue,xvde,r']
```

Then, in */boot/grub/menu.lst* on the rescue disk:

```
default=0
timeout=5

title Xen domain boot
        root (hd1)
        kernel /boot/pv-grub-x86_64.gz (hd0,0)/boot/grub/menu.lst

title CentOS-rescue (2.6.18-53.1.14.el5xen)
        root (hd1)
        kernel /boot/vmlinuz-2.6.18-53.1.14.el5xen ro root=LABEL=RESCUE
        initrd /boot/initrd-2.6.18-53.1.14.el5xen.img

title CentOS installer
        root (hd1)
        kernel /boot/centos-5.1-installer-vmlinuz
        initrd /boot/centos-5.1-installer-initrd.img

title NetBSD installer
        root (hd1)
        kernel  /boot/netbsd-INSTALL_XEN3_DOMU.gz
```

(continued)

The first entry is the normal boot, with 64-bit PV-GRUB. The rest are various types of rescue and install boots. Note that we specify (hd1) for the rescue entries; in this case, the second disk is the rescue disk.

The normal boot loads PV-GRUB and the user's */boot/grub/menu.lst* from (hd0,0). Our default user-editable *menu.lst* looks like this:

```
default=0
timeout=5

title CentOS (2.6.18-92.1.6.el5xen)
        root (hd0,0)
        kernel /boot/vmlinuz-2.6.18-92.1.6.el5xen console=xvc0
root=LABEL=PRGMRDISK1 ro
        initrd /boot/initrd-2.6.18-92.1.6.el5xen.img
```

PV-GRUB only runs on Xen 3.3 and above, and it seems that Red Hat has no plans to backport PV-GRUB to the version of Xen that is used by RHEL 5.x.

Making PyGRUB Work

The domain's filesystem will need to include a */boot* directory with the appropriate files, just like a regular GRUB setup. We usually make a separate block device for */boot*, which we present to the domU as the first disk entry in its config file.

To try PyGRUB, add a `bootloader=` line to the domU config file:

```
bootloader = "/usr/bin/pygrub"
```

Of course, this being Xen, it may not be as simple as that. If you're using Debian, make sure that you have `libgrub`, `e2fslibs-dev`, and `reiserfslibs-dev` installed. (Red Hat Enterprise Linux and related distros use PyGRUB with their default Xen setup, and they include the necessary libraries with the Xen packages.)

Even with these libraries installed, it may fail to work without some manual intervention. Older versions of PyGRUB expect the virtual disk to have a partition table rather than a raw filesystem. If you have trouble, this may be the culprit.

With modern versions of PyGRUB, it is unnecessary to have a partition table on the domU's virtual disk.

Self-Support with PyGRUB

At prgmr.com, we give domU administrators the ability to repair and customize their own systems, which also saves us a lot of effort installing and supporting different distros. To accomplish this, we use PyGRUB and see to it that every customer has a bootable read-only rescue image they can boot into if their OS install goes awry. The domain config file for a customer who doesn't want us to do mirroring looks something like the following.

```
bootloader = "/usr/bin/pygrub"

memory = 512
name = "lsc"
vif = [ 'vifname=lsc,ip=38.99.2.47,mac=aa:00:00:50:20:2f,bridge=xenbr0' ]

disk = [
        'phy:/dev/verona/lsc_boot,sda,w',
        'phy:/dev/verona_left/lsc,sdb,w',
        'phy:/dev/verona_right/lsc,sdc,w',
        'file://var/images/centos_ro_rescue.img,sdd,r'
]
```

Note that we're now exporting four disks to the virtual host: a */boot* partition on virtual sda, reserved for PyGRUB; two disks for user data, sdb and sdc; and a read-only CentOS install as sdd.

A sufficiently technical user, with this setup and console access, needs almost no help from the dom0 administrator. He or she can change the operating system, boot a custom kernel, set up a software RAID, and boot the CentOS install to fix his setup if anything goes wrong.

Setting Up the DomU for PyGRUB

The only other important bit to make this work is a valid */grub/menu.lst*, which looks remarkably like the *menu.lst* in a regular Linux install. Our default looks like this and is stored on the disk exported as sda:

```
default=0
timeout=15

title centos
        root (hd0,0)
        kernel /boot/vmlinuz-2.6.18-53.1.6.el5xen console=xvc0 root=/dev/sdb ro
        initrd /boot/initrd-2.6.18-53.1.6.el5xen.XenU.img

title generic kernels
        root (hd0,0)
        kernel /boot/vmlinuz-2.6-xen root=/dev/sdb
        module /boot/initrd-2.6-xen

title rescue-disk
        root (hd0,0)
        kernel /boot/vmlinuz-2.6.18-53.1.6.el5xen console=xvc0 root=LABEL=RESCUE
ro
        initrd /boot/initrd-2.6.18-53.1.6.el5xen.XenU.img
```

NOTE /boot/grub/menu.lst *is frequently symlinked to either* /boot/grub/grub.conf *or* /etc/grub.conf. /boot/grub/menu.lst *is still the file that matters.*

As with native Linux, if you use a separate partition for */boot*, you'll need to either make a symlink at the root of */boot* that points boot back to `.` or make your kernel names relative to `/boot`.

Here, the first and default entry is the CentOS distro kernel. The second entry is a generic Xen kernel, and the third choice is a read-only rescue image. Just like with native Linux, you can also specify devices by label rather than disk number.

WORKING WITH PARTITIONS ON VIRTUAL DISKS

In a standard configuration, partition 1 may be */boot*, with partition 2 as */*. In that case, partition 1 would have the configuration files and kernels in the same format as for normal GRUB.

It's straightforward to create these partitions on an LVM device using `fdisk`. Doing so for a file is a bit harder. First, attach the file to a loop, using `losetup`:

```
# losetup /dev/loop1 claudius.img
```

Then create two partitions in the usual way, using your favorite partition editor:

```
# fdisk /dev/loop1
```

Then, whether you're using an LVM device or loop file, use `kpartx` to create device nodes from the partition table in that device:

```
# kpartx -av /dev/loop1
```

Device nodes will be created under */dev/mapper* in the format *devnamep#*. Make a filesystem of your preferred type on the new partitions:

```
# mke2fs /dev/mapper/loop1p1
# mke2fs -j /dev/mapper/loop1p2

# mount /dev/mapper/loop1p2 /mnt
# mount /dev/mapper/loop1p1 /mnt/boot
```

Copy your filesystem image into */mnt*, make sure valid GRUB support files are in */mnt/boot* (just like a regular GRUB setup), and you are done.

Wrap-Up

This chapter discussed things that we've learned from our years of relying on Xen. Mostly, that relates to how to partition and allocate resources between independent, uncooperative virtual machines, with a particular slant toward VPS hosting. We've described why you might host VPSs on Xen; specific allocation issues for CPU, disk, memory, and network access; backup methods; and letting customers perform self-service with scripts and PyGRUB.

Note that there's some overlap between this chapter and some of the others. For example, we mention a bit about network configuration, but we go into far more detail on networking in Chapter 5, Networking. We describe `xm save` in the context of backups, but we talk a good deal more about it and how it relates to migration in Chapter 9. Xen hosting's been a lot of fun. It hasn't made us rich, but it's presented a bunch of challenges and given us a chance to do some neat stuff.