The Mock Mainframe Mini–HOWTO

Scot W. Stevenson

<scot@possum.in-berlin.de>

Revision History

Revision 1.2	06. October 2005	Revised by: SWS

Table of Contents

1. Introduction

1.1. Copyright and License

1.2. Disclaimer

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors and inaccuracies that could be damaging to your system. Although this is highly unlikely, the author does not take any responsibility. Proceed with caution.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

1.3. Credits

This document has benefitted greatly from the feedback, commentary, and corrections provided by the following people among others:

Gareth Anderson, Jonathan Clark, Jim Dennis, Adam Hawes, Doug Jensen, Jim McQuillan, Volker Meyer, Binh Nguyen, Douglas K. Stevenson, Paul Sunners

1.4. Feedback

Feedback is most certainly welcome for this document. Please send your additions, comments and criticisms to the following email address: scot@possum.in−berlin.de. Feedback can be in English or German.

1.5. Translations

There are currently no translations.

1.6. Future Versions

At some point in 2004, my wife and I realized that we were spending so much time on the road that it made more sense to move all our stuff to laptops. The home network has been restructured accordingly. In other words, the maintainer of the Mock Mainframe HOWTO isn't using a mock mainframe anymore, which means I am looking for somebody who is to take over this document. Volunteers are welcome to email me at the address given above.

2. Background

2.1. Why This Text?

In the last decade of the past millennium, I moved out of my parents' house and into a small apartment with my girlfriend. I left behind not only the comfort of a magically refilling refrigerator, but also a computer network that suddenly had to survive daily and sometimes creative usage by my mom, dad, and kid sister for months without me. After some gentle persuasion, my girlfriend not only switched from Windows to Linux, but also became my fiancee. I left grad school and got a real job, which left me with even less time to fool around even with my er, *our* network, let alone my parents' computers. My fiancee became my wife, we left the apartment for a small house, and then I found myself spending more time changing diapers than floppies.

In other words, somewhere along the way, I turned into an adult.

It happens to the best of us, I'm told, and there are benefits that go beyond a *de facto* unlimited budget for ice cream. Having all the time in the world to keep computers running, however, is not one of them. I needed some sort of setup for the systems I am responsible for that is

Easy to administer. I don't have the time to do the same thing on three different machines, or figure out which machine needs which patch. Ideally, I only have to take care of one single computer in each network, and that infrequently. Some of the computers should not require *any* maintenance at all for months at a time.

Easy to afford. My hardware budget now competes with house payments, food bills, and the cost of clothes that my daughter seems to grow out of faster than we can buy them. Getting more done with less is not just an intellectual challenge, but a pressing necessity.

Easy to secure. The network's very structure should make it harder for outsiders to do evil things, and, more important, make it easy for me to create a safe "lock–down" state where threats are minimal until I find the time to patch holes.

After a few years of trial and error and a lot of time spent thinking about setting up computers while rocking screaming babies in the middle of the night, I created a "standard" setup. It is not a terribly clever or ingenious way of doing things, and there are probably thousands of systems out there organized along exactly the same lines. The aim of this text is to present this setup in a coherent form so that other people don't have to invent the wheel all over again when faced with the same problem.

2.2. Reasoning and Overview

Most desktop computers nowadays are insanely overpowered for what they are doing most of the time: Email, surfing, and text processing, while maybe listening to music. Unless you are still using a 486DX at 66 MHz, your processor is probably bored out of its registers even if it is doing all of this at once. Check any program that shows the system load such as `xload`, `top`, or `uptime` and you'll see just how much of your expensive hardware is busy doing nothing.

With all of those resources left over, there is no technical reason why more than one person can't use the computer at the same time. This concept seems strange and downright alien to most home users today, thanks in no small part to Microsoft's philosophy of "a computer on every desktop" and the hardware companies' ad campaigns that imply that you are, among other things, sexually inadequate if you don't have your very own super−charged computer under your desk.

There are good commercial reasons for hard− and software companies not to like *multiuser* setups. Even if you have to upgrade the central machine, you are going to need less high−quality hardware than if everybody has their own computer; and if four people could use one Windows machine at the same time, that would be three copies less for Microsoft to make money on. You obviously don't save money if you just install Linux on one machine instead on four, but your hardware costs and administration time will drop.

Of course there are other reasons than big company ad pressure why few people have multiuser setups. One is computer games: Many of them suck up so much hardware that a multiuser−system is usually not the best idea. Also, until a short time ago, there was no easy way to actually have more than one person log on, since most desktop computers come with only one keyboard, one mouse, and one monitor. This has changed: You can now create inexpensive and reliable graphic terminals (also known as *thin clients*) with very little hassle and expense. This allows us to get away with one big machine and a couple of little ones. Last but not least, sharing a machine means you have to behave and get along with other users.

In a nutshell, this text is about *centralizing small computer systems to save time and money*. The mainframe vendor IBM wants us to believe that this is just what the big boys are doing, too. Now that the age of server mania is over, they say, companies are moving stuff back onto those mainframes. Since more and more of those mainframes are running roughly the same Linux you have at home, the only difference between a real mainframe and your computer is a bit of hardware. A few hundred thousand dollars worth of hardware at least, granted, but that doesn't mean that you can't use the same design principle and enjoy the benefits of a "little" mainframe   a "mock" mainframe, if you will.

The basic setup has three parts:

The Mock Mainframe. The one and only core machine. All users access this computer, either by sitting in front of it or (more likely) from a terminal, and they can do so at the same time. In the simplest setup, this machine is home to all users, holds all files, and runs all programs.

The Terminals. What the user actually touches. Cheap, easy to maintain, and expendable, they can be dual−boot machines, Linux Terminals, thin clients, or even X Window server programs for other operating systems.

Support Machines. Optional computers that perform a special task that for reasons of security or performance you'd rather not have on the mock mainframe. The most common support machine is a "guardian" that handles Internet connections.

Parts of this text will deal with installing software that is covered in far greater detail in other Linux HOWTOs. Caught between the extremes of just referring to those texts and copying everything, I have decided to give a very brief description of the installation procedure on a standard system. You'll get a general idea of what needs to be done, but for the details, you'll need the specialized text. This does mean that there are a *lot* of references, but that just goes to show how much I am standing on the shoulders of others here.

(Various nice people have suggested various ways of combining the basic idea with VNC. After reading their explanations and studying the documentation, I have realized that I am too out of my depth here to give sensible advice. This would be the first project for the next caretaker of this document.)

2.3. What You Should Be Aware Of

A mock mainframe setup is not for everybody. It is based on the following assumptions:

A small group of users. Though it should scale well from a family setup to at least a classroom full of people (depending on the hardware and programs used), this is not something you want to run a university or Fortune−500−company with. If you are alone, it doesn't make much sense either. Go find somebody to move in with, then read on.

A sane system load. Unless you can really, really fork out a lot of money for serious hardware (in which case, you should probably not be looking for a *mock* mainframe), this is not a setup where you should have your kids playing *Quake 3* while you are encoding Ogg Vorbis files and your partner is watching a DVD, all at the same time. It is designed primarily for pedestrian workloads like email, browsing, chatting, and text processing.

Some downtime tolerance. We will be using standard, off−the−shelf, home−user−grade hardware. These parts are not built for enterprise strength work, and sooner or later, something is going to break or fail. If whatever you are doing urgently requires anything even close to 24/7 uptime, you'll have to go out and buy industrial strength hardware   and remember to get somebody to guarantee that uptime in writing while you are at it.

Some examples of when a mock mainframe might make sense:

- You have a family of email, surfing and chat freaks who all want to be online at the same time but don't use serious resources when they are.
- You have a small, closed teaching system that can't be expensive or take too much time to administer.
- You and your dorm buddies each have those high−powered computers to blow each other away with computer games, but don't want to go through the hassle of installing a serious Linux system on every one to do something as trivial as your actual course work.
- Your organization has absolutely no money and the only hardware you can get is stuff so old, it doesn't even have scrap value anymore, but you still have to give your people computer access.

(If you have found other situations where this setup works, please let me know.)

2.4. How This Text Is Organized

First, we will take a look at the individual parts of the setup   the mock mainframe, the terminals, the support computers. Then we'll discuss ways of putting these elements together. This is also where we will talk about security. We'll also discuss life with more than one user and setups for very weak hardware.

3. The Individual Pieces

3.1. The Mock Mainframe

3.1.1. The Hardware

Examining your needs. If the load that is going to be placed on the mock mainframe is more or less constant and won't change too much over time, you are in the wonderful position of being able to tailor your hardware to your needs. This probably will let you get away with second–hand hardware, which leaves you with more money for, say, a new surround sound system (or more realistically, a new dish washer).

The simple way to find out just what you need is to throw together a machine, just about *any* machine, and then see how it performs under the load it will actually be asked to bear. Then experiment a bit: Will the computer start to swap if you take out half of the RAM, will it speed up if you put in double the amount? See if you can get away with a slower processor or a smaller hard disk. If you can, get feedback from your users.

These trial runs can take time and may seem like a lot of work. The idea here is to fit the mock mainframe's hardware as exactly as possible to the task at hand so you can use the rest of the hardware for other stuff. Also, these trial runs can have surprising results. Most people have little experience in building a system for more than one user, and tend to overestimate the processor strength required while underestimating the amount of memory they need.

For example, for our setup at home in 2003   two people running SuSE 8.2 and KDE 3.1 with a regular load of email clients, multiple browser windows, chatting and music playback   an AMD Duron 1.0 GHz processor turned out to be overkill. We ended up with a secondhand SMP mainboard with two used Intel Pentium II Xeon 450 MHz CPUs (yes, Pentium "two"). Further experiments showed that 512 MByte RAM was slightly too much RAM: 384 MByte is fine, if you can live with the system going into swap once in a blue moon.

Multiple vs. single processors With more and more people working on one computer at the same time, you'll start having moments when a single processor machine seems to stall. Also, if somebody's process goes berserk and starts hogging the CPU, it can freeze the whole system. This is bad.

Decades of hardware marketing have produced computer users who reflexively go out and buy a faster processor when things slow down. But even the fastest CPU can't do more than one thing at once (we're ignoring tricks like hyperthreading here), it is just somewhat better at faking it. To really do two things at the same time, you need more than one processor. Such systems are usually referred to as "SMP"–computers, from *symmetrical nultiprocessing*. You can get them with eight processors or more (Intel Pentium II Xeon, AMD Opteron, Intel Xeon) but in our price range, two CPU (*dual–processor*) systems are the most common.

More than one processor will go a long way towards keeping the system responsive even when there are lots of processes running. What it will *not* do is make a single process run twice as fast as on a system with a single processor of the same speed. One way to visualize this is to imagine you are doing your laundry: Two washing machines will get the whole job done in about half the time, but that does not mean that they now each spin twice as fast; each load still takes just as long as before. Things are actually more complicated, but this is a good rule of thumb.

Although processor speed might be important for gamers on the bleeding edge or people who want to simulate nuclear explosions on their desktop machine, the current clock speeds are simply perverse for normal use.

You can usually get away with far slower processors than the market is trying to force down your throat, especially if you have more than one CPU. This is a good thing because SMP−mainboards are more expensive than normal, single−processor boards, and then you still have to buy that second processor. Keep in mind that more recent (AMD Opteron / Intel Xeon) SMP systems can have expensive needs such as a special power supply and extra large cases.

A multi−processor mainboard is not a must for a mock mainframe. But if you find your system groaning under multiple users, adding processors might give you a better deal than adding MHz.

(At the time of writing, there was also the problem of latency in the Linux kernel. In the 2.4.x series, the kernel is not pre−emptable, so occasionally a one−processor system will stall while something is happening in the bowels of the operating system. The 2.6.x kernels are supposed to be far more responsive, which would be the end of that problem and of this paragraph,too).

Storage: SCSI vs. IDE, RAID. You might want to take a look at using SCSI instead of IDE for hard disks and other drives. One advantage of SCSI is that you can connect more drives to one computer than the four you are usually limited to with IDE. SCSI drives are also better at moving data back and forth amongst themselves without bothering the processor. They are, however, more expensive and can be louder. On smaller systems with few users and low loads, you should be able to use IDE drives with no problem.

If you are going to build a system where it is important you don't loose your data even between those regular backups you perform every night right after you floss your teeth, you might want to consider a RAID (*Redundant Array of Inexpensive Disks*) setup. Very roughly speaking, a RAID setup duplicates the data on more than one hard disk, so that if one drive crashes, the others still have copies.

Sane graphics. Most graphics cards cater to the game freak who has an unlimited hunger for speed, speed, and more speed and the pocket depth to match. An AGP graphics card with 128 MByte of RAM and dazzling 3D functions is not necessarily a bad thing in a mock mainframe, but be sure that you actually need it. A good used PCI card will usually do just fine for email and surfing.

Heat and Lightning. Beyond the normal hardware considerations mentioned here, give some thought to the parts that protect your machine from threats such as power surges or brown outs, or makes sure that everything stays cool, or shields your drive bays from inquisitive little children with popsicle sticks. A good modern mainboard has temperature alarms and all sorts of other features to help you monitor your system's heath.

In summary:

1. *Think RAM before processor speed.* With more than one user, you'll be using more memory and less CPU time than you expect.
2. *Two slower processors can be better than one fast one.* A faster processor can switch between more than one task faster than a slow one, but two processors don't have to switch at all. This means you can use older hardware, which will almost always be less expensive even though you will need more of it.
3. *Consider SCSI and RAID.* SCSI instead of IDE gives you more drives on one machine, and they are able to play amongst themselves without processor supervision. However, SCSI drives are more expensive and make more noise. RAID helps protect your data from hard disk failure. Both are for more ambitious setups.

When buying hardware for a mock mainframe, online auctioneers are your friends. Whereas your local computer store will try to sell you the newest fad, there is no shortage of previous−generation hardware at affordable prices online.

3.1.2. The Software

Some background on X. The X Window System (*X Windows* or just *X* for short) is the graphics layer that most Linux systems use. Almost all current window managers KDE, Gnome, Blackbox sit on top of X, and almost all variants of Unix use X.

X Windows has one important aspect that we make extended use of with the mock mainframe: It is *network transparent*. The software responsible for controlling the input/output devices screen(s), keyboard, and mouse can be on a different computer than the programs you are actually running. With X, it is possible to sit in Beijing, China, with a 486DX and run your programs on a supercomputer in Langley, Virginia.

This has a whole number of advantages. Graphics are hard work for a computer; having them processed on a different machine than the program they belong to takes a big load off of the central computer. They are not so hard, however, that they can't be handled by an older processor. In the distant past of computer technology, there were special machines called *X Terminals* that did nothing but display graphics. Today, a spare computer with an Intel PentiumPro or an AMD K6 with 300 MHz is enough. This lets you have one big, fat machine running the actual programs and a whole host of cheap, small machines doing all the graphics. Which is exactly what we are looking for.

X Windows does have some drawbacks. It gobbles up a lot of bandwidth, so you will want a fast network. Also, some of the terminology is strange. The computer (or rather the software) that controls screen, mouse, and keyboard is called the "X server", because it "serves" the actual program, which in turn is called the "X client". In this text, we'll stick to "host" and "terminals" to avoid confusion.

There are all kinds of good Linux HOWTOs about X Windows, so again we'll just go through the basic steps and let you consult the special texts. I'm assuming that you already have X set up on the mock mainframe; your distribution should handle that part for you.

First, we have to start the program that handles remote X logins. This is `xdm` (*X Display Manager*). Depending on your system and taste, you might want to use the KDE version *kdm* or Gnome version `gdm` instead; both have nicer graphics and more features. Check the *XDMCP Mini−HOWTO* by Thomas Chao for more details. Normally, you'll want `xdm` (or whatever) to start up in the run level that you ususally use for graphics (for example, run level 5 for SuSE 8.2).

Even when `xdm` is running, the mock mainframe should not let you connect from the outside, which is good security. You distribution might let you change this with a simple entry in one of its configuration files (for example, SuSE 8.2 uses `/etc/sysconfig/displaymanager`). If you have to do it the hard way, you will want to change `/etc/X11/xdm/xdm−config` and `opt/kde3/share/config/kdm/kdmrc` if you are using `kdm`.

After all of this is done, you are ready to test the link. Get a computer you know has a functioning X system, boot it in console mode *not* in graphics mode (runlevel 3 instead of 5 on SuSE systems, use `init 3` as root from a shell). Log in and type

```
        /usr/X11/bin/X −terminate −query <host>
```

where "<host>" is the name or IP−address of the mock mainframe. You should get the same X login prompt as if were sitting at the host machine.

Even if you have booted into graphical mode, you can try the following to force the X server to start a second display:

```
        /usr/X11/bin/X :1 −terminate −query <host>
```

This can be done from within a terminal program such as `xterm` on the running display. Note that by default, the first display is :0.

The rest of the text is written under the assumption that you will be using some standard distribution such as SuSE or RedHat or Gentoo for the mock mainframe. However, it should also be little trouble to adapt the Knoppix terminal server package (see http://www.knoppix.net Knoppix) so that it boots right off a ramdisk.

---

4. The Terminals

The machines you use to connect to the mock mainframe should be inexpensive, easy to maintain and, from a security point of view, expendable.

4.1. Dual Boot Machines

Some people those without a time consuming job, a spouse, or children, for example will want to be able to spend lots of time playing hardware intensive computer games. Although more and more games are coming out for Linux, this usually means running a machine that has a closed source operating system such as Microsoft Windows. The solution to this problem is to set up the game computers as *dual boot machines*. The messy details are usually handled automatically by whatever distribution you are using; if not, check out the *Linux Installation Strategies mini−HOWTO* Tobby Banerjee.

The mock mainframe setup lets you keep the size and complexity of the Linux partition on a dual boot machine to a minimum: All it has to do is to get X running and connected. There are various way to do this, I usually just do the following:

1. Go to `/etc/X11/xdm/`. In the file `Xservers`, comment out the line that is either `:0 local /usr/X11R6/bin/X :0 vt07` or something similar by putting a hash mark ("#") at the beginning. This will stop the computer from starting up X locally during boot time.
2. In `etc/inittab`, insert a new line such as (for SuSE 8.2) `xx:5:respawn:/usr/X11R6/bin/X -query <host>` where "<host>" again is the name of the mock mainframe. The "5" is the runlevel that boots with X; "xx" is just a label I picked; you might have to adapt both to your system *(please be careful:* Playing around with `inittab` can cause serious trouble). This will start X with a call to the mock mainframe, and you should get the login window when you are on the dual boot computer.

Dual boot machines are nice if you don't have to switch between operating systems too often. All of the rebooting can quickly become a bore, though, and a dual boot machine cannot be considered truly expendable, given the price of closed source operating systems.

4.2. Linux Terminals

The Linux Terminal Server http://www.ltsp.org (LTSP) lets you use old hardware to put together bare−bones computers without hard disks that run as thin clients. These machines are cheap, quiet, quick to set up, and once they are running, require just about zero maintenance (unless, say, a fan breaks). The LTSP has taken all kinds of awards and is being used in situations far more demanding than a small mock mainframe for your home. For example, Orwell High School in England used LTSP machines and IBM Blade Servers for their complete system (see http://www.cutterproject.co.uk/Casestudies/orwell_high_school_cutter_case_study.php). If you are going to have terminals that are in use constantly, it is hard to see how this would not be the best solution.

Required hardware. More likely than not, somewhere in your cellar or garage (or wherever you keep the stuff your partner lovingly calls "all that crap"), you probably have a hopelessly outdated mainboard and processor that you've been saving because you never know. Well, guess what.

If you are using a 100 Mbit ("Fast") Ethernet network, stay above a 486DX; a Pentium II should be fine. See if you can scrape together about 32 MByte of RAM. You'll need a floppy drive for the initial phase. You'll also need a decent graphics card and a monitor   "decent" doesn't necessarily mean a AGP graphics card with 128 MByte RAM, it means a clear, crisp picture.

The only thing you have to pay slightly more attention to is the network card. Find one that has a socket to plug ROM chips in: a "bootable" network card. You can get away with one that doesn't have the socket, but then you have to keep booting from the floppy. We'll also need the unique number (*Media Access Control* or MAC number) of the network card. On good cards, it is included on a little sticker on the board and looks something like this:

```
        00:50:56:81:00:01
```

If you can't find it on the card, try booting the system with a Linux rescue floppy or any other kernel. The number should be displayed during boot when the card is detected.

Add a keyboard and a case and that's it. Notice we don't have a hard disk, let alone a CD−ROM. With the right kind of fans for the power supply and the processor, you have a very quiet machine.

How they work.

The LTSP home page has an in−depth technical discussion of what happens when the system powers up. In brief, human terms:

When turned on, the Linux Terminal, like any other computer, looks around to see what it has been given in way of hardware. It finds a network card with a MAC and notices that it has a floppy with a boot disk (*or* a boot ROM in the network card.) It starts the boot program. This in effect tells the Linux Terminal:

```
       Got your MAC? Good. Now scream for help as loud as you can.
```

The terminal's call goes through the whole (local) network. On the mock mainframe, a program called `dhcpd` (*Dynamic Host Configuration Protocol Server Daemon*) is listening. It compares the MAC the terminal sent to a list of machines it has been told to take care of, and then sends the terminal an answer that includes an IP address and a location where the terminal can get a kernel. The terminal then configures itself with its new name.

Using some more code from the boot program, the terminal starts a program called `tftp` (*Trivial File Transfer Protocol*, a stripped−down version of the venerable `ftp`. This downloads the kernel from the host machine. The terminal then boots this kernel.

Like every other Linux system, the terminal needs a root filesystem. Instead of getting it from a harddisk, it imports it from the mock mainframe via `NFS` (*Network File System*). If the terminal has very little memory, it can also mount a swap partition this way. The terminal then starts X, connects to the mock mainframe via `xdm`, and throws up the login screen.

This all happens amazingly fast. If you turn off all of the various BIOS boot checks on the terminal and boot off of an EPROM in the network card instead of a floppy, it happens even faster.

Running `dhcpd`, `tftpd`, and `nfsd` on the mock mainframe is a security risk you might not be willing to take. In the chapter on Support Machines, we'll show a way of getting around this.

Setting up the software. On the server (mock mainframe) side, you need to install `nfsd` `tftpd`, and `dhcpd`, which your distribution should include as standard packages.

Leave their configuration files untouched for now. The LTSP configuration and installation programs will do most of the work for you. Some files you should be aware of:

`/etc/dhcpd.conf`
: Provide the IP address of the terminal, the hostname, the IP address of the mock mainframe, the MAC of the terminal, and the default gateway. Check to see if the kernel pathname is correct.

`/opt/ltsp/i386/etc/lts.conf`
: These options control the terminal itself.

`/etc/hosts`
: The names of the Linux Terminals and their IP addresses must be listed here. Further down, while describing the network, we'll introduce a systematic naming convention to make this easier.

`/etc/hosts.allow`
: Though not mentioned in the current LTSP documentation, you probably will have to add the following lines to this file: `rpc.mountd : <terminal> : ALLOW` `rpc.mountd : ALL : DENY` where "<terminal>" is the terminal's IP address. This tells the host to allow the terminal to mount the NFS file system.

Creating a boot floppy for the Linux Terminal is usually trivial. Armed with your type of Ethernet card, go to the website mentioned in the LTSP documentation (currently Marty Connor's ROM−O−Matic Website http://www.rom−o−matic.net/, and follow the instructions for a boot floppy. This should produce a file of a few dozen kilobytes that you can then put on a floppy and boot from. Later, when you are sure that your hardware configuration is not going to change and your setup works, replace the floppy by an EPROM that you plug into your Ethernet card.

If you have a more modern motherboard on your Terminal machine, you might be able to get around all of this by selecting "PXE" (*Pre−eXecution Environment*), "MBA" (*Management Boot Agent*) or "Network" from the boot sequence (CMOS) menu.

Using the terminals. Just how many Linux Terminals can one mock mainframe support? The LTSP documentation gives the following example:

```
    It's not unusual to have 40 workstations [Linux Terminals], all running
    Netscape and StarOffice from a Dual PIII−650 with 1GB of ram. We know
    this works. In fact, the load−average is rarely above 1.0!
```

(This part of the documentation was written in March 2002, hence the reference to Netscape, an ancestor of Mozilla FireFox. StarOffice is a commercial variant of OpenOffice.org.)

Linux Terminals will probably require some user education. People who have only ever used Windows tend to have trouble visualizing a system where the graphics layer is not only independent from the rest of the operating system, but can also be accessed from multiple screens. The best way to explain this is with examples. One trick that people new to X just love is when programs start on one terminal and then appear on a different one. To enable this (*but only in a safe environment!*), sit down at a terminal and type

```
        xhost +<host>
```

where "<host>" is the name of the mock mainframe. Then, move to a different terminal and start a program such as `xeyes` or `xroach`:

```
        xeyes −display <terminal>:0 &
```

The eyes should appear on the first terminal's monitor, providing endless amusement for all. When you are done explaining what happened, remember to retract the privileges again on the first terminal with

```
        xhost -<host>
```

You can also use this example to point out why it is dangerous to use the `xhost` command.

Another question that usually comes up is the speed of Linux Terminals. One nice way to demonstrate this is to run a series of screen savers from the `xlock` suite. For example

```
        xlock -inwindow -mode kumppa
```

or more generally

```
        xlock -inwindow -mode random
```

Though the results will depend on your hardware, this usually takes care of any doubts.

If you are using a desktop such as KDE that allows you to shut down the computer when you log off, make sure that this function is disabled. Otherwise, your users will shut down the mock mainframe when trying to get out of the terminal. Tell them to *just turn off the power* once they have logged out. Older users will feel a sense of nostalgia, and younger users will stare at you as if you have gone mad. Such is progress.

4.3. Real X Terminals

If fortune smiles on you or you are rich, you might find yourself with a real thin client. Installing one is usually not much different than setting up a Linux Terminal, except that you will need the software from the vendor, you will probably have to pay for support, and when something goes wrong, you won't be able to fix it yourself.

The Linux Documentation Project has a number of general and special HOWTOs on how to set up X Terminals, for example the *Connecting X Terminals to Linux Mini−HOWTO* by Salvador J. Peralta or the *NCD−X−Terminal Mini−HOWTO* by Ian Hodge.

4.4. X11 Forwarding with ssh

If you are on a machine that already supports X, you might be able to use the X11 Forwarding function of the Secure Shell (`ssh`) program. This is invoked with

```
ssh -X <HOST>
```

and creates a cryptographically protected tunnel to the host machine. X forwarding has to be configured on both machines in `/etc/ssh/sshd_config` on the host machine, `X11Forwarding` must be set to `yes` and it can be a little clumsy to use every day. However, for quick and dirty work, this is a good alternative.

4.5. X Server Programs

As a final way of connecting to the mock mainframe, there are "X server" programs that run under different operating systems (remember, the X server powers the terminal side of things). These let you log onto Linux machines with an operating system that does not natively run X.

Most X servers for ***Windows*** cost money, in some cases a lot of money. The single Cygwin http://cygwin.com/xfree/, which ports X (and GNU tools) to Windows machines.

If you have an ***Apple*** computer with OS X, you are in better shape. For OS 10.3 "Panther", you need to install the X11 package from the installation disks. Then, with any text editor, create an executable bash shell script such as:

```
#!/bin/bash
/usr/X11R6/bin/X -terminate -query "<HOST>" :1
exit
```

Note the window number is :1, because :0 is used by Aqua. Do not use the X11 Server in `/Applications/Utilities/X11.app/Contents/MacOS/X11`, as this doesn't understand the `−query` command: Apple doesn't seem to want people to run remote Aqua sessions. Then, tell the firewall what you are up to (you do have the firewall on, don't you): In `SystemPreferences −> Sharing −> Firewall` create an new entry "X Window System" for Port 6001 (not: 6000). Then, move the shell script icon to wherever you want to keep it. To start an X session, click on the icon. An "EXEC"−icon called "X" will appear in the Dock. Click on this. Enjoy your connection. To get out again, press `Command−Option−a`. (Note: This has not been tested with Mac OS X 10.4 "Tiger")

You can also check the XDarwin http://www.xdarwin.com/ project. XDarwin is an Apple version of the X Window System that sits on the Darwin operating system   a variant of BSD   that is the core of OS X.

(There is one GPL X Server written in ***Java*** you might try: Weirdx http://www.jcraft.com/weirdx/, though the author points out it is not made for heavy loads.)

In this chapter, we have examined terminals that will give you a GUI (*graphical user interface*). If you are tough enough, you can also hook up a text terminal to your mock mainframe and access the system via a CLI (*command line interface*). This option is covered further down.

5. The Support Machines

In theory, you should need no other computers than the mock mainframe and whatever you use as terminals. In practice, you'll probably want additional machines for specific tasks. Usually this will be because of security, not performance.

For example, let's assume you have a network with a dial−up connection to the Internet for email and browsing. Of course you could put all the hard− and software required on the mock mainframe and not see much of a performance hit (in fact, if your network is slow, it might even be faster). But that puts your most valuable computer right where everybody who is on the Internet   which increasingly means anybody on the planet   can attack it.

For better security, put a machine between the mock mainframe and the outside world. Make sure this ***Guardian*** machine is not only heavily fortified, but also expendable, so if it is taken over by the forces of evil or compromised in any other way, you won't lose anything valuable. To lock down the network in an emergency, all you have to do now is to physically turn off the power of the guardian machine (assuming this is the only entry point to your local net). This can be very useful if you can't sit down and go though security measures the moment you see a problem, because, say, your boss at the burger grill just does not realize how important that dorm network is and unfeelingly insists you show up on time to flip the meat.

Other functions you might want to isolate on different machines are web− or other servers on your net that people from the Internet can access. You can also have a support machine tend your Linux Terminals (a ***Terminal Mother***) or to burn CDs (a ***Burner***).

6. Putting the Pieces together

So after reading this far, you know what you want, know where to get it, how to set it up, and want to get going. There are few things you should think about, however, before you start editing configuration files and stringing cables.

6.1. Security

There is only a limited amount I can tell you about your security needs: Everybody faces different threats. All I can do here is give some basic background on how to secure a mock mainframe setup. If you are looking for a good general introduction to security, try the book *Secrets & Lies* by Bruce Schneier.

6.1.1. Needs revisited

In most books on securing computer systems, there comes a point where the author tells you to sit down and "formulate a security policy". This sounds like such a bureaucratic nightmare that most people skip the whole chapter. I'm not going to suggest you formulate anything. But the next time you're taking a shower, ask yourself what kind of defenses you need.

What are you trying to protect? Are you worried about somebody hacking into the mock mainframe and stealing your data, the classic Hollywood threat to computers? Or that your hardware could be destroyed by lightning? Or that somebody will sit down in front of a terminal when the user is off to the bathroom and write emails in his name? Or that people will open the computer cases and steal the processors? Another way to look at this is to figure out what parts of the system would be hard or even impossible for you to do without. For example, the digital photos and films of my daughter when she was a baby are simply irreplaceable.

Who or what are the forces of evil? Once you know what you are trying to protect, think about whom you are protecting it against, maybe while you are brushing your teeth. Are you worried about crackers from the Internet, or that the flaky power company you are stuck with will zap your computers with a power surge? Remember those little kids with popsicle sticks?

If your system is connected to the Internet 24/7, you need to worry about worms and crackers. If you are only online for as long as it takes to pick up those three emails from your mother, you risk in this area is drastically reduced. This shows how the "probability" of an attack figures in. How likely is it for somebody to hit your system during those 20 seconds? If an attack is highly improbable, you won't want to go to the effort of protecting yourself against it. Some things you will probably dismiss without even thinking: Just how were you going to defend your system against attacks by rust monsters?

Once you know what you are afraid of and how probable an attack is, you should have a feeling for the *risks* you are facing. There are three ways of handling risk: You can *take it*, *minimize it*, or *insure against it*. The first option is not as negligent as you might think: Given our budget, most of us are simply taking the risk of meteor strikes. The third option usually costs money, which we don't have, so we will ignore it here.

The second option is touches the three major parts of any security process: *prevention*, *detection*, and *response*. Most computer security deals with prevention: Making sure the cases are locked so nobody can steal the CPUs, for example. Detection is usually skimped   when is the last time you looked at one of the files in `/var/log/`?   and usually little thought is given to the response, because people figure none of this is going to happen anyway. Unfortunately, you need all three, always, at least to some extent.

Even if you decide that detection systems like `tripwire` are too much of a hassle to install and you don't have the time to read log files every day, give some thought to how you could tell that your system has been compromised. In some cases, it will be hard to miss, say, when men with badges knock at your door and take you away because your computer has been sending spam related to an improbable sexual act with squirrels to all of South Korea. Other intrusions might be more subtle. Would you know if somebody copied the files from your letter folder?

Think about how you would respond to at least the most likely attacks and failures. What would you do if your hard disk crashed? If you logged in as root and the system told you that your last log in was on Friday except that you were still in London, England on Friday, singing drinking songs as you happily stumbled from one pub to the next. With a normal home system and good backups, you might be able to get away with *reinstall from scratch* as the standard response all problems great or small (but make sure that your backups are not compromised).

By the time you are putting on your socks, you'll have probably found out that your greatest risks are quite different from those the press talks about all of the time. If you have no Microsoft products on your network, you don't have to worry too much about anti−virus software or Active X vulnerabilities. However, Linux does not enjoy any special bonuses when it comes to power surges, flooding, or broken fans.

6.1.2. Security Principles

Back to prevention: When you design your system, keep these security principles in mind:

Building better baskets. Putting all of your files on one computer might seem like putting all of your eggs in one basket, which proverbial grandmothers say is a bad thing to do. In fact, from a security point of view, this can actually be a good strategy: Since it is almost always easier to defend one thing than it is to defend many, one basket my be fine as long as you make sure that is a *very, very good* basket.

Avoiding complexity. A centralized system is usually less complex to set up and to administer: If you have all of your users on one machine, you don't have to worry about network file systems, network logins, network printers, and all other kinds of clever but complicated ways to connect computers. Keeping things simple keeps things safe. This is true as well for the support machines: They should do one job and one job only.

Encapsulation. This is the process of isolating a part of the system so that if it is compromised, the whole of the system doesn't go down with it. The Guardian is an example of encapsulation: The dangerous work of connecting to the Internet is handled by a cheap, expendable machine that gives attackers few tools to work with. Another example is taking those parts of the system that the user can actually touch with his grubby little hands monitor, keyboard, and mouse and putting them on a Linux Terminal. The mock mainframe setup, however, is obviously not that good at encapsulation: The whole idea of doing everything on one machine runs contrary to this concept.

Defense in Depth. Preventative security measures are only ways of buying time until your response kicks in given enough uninterrupted time, the attacker will always win. To increase the time you have to respond, deploy your defenses in depth: After the attacker has trekked through kilometers of dense jungle, he reaches the moat which surrounds a twenty meter high outside wall, which is followed by a mine field and poisoned bamboo spikes. And in the end, the secret plans to your magical chocolate machine will not only be in code, but also written in invisible ink. That's defense in depth.

The guardian is an extention of your defenses; installing a second firewall on the mock mainframe is another one. It might sound trivial, but use different passwords for the mock mainframe and the guardian. If you have other support machines, putting them on a different network also means more room between them and the attacker. If you have data that you have to keep confidential at all costs (wink−wink nudge−nudge), encrypt it, or at least those backup CDs. After a few years of backups, you won't know where they have all ended up.

But keep in mind that even the deepest defenses only buy you more time. As Indiana Jones and Lara Croft will tell you, getting by the preventative measures is the easy part: All you need is a whip or a few well−timed jumps. The problems start when the locals start shouting and the guys with the guns arrive.

Choke Points. If there is only one way to get into the system and you can control that way completely, that system will be easier to secure in time of danger. We turn to the guardian one more time for an example of a choke point: Turn off the machine, and you are safe from Internet villains, *provided it is really the only access point*. The problem with many networks is that somewhere, somebody has a connection to the outside that the system administrator doesn't know about. Think of all the laptops that now come with a modem or, even worse, a wireless LAN card built in. Connect those laptops to your net, and you have an instant back door. Remember your history: Your main gate can be high and strong and crawling with orcs, but miss one single little spider hole, and two hobbits can ruin your whole day.

6.2. Network Hardware

If you are setting up a network from scratch, go with Fast Ethernet. The cables and network cards are not that much more expensive than the older, 10 MBit/sec Ethernet. X Windows is bandwidth−hungry, and needs always grow before they shrink.

One note about running X terminals over a wireless LAN: I have been told in no uncertain words to avoid this. Two problems are mentioned: Variable bandwidth, which can leave your session slowing to a crawl when your neighbor does something major, and dropouts, which can lead to the whole session being shut down with all X programs using the connection terminated and your work gone. There are also the usual caveats about the security of WiFi connections.

6.3. Network Geography

You can make life a little easer for yourself by picking a sane and systematic way to name your computers: Pick a set of addresses for your system based on what each machine does. Internally, use the IPv4 address space of 192.168.. that is reserved for networks without direct contact to the Internet. For example, let's take 192.168.1.*. The mock mainframe could be 192.168.1.1, the support machines 192.168.1.10 to 192.168.1.19, and the terminals 192.168.1.100 to 192.168.1.199. This way, you can immediately see the type of computer based on the IPv4 number, and the less trusted a machine is, the larger the last number will be.

Combine this with a naming system that is easy to use. For example, you can name the mock mainframe *fatcat* and the terminals *kitten00* to *kitten99* (with IPv4 numbers from 192.168.1.100 192.168.1.199). Giving the support machines names based on their function is probably easier than something systematic. In the feline theme, try *claws* for a guardian machine or *mamacat* for a terminal mother.

7. Life With Multiple Users

Having everybody using a common machine gives your users far more opportunity to get in each other's hair. Even though Unix (and therefore Linux) was designed from the bottom up as a multi−user system, there are only so many resources available, and having one user hogging them will annoy everybody. And they will all come to you and say it is your fault.

7.1. Shared Resources

One of your biggest headaches will probably be ***CD−ROMs*** and ***CD−R/Ws***. In theory, they belong on the mock mainframe like everything else, but this creates lots of problems. Your users need to be able to physically get to the machine to put the CDs in, which might not be a good idea from the security point of view. Once the CD is in the drive, you can get various people trying to mount, unmount or eject it, and getting upset if they can't. Reading a CD (for example with `cdparanoia`) can interfere with multimedia programs and cause sound tracks to skip. Writing CDs is even worse because it requires the system to pay attention for a certain uninterrupted amount of time. If you only have one processor on the machine and other users decide to do something intensive while the burn is going on, the write might fail, and somebody is going to be really upset, because he just lost a blank CD.

One thing you can do is to move the CD−R/W onto a dedicated support machine (the ***Burner***) that does nothing else. Such a machine can be set up with Jörg P. M. Haeger's http://joerghaeger.de/webCDwriter webCDwriter. It has a graphical interface written in Java and runs under any operating system with a Java Virtual Machine. This preserves the principle of encapsulation. Make sure there is nothing else on the Burner that you can't afford to use if the system is compromised. There are of course other, more primitive ways: You could export the user's home directory by NFS, which is, however, exactly the sort of thing we are trying to avoid. Or have the user create an image of the CD as an ISO file, and then let him send it to the support machine via `sftp` or `scp`. Then the user can walk over to the machine and burn it by hand.

In a family setting, none of this might be a problem. For a larger configuration, with untrusted users, it could be a big problem. You might end up telling everybody that they can't burn CDs on this system, period.

Other resources are less of a problem. Traditionally, you used a ***quota*** setting to limit the amount of disk space any single user could use. With hard disks becoming less expensive by the month, this is not much of a problem anymore, but depending on your user base, you might consider installing very large quotas just to be safe. Users, however, are easily upset by the very idea of quotas, especially if they hit their limit while most of the harddisk is still free.

7.2. Screen Savers and Other Gimmicks

The original aim of screen savers was to keep whatever was being displayed on the screen from burning itself into the monitor's phosphorous coating while you were off to the water cooler. Soon, however, clever, cute, and intricate screen savers became an end in themselves. Today's screen savers have become so resource−hungry that some actually require you to have certain types of hardware (like OpenGL support) before they will run.

If you have a mock mainframe with X Windows, you can be sure that every single one of your users will have a screen saver setup that will test the system to its limits (just for fun, log into every terminal attached to the mainframe once you have set everything up, and let each one run a different screen saver. Watch the system load while you do this. Try not to whimper too loudly). To make matters worse, some desktops like KDE let the user set the screen saver's priority. The idea is that the user can set a low priority, but in reality, they increase the priority until their jumping OpenGL balls don't jerk anymore.

Users consider playing around with their screen savers one of their basic computer rights, so just blocking everything except the "blank screen" mode can lead to people showing up in your office with pitchforks and torches. One way around this is to put a *wrapper* around the screen saver that makes sure the priority is set low. For example, if your setup uses the `xlock` command as a screen saver, you can move it to `xlock.real` and then create a shell script named `xlock`:

```
        #!/bin/bash
        nice −19 xlock.real "$@"
```

This is a very crude script, but you get the point. This lets your users keep their beloved screen savers but makes sure that the performance hit won't be deadly to the whole system.

7.3. Idle Terminals

Another annoying habit users have is to walk away from their terminals while they are still logged in. KDE and Gnome have a "Lock Screen" button right next to their "Logout" button, but you might have problems getting your users to use it, at least until the first person finds that somebody has had fun with his email account.

One way to deal with this is to have the system shut down abandoned terminals with the idle daemon, which should be included in your distribution. Use this with care: If you force a user off the system when he still has some half−written letter on his screen, he isn't going to like it. The program `xautolock` can be set up to invoke a screen saver or a different program after an X session has been idle for a configured amount of time.

8. Going Hardcore: Non−GUI Systems

As nice as KDE and Gnome are, they use system resources like popcorn. If you are only starting an application, try a desktop that is more lightweight such as `Blackbox`. Though your distribution should set up the basics for you, you will probably have to edit the configuration files (in this case, the `Blackbox` menu file that is specified in `~/.blackbox`) for each user. Also, make sure your users know how to work the environment. At the very least, teach them that `CTRL−ALT−BACKSPACE` kills the X server.

But real men and women don't need a graphical user interface (GUI) at all: They use a command shell such as `bash`. Before X Windows gave us graphics, the Free Software Foundation (FSF) had created the GNU tools that are as rock steady as any piece of software on the planet. They are the heart of every distribution, and without them, there would be no "Linux" system (which is why "GNU/Linux" is the more percise term). If you have no choice but to get by with really weak hardware we're talking anything down to a 386SX here you can dump X Windows altogether and get along just fine. Even if you stick to GUIs, some basic knowledge of the shell can help you get far more out of your system.

8.1. Why the Command Line Is Cool

Think of Linux on the command line as the Willow Rosenberg approach to computers: Whereas GUIs are as spectacular as a punch on the nose by vampire slayer Buffy Summers, even a little knowledge of the shell will let you work nuanced magic of nearly unlimited power with little effort. True fans of the TV series will realize that there is a warning implied here: The power of the shell can become habit forming, if not downright addictive, and you can destroy your whole system with no chance of recovery if you mess things up. Using `bash` takes you as close to the raw energies of your machine as you can get without using a C compiler, and the danger rises accordingly.

It took Willow six years to become a witch powerful enough to end the world, but it should take you a few weeks at most to become familiar with the command line. Here are four paragraphs to help you decide if you want to make the effort:

The power of the command line environment is rooted in its design philosophy: Each tool is designed to do one job and one job only, but to do that job superbly. Also, almost every tool can be connected to every other tool to create processing chains with just a few commands. Since these tools are (almost) all general purpose, you can solve just about any problem with the right combination. With these same commands, you can write little programs (*shell scripts*) for everyday tasks. If you look closely at the programs your distributor includes, you will see that a lot of the are in `bash`. Other script languages such as Python or Perl might be more powerful, but the command line is always included and has far less overhead.

It is learning the individual tools of the CLI that is somewhat daunting. A lot of commands have strange names that don't even pretend to be mnemonic (the pattern scanning tool `awk` is named for its creators Aho, Kernighan, and Weinberger), only make sense in a historical context (the *tape archiving utility* `tar` is now used to distribute compressed files), or look like they are typos (`umount` instead of "unmount", `passwd` instead of "password"). There can be dozens of options for each command, and they can be just as cryptic. Because the system was written by hackers in the true sense of the word who wanted the computer to get the job done and not talk about it, the shell normally will not ask you for confirmation, even if you tell it to delete every single file on your hard disk. This is where the end of the world scenario from *Buffy* comes in.

Once you *have* mastered the basics of the shell, however, you will get stuff done a lot faster, you will understand jokes such as `rm −rf /bin/laden`, and you will develop a spring in your step and a glint in your eye. This is why even people who are young enough to have been born after the invention of the mouse develop a tendency to use X Windows merely as a comfortable way to open a lot of terminal windows (either `xterm` or the less resource−hungry `rxvt`).

The CLI has just about every tool you'll need: `mutt` or `pine` for email (real hard−core basket cases use `mail`) `w3m` or `lynx` for surfing, and of course the legendary editors `vi` (more commonly `vim` these days) or `emacs`. The obvious exception to this rule are programs that let you view pictures. But then you probably aren't interested in that sort of thing anyway, are you.

8.2. Setting Up Text Terminals

Basically, you have the same options for text terminals as you do with X terminals. Everything is just a bit easier.

For example, you don't have to reboot if you are forced to use a different operation system: Any program that lets you log in via `telnet` (on secure, closed networks) or `ssh` (everywhere else) will do. Microsoft Windows includes a `telnet` client that is best described as rudimentary; for serious work, try a free Win32 implementation such as Simon Tathamt's PuTTY http://www.chiark.greenend.org.uk/sgtatham/putty/. Apple users with Mac OS X should have no problems with their clients.

The Linux Terminal Server Project also has a package for text terminals. The hardware can be as basic as it gets: Go find yourself a 386DX (for those of you who don't remember the Soviet Union or the first `Star Trek` series: This is the original Pentium's grandaddy). The mainboard will probably not have a PCI slot, so you'll need an ISA graphics card and an ISA network card. These are so low down the hardware chain you might have problems finding them, because they are being junked, not sold second hand.

There is no reason, though, why your computer has to be advanced enought to understand the TCP/IP protocol and be part of your local network at all. You can connect just about any computer to the serial port(s) of the mock mainframe: For example, there is a Linux HOWTO for older Macs by Robert Kiesling (*The MacTerminal MINI−HOWTO*); in an article in The Linux Gazette http://www.linuxgazette.com/issue70/arndt.html, Matthias Arndt shows how to convert an Atari ST into a terminal; Nicholas Petreley explains in IT World.com http://www.itworld.com/Comp/2384/LWD010511penguin2/ how to use your Palm Pilot. If you can get it connected to the serial port, chances are you can get it running on Linux. There are special cards with multiple serial ports for larger setups. Of course, there is a HOWTO for that as well: *The Serial HOWTO* by David S.Lawyer.

You can also get special text terminals as individual machines. David S. Lawyer has written an extensive Linux HOWTO on the subject (*Text−Terminal−HOWTO*) that explains how they work, how you set them up, and why you would want one.

8.3. Useful Shell Commands

To get you started on the shell, here are a few commands that are especially useful if you are sharing a system. These very basic examples were chosen to be useful to normal users.

Play nice. The `nice` command is one of those things that would make the world a better place if everybody used it more often, but nobody does. It allows you to lower the *scheduling priority* of a process so that less important programs don't get in the way of the important ones.

For example, assume you have a WAV recording of your own voice as you sing a song under the shower, and you want to convert it to the Ogg Vorbis format to distribute to your fans on the Internet, all three of them. A simple command to do this is

```
oggenc −o showersong.ogg showersong.wav
```

Encoding music formats is a CPU intensive process, so performance will drop. Now, if a few minutes more or less don't matter, just start the line off with `nice`:

```
nice oggenc −o shower.ogg shower.wav
```

Now the encoding will be run with a lower priority, but you will still have to wait for it to finish before you can use the shell again. To have the computer execute a command in the background, add an ampersand ("&") to the end of the line:

```
nice oggenc −o shower.ogg shower.wav &
```

The shell will respond by giving you a *job number* and a *process id* (PID), and then will ask you for the next command.

The `nice` command is a good example of the power that was lost when graphical interfaces became the default: There is no simple way to adjust the priority of a process with a mouse−driven interface.

Do it later. Another way to spread the load is to have an intensive process start at a time when the system is not being used much. Depending on who is on the system with you, this could be three o'clock in the morning or any time until two o'clock in the afternoon.

The `at` command lets you set a time to start a program or any other process that can be run from the command line. To have our shower song encoded at eight in the evening when you are out watching meaningful French love films, you enter the command "at" followed by the time you want execution to start, and then hit the `ENTER`. Then you type in the command itself, followed by another `ENTER`, and finally a `CTRL−d` to finish the sequence:

```
me@mycomputer:> at 20:00
warning: commands will be executed using /bin/sh
> nice oggenc −o shower.ogg shower.wav
> <CTRL−d>
job 1 at 2003−09−28 20:00
```

The `at` command accepts just about any time format: Americans get to use their quaint "08:00pm" notation instead of "20:00", and there are a whole set of shortcuts like `midnight`, `noon` or even `teatime`. `at` sends the output of the command to your mailbox.

Do it when you are bored. A close relative of `at` uses system load, not time of day to determine when a command should be run: `batch` saves the execution for a time when the system load has fallen below a certain value (to see what your system load currently is, run `uptime` from a shell or `xload` under X Windows). The documentation gives this value as 0.8. The syntax for `batch` is basically the same as for `at`, except that the time field is optional.

---

9. Odds and Ends

9.1. Mock Mainframe Case Studies

```
Two People Home Setup

Mainframe: Dual Intel Pentium II Xeon 450 MHz 512 KByte cache CPU, 384 MByte
           PC-100 RAM, PCI graphics card.
Terminal:  AMD K6-2 300 MHz, 64 MByte SDRAM.
Guardian:  PentiumPro 200 MHz, 64 MByte RAM.
Other:     AMD Duron 1.0 GHz, 512 MByte DDR RAM (for games).
```

Guardian and Terminals are on two different networks. Regular load: Two people with KDE 3.1 with kmail, konqueror and/or Mozilla Firebird under SuSE 8.2.

9.2. And Finally

This text is dedicated to my uncle Gary W. Marklund, who gave me the Unix bug.