PREPARING A MINI-LINUX SYSTEM

•Most of the initial work on making Linux image small came from trying to make Linux boot from a floppy disk – Boot disk

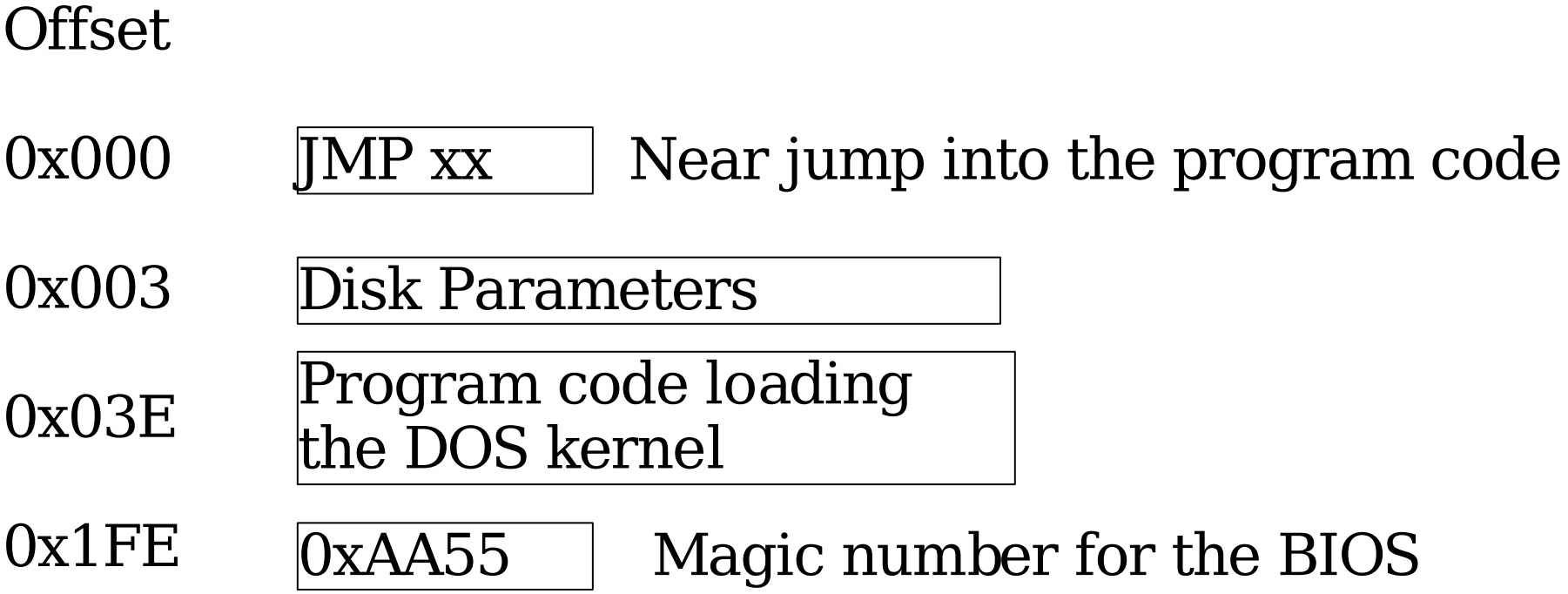
•Boot disk is a miniature self-contained linux system on a floppy – 1.44 MB ?
•It must perform many of the same functions that a complete full-size Linux system performs. Before trying to build one we should understand the basic Linux boot process.

The boot process

All PC systems start the boot process by executing code in ROM (specifically, the BIOS) to load the sector from sector 0, cylinder 0 of the boot drive. The boot drive is usually the first floppy drive (designated A: in DOS and /dev/fd0 in Linux). The BIOS then tries to execute this sector. On most bootable disks, sector 0, cylinder 0 contains either:

* code from a boot loader such as LILO, which locates the kernel, loads it and executes it to start the boot proper; or
* the start of an operating system kernel, such as Linux.

If a Linux kernel has been raw-copied to a diskette, the first sector of the disk will be the first sector of the Linux kernel itself. This first sector will continue the boot process by loading the rest of the kernel from the boot device.



MS-DOS Boot Sector

- Booting from floppy disk : only one boot sector (first one)
- Booting from HD : Slightly difficult, divided into partitions
- First sector of HD : MBR (Master Boot Record)
- End of MBR : partition table stored

- Hard disk can be divided into four partitions : primary partitions
- If they do not suffice : Extended partitions

Offset	Length	
0x000	0x1BE	Code loading and starting the boot sector of the active parition
0x1BE	0x010	Partition 1
0x1CE	0x010	Partition 2
0x1DE	0x010	Partition 3
0x1EE	0x010	Partition 4
0x1FE	0x002	0xAA55

The structure of the master boot record and the extended parition table

1	Boot	Boot flag : 0 = not active, 0x80 active
1	HD	Begin: head number
2	SEC CYL	Begin : sector; cylinder number of Boot Sector
1	SYS	System code : 0x83 Linux, 0x82 Linux swap etc
1	HD	End : head number
2	SEC CYL	End : sector; cylinder number of last sector
4	low byte high byte	Relative sector number of start sector
4	low byte high byte	Number of sectors in the partition

Structure of a parition entry

```
0001a0 ac10 003c f475 00c3 0000 0000 0000 0000
0001b0 0000 0000 0000 0000 ada8 ada8 0000 0100
0001c0 0001 0f0b ffff 003f 0000 d571 002e 0f80
0001d0 ffff 0f0b ffff d5b0 002e 91d0 00bb 0f00
0001e0 ffff 0f83 ffff 6780 00ea 85f0 003e 0f00
0001f0 ffff 0f0f ffff ed70 0128 6590 037f aa55
000200 5652 03be e821 012a bf5e 21f8 8b66 832d
000210 047d 0f00 ca84 8000 ff7c 7400 663e 1d8b
000220 3166 b0c0 397f 0445 037f 458b 2904 0445
```

Device	Boot	Start	End	Blocks	Id	System
/dev/hda1		1	3045	1534648+	b	W95 FAT32
/dev/hda2	*	3046	15240	6146280	b	W95 FAT32
/dev/hda3		15241	19305	2048760	83	Linux
/dev/hda4		19306	77520	29340360	f	W95 Ext'd (LBA)
/dev/hda5		19306	35565	8195008+	83	Linux
/dev/hda6		35566	47760	6146248+	83	Linux
/dev/hda7		47761	57915	5118088+	83	Linux
/dev/hda8		57916	63000	2562808+	83	Linux
/dev/hda9		63001	64035	521608+	82	Linux swap / Solaris
/dev/hda10		64036	77520	6796408+	83	Linux

Originally, there were only primary paritions: The code in the MBR thus only carry out the following operations:

* determine the active parition
* load the boot sector of the active partition, using the BIOS
* jump into the boot sector at offset 0

- Since there is extra space in the MBR, replacement boot managers came into existence.
- They either substitute the MBR with their own code or occupy the boot sector of an active parition.
- LILO, Grub

•When the kernel is completely loaded, it initializes device drivers and its internal data structures.

•Once it is completely initialized, it consults a special location in its image called the ramdisk word. This word tells it how and where to find its root filesystem.

•A root filesystem is simply a filesystem that will be mounted as ``/''. The kernel has to be told where to look for the root filesystem; if it cannot find a loadable image there, it halts.

•In some boot situations (often when booting from a diskette) the root filesystem is loaded into a ramdisk, which is RAM accessed by the system as if it were a disk.

•RAM is several orders of magnitude faster than a floppy disk, so system operation is fast from a ramdisk.

•Also, the kernel can load a compressed filesystem from the floppy and uncompress it onto the ramdisk, allowing many more files to be squeezed onto the diskette.

•Once the root filesystem is loaded and mounted, you see a message like:

VFS: Mounted root (ext2 filesystem) readonly.

Once the system has loaded a root filesystem successfully, it tries to execute the init program (in /bin or /sbin).

init reads its configuration file /etc/inittab, looks for a line designated sysinit, and executes the named script.

The sysinit script is usually something like:

/etc/rc or /etc/init.d/boot.

This script is a set of shell commands that set up basic system services, such as running fsck on hard disks, loading necessary kernel modules, initializing swapping, initializing the network, and mounting disks mentioned in /etc/fstab.

This script often invokes various other scripts to do modular initialization.

For example, in the common SysVinit structure, the directory /etc/rc.d/ contains a complex structure of subdirectories whose files specify how to enable and shut down most system services.

However, on a bootdisk the sysinit script is often very simple.

When the sysinit script finishes control returns to init, which then enters the default runlevel, specified in inittab with the initdefault keyword.

The runlevel line usually specifies a program like getty, which is responsible for handling commununications through the console and ttys.

It is the getty program which prints the familiar ``login:'' prompt.

The getty program in turn invokes the login program to handle login validation and to set up user sessions.

Disk types

Having reviewed the basic boot process, we can now define various kinds of disks involved. We classify disks into four types. ``disk'' refers to floppy diskettes unless otherwise specified, though most of the discussion could apply equally well to hard disks and other storage device.

boot

A disk containing a kernel which can be booted. The disk can be used to boot the kernel, which then may load a root file system on another disk. The kernel on a bootdisk usually must be told where to find its root filesystem.

Often a bootdisk loads a root filesystem from another diskette, but it is possible for a bootdisk to be set up to load a hard disk's root filesystem instead. This is commonly done when testing a new kernel (in fact, ``make zdisk'' will create such a bootdisk automatically from the kernel source code).

root

•A disk with a filesystem containing files required to run a Linux system.

•Such a disk does not necessarily contain either a kernel or a boot loader.

•A root disk can be used to run the system independently of any other disks, once the kernel has been booted.

•Usually the root disk is automatically copied to a ramdisk.

•This makes root disk accesses much faster, and frees up the disk drive for a utility disk.

boot/root

- A disk which contains both the kernel and a root filesystem. In other words, it contains everything necessary to boot and run a Linux system without a hard disk.
- The advantage of this type of disk is that is it compact -- everything required is on a single disk.
- However, the gradually increasing size of everything means that it is increasingly difficult to fit everything on a single diskette, even with compression.

utility

•A disk which contains a filesystem, but is not intended to be mounted as a root file system.

•It is an additional data disk. One would use this type of disk to carry additional utilities where you have too much to fit on your root disk.

•In general, when we talk about ``building a bootdisk'' we mean creating both the boot (kernel) and root (files) portions.

•They may be either together (a single boot/root disk) or separate (boot + root disks).

Building a root filesystem

Creating the root filesystem involves selecting files necessary for the system to run.

A root filesystem must contain everything needed to support a full Linux system.

To be able to do this, the disk must include the minimum requirements for a Linux system:

* The basic file system structure,
* Minimum set of directories: /dev, /proc, /bin, /etc, /lib, /usr, /tmp,
* Basic set of utilities: sh, ls, cp, mv, etc.,
* Minimum set of config files: rc, inittab, fstab, etc.,
* Devices: /dev/hd*, /dev/tty*, /dev/fd0, etc.,
* Runtime library to provide basic functions used by utilities.

Of course, any system only becomes useful when we can run something on it, and a root diskette usually only becomes useful when we can do something like:

* Check a file system on another drive, for example to check your root file system on your hard drive, you need to be able to boot Linux from another drive, as you can with a root diskette system. Then you can run fsck on your original root drive while it is not mounted.

* Restore all or part of your original root drive from backup using archive and compression utilities such as cpio, tar, gzip and ftape.

- We describe how to build a compressed filesystem, so called because it is compressed on disk and, when booted, is uncompressed onto a ramdisk.
- With a compressed filesystem we can fit many files (approximately six megabytes) onto a standard 1440K diskette.
- Because the filesystem is much larger than a diskette, it cannot be built on the diskette.
- We have to build it elsewhere, compress it, then copy it to the diskette.

Creating the filesystem

In order to build such a root filesystem, we need a spare device that is large enough to hold all the files before compression. We will need a device capable of holding about four megabytes. There are several choices:

* Use a ramdisk (DEVICE = /dev/ram0). In this case, memory is used to simulate a disk drive.

* The ramdisk must be large enough to hold a filesystem of the appropriate size. If you use LILO, check your configuration file (/etc/lilo.conf) for a line like RAMDISK = nnn which determines the maximum RAM that can be allocated to a ramdisk. The default is 4096K, which should be sufficient.

* Check to make sure you have a device like /dev/ram0, /dev/ram or /dev/ramdisk. If not, create /dev/ram0 with mknod (major number 1, minor 0).

* If we have an unused hard disk partition that is large enough (several megabytes), this is acceptable.

* Use a loopback device, which allows a disk file to be treated as a device. Using a loopback device you can create a three megabyte file on your hard disk and build the filesystem on it.

* losetup – setting up loopback devices

* We need to have a loop device (/dev/loop0, /dev/loop1, etc.) on our system, or we can create one with

```
mknod /dev/loop0 b 7 0
```

* Once we have installed these special mount and umount binaries, create a temporary file on a hard disk with enough capacity (eg, /tmp/fsfile). We can use a command like:

```
dd if=/dev/zero of=/tmp/fsfile bs=1k count=nnn
```

to create an nnn-block file.

* Use the file name in place of DEVICE below. When we issue a mount command we must include the option -o loop to tell mount to use a loopback device.

After we have chosen one of these options, prepare the DEVICE with:

```
dd if=/dev/zero of=DEVICE bs=1k count=4096
```

This command zeroes out the device.

Important : Why zero?

- Zeroing the device is critical because the filesystem will be compressed later, so all unused portions should be filled with zeroes to achieve maximum compression.

- Keep this in mind whenever you move or delete files on the filesystem.

- The filesystem will correctly de-allocate the blocks, but it will not zero them out again.

- If we do a lot of deletions and copying, our compressed filesystem may end up much larger than necessary.

Creating the filesystem

•The Linux kernel recognizes two file system types for root disks to be automatically copied to ramdisk.

•These are minix and ext2, of which ext2 is preferred(Now ext3 also).

•If using ext2, we may find it useful to use the -N option to specify more inodes than the default;

•Alternatively, we can save on inodes by removing lots of unnecessary /dev files.

•mkfs will by default create 360 inodes on a 1.44Mb diskette.

•Using a compressed root filesystem allows a larger filesystem, and hence more inodes by default, but we may still need to either reduce the number of files or increase the number of inodes.

•So the command you use will look like:

```
mkfs -m 0 -N 2000 DEVICE
```

•(If we are using a loopback device, the disk file you're using should be supplied in place of this DEVICE.)

The mkfs command will automatically detect the space available and configure itself accordingly.

The ``-m 0'' parameter prevents it from reserving space for root, and hence provides more usable space on the disk.

Next, mount the device:

```
mount  -t  ext2  DEVICE  /mnt
```

(You must create a mount point /mnt if it does not already exist.)

Populating the filesystem

•Here is a reasonable minimum set of directories for root filesystem:

* /dev -- Device files, required to perform I/O
* /proc -- Directory stub required by the proc filesystem
* /etc -- System configuration files
* /sbin -- Critical system binaries
* /bin -- Essential binaries considered part of the system
* /lib -- Shared libraries to provide run-time support
* /mnt -- A mount point for maintenance on other disks
* /usr -- Additional utilities and applications

•Three of these directories will be empty on the root filesystem, so they only need to be created with mkdir.

•The /proc directory is basically a stub under which the proc filesystem is placed.

•The directories /mnt and /usr are only mount points for use after the boot/root system is running. Hence again, these directories only need to be created.

/dev

A `/dev` directory containing a special file for all devices to be used by the system is mandatory for any Linux system.

The device special files, however, must be created in a special way, using the **mknod** command.

There is a shortcut, though — copy devices files from your existing hard disk `/dev` directory.

The only requirement is that you copy the device special files using `-R` option.

This will copy the directory without attempting to copy the contents of the files. Be sure to use an *upper case R*. For example:

```
cp -dpR /dev/fd[01]* /mnt/dev

cp -dpR /dev/tty[0-6] /mnt/dev
```

assuming that the diskette is mounted at `/mnt`.

The `dp` switches ensure that symbolic links are copied as links, rather than using the target file, and that the original file attributes are preserved, thus preserving ownership information.

You can create them on the diskette using `mknod also`.

However the devices files are created, check that any special devices you need have been placed on the rescue diskette.

For example, `ftape` uses tape devices, so you will need to copy all of these if you intend to access your floppy tape drive from the bootdisk.

Note that one inode is required for each device special file, and inodes can at times be a scarce resource, especially on diskette filesystems.

For example, if you do not have SCSI disks you can safely ignore `/dev/sd*`; if you don't intend to use serial ports you can ignore `/dev/ttyS*`.

If, in building your root filesystem, you get the error

```
No space left on device
```

but a `df` command shows space still available, you have probably run out of inodes. A `df -i` will display inode usage.

Be sure to include the following files from this directory: `console`, `kmem`, `mem`, `null`, `ram0` and `tty1`.

/etc

The /etc directory contains configuration files. What it should contain depends on what programs you intend to run. On most systems, these can be divided into three groups:

1. Required at all times, *e.g.* `rc`, `fstab`, `passwd`.
2. May be required, but no one is too sure.
3. Junk that crept in.

Files which are not essential can usually be identified with the command:

```
ls -ltru
```

This lists files in reverse order of date last accessed, so if any files are not being accessed, they can be omitted from a root diskette.

Many times on root diskettes, the number of config files can be brought down to 15. This reduces work to deal with three sets of files:

The ones we must configure for a boot/root system:

`rc.d/*`-- system startup and run level change scripts

`fstab`-- list of file systems to be mounted

`inittab`-- parameters for the **init** process, the first process started at boot time.

`gettydefs`-- parameters for the **init** process, the first process started at boot time.

The ones we should tidy up for a boot/root system:

`passwd`-- Critical list of users, home directories, etc.

`group`-- user groups.

`shadow`-- passwords of users. You may not have this.

`termcap`-- the terminal capability database.

If security is important, `passwd` and `shadow` should be pruned to avoid copying user passwords off the system, and so that unwanted logins are rejected when you boot from diskette.

Be sure that `passwd` contains at least `root`. If you intend other users to login, be sure their home directories and shells exist.

`termcap`, the terminal database, is typically several hundred kilobytes.

The version on your boot/root diskette should be pruned down to contain only the terminal(s) you use, which is usually just the `linux` or `linux-console` entry.

The rest. They work at the moment, so we can leave them alone.

Out of this, we only really have to configure two files, and what they should contain is surprisingly small.

`rc` should contain:

```
#!/bin/sh
/bin/mount -av
/bin/hostname Kangaroo
```

Be sure it is executable, be sure it has a "#!" line at the top, and be sure any absolute filenames are correct. You don't really need to run **hostname** — it just looks nicer if you do.

`fstab` should contain at least:

```
/dev/ram0       /               ext2    defaults
/dev/fd0        /               ext2    defaults
/proc           /proc           proc    defaults
```

One can copy entries from existing `fstab`, but we should not automatically mount any of your hard disk partitions; use the `noauto` keyword with them.

Our `inittab` should be changed so that its `sysinit` line runs `rc` or whatever basic boot script will be used. Also, if we want to ensure that users on serial ports cannot login, comment out all the entries for `getty` which include a `ttys` or `ttyS` device at the end of the line. Leave in the `tty` ports so that you can login at the console.

A minimal `inittab` file looks like this:

```
id:2:initdefault:

si::sysinit:/etc/rc

1:2345:respawn:/sbin/getty 9600 tty1

2:23:respawn:/sbin/getty 9600 tty2
```

The `inittab` file defines what the system will run in various states including startup, move to multi-user mode, etc. Check carefully the filenames mentioned in `inittab`; if `init` cannot find the program mentioned the bootdisk will hang, and we may not even get an error message.

Note that some programs cannot be moved elsewhere because other programs have hardcoded their locations.

For example, on some system, `/etc/shutdown` has hardcoded in it `/etc/reboot`.

If we move `reboot` to `/bin/reboot`, and then issue a `shutdown` command, it will fail because it cannot find the `reboot` file.

/bin and /sbin

The `/bin` directory is a convenient place for extra utilities we need to perform basic operations, utilities such as **ls**, **mv**, **cat** and **dd**.

Be sure to include the following programs: `init`, `getty` or equivalent, `login`, `mount`, some shell capable of running rc scripts, a link from `sh` to the shell.

/lib

In `/lib` you place necessary shared libraries and loaders. If the necessary libraries are not found in your `/lib` directory then the system will be unable to boot.

```
% ls -l /lib/libc*
-rwxr-xr-x   1 root     root      4016683 Apr 16 18:48 libc-
2.1.1.so*
lrwxrwxrwx   1 root     root           13 Apr 10 12:25 libc.so.6
-> libc-2.1.1.so*
```

In this case, we want `libc-2.1.1.so`. To find other libraries we should go through all the binaries we plan to include and check their dependencies with **ldd**. For example:

```
% ldd /sbin/mke2fs
libext2fs.so.2 => /lib/libext2fs.so.2 (0x40014000)
libcom_err.so.2 => /lib/libcom_err.so.2 (0x40026000)
libuuid.so.1 => /lib/libuuid.so.1 (0x40028000)
libc.so.6 => /lib/libc.so.6 (0x4002c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Each file on the right-hand side is required. The file may be a symbolic link.

Some libraries are *quite large* and will not fit easily on our root filesystem. For example, the `libc.so` could be about 4 meg. We will probably need to strip libraries when copying them to root filesystem.

In `/lib` we must also include a loader for the libraries. **ldd** will tell exactly which loader is needed.

If you're unsure which one you need, run the **file** command on the library. For example:

```
% file /lib/libc.so.4.7.2 /lib/libc.so.5.4.33 /lib/libc-2.1.1.so

/lib/libc.so.4.7.2: Linux/i386 demand-paged executable (QMAGIC),
stripped

/lib/libc.so.5.4.33: ELF 32-bit LSB shared object, Intel 80386,
version 1, stripped

/lib/libc-2.1.1.so: ELF 32-bit LSB shared object, Intel 80386, version
1, not stripped
```

The `QMAGIC` indicates that `4.7.2` is for A.OUT libraries, and `ELF` indicates that `5.4.33` and `2.1.1` are for ELF.

Providing for PAM and NSS (skip)

Our system may require dynamically loaded libraries that are not visible to `ldd`. If we don't provide for these, we may have trouble logging in or using our bootdisk.

PAM (Pluggable Authentication Modules)

If our system uses PAM (Pluggable Authentication Modules), we must make some provision for it on our bootdisk.

PAM is a sophisticated modular method for authenticating users and controlling their access to services.

An easy way to determine if our system uses PAM is run `ldd` on our `login` executable; if the output includes `libpam.so`, we need PAM.

If security is of no concern with bootdisks we can effectively disable PAM by creating a simple `/etc/pam.conf` file in our root filesystem that looks like this:

```
OTHER  auth      optional     /lib/security/pam_permit.so
OTHER  account   optional     /lib/security/pam_permit.so
OTHER  password  optional     /lib/security/pam_permit.so
OTHER  session   optional     /lib/security/pam_permit.so
```

We also need to copy the file `/lib/security/pam_permit.so` to root filesystem. This library is only about 8K so it imposes minimal overhead.

This configuration allows anyone complete access to the files and services on our machine.

If we care about security on our bootdisk for some reason, we will have to copy some or all of our hard disk's PAM setup to root filesystem.

Copy any libraries needed in `/lib/security` onto our root filesystem.

We must also include `/lib/libpam.so` to our bootdisk. But we already know this since you ran ldd on `/bin/login`, which showed this dependency.

NSS (Name Service Switch)(skip)

If we are using glibc (aka libc6), we will have to make provisions for name services or we will not be able to login.

The file `/etc/nsswitch.conf` controls database lookups for various servies.

If we don't plan to access services from the network (eg, DNS or NIS lookups), we only need to prepare a simple `nsswitch.conf` file should have files against all the services.

We will also need to include `/lib/libnss_files.so.`*X*, where *X* is 1 for glibc 2.0 and 2 for glibc 2.1. This library will be loaded dynamically to handle the file lookups.

If we plan to access the network from our bootdisk, we may want to create a more elaborate `nsswitch.conf` file. We must include a file `/lib/libnss_`*service*`.so.1` for each *service* we specify.

Sample /etc/nsswitch.conf

```
passwd:     files
shadow:     files
group:      files
hosts:      files
services:   files
networks:   files
protocols:  files
rpc:        files
ethers:     files
netmasks:   files
bootparams: files
automount:  files
aliases:    files
netgroup:   files
publickey:  files
```

This specifies that every service be provided only by local files.

Modules

For a modular kernel, we must consider which modules we may want to load from our bootdisk after booting. e.g. we may want to include modules for PPP support if we want to access the net.

These modules may be placed in `/lib/modules`.

We should also include **insmod**, **rmmod** and **lsmod**. Depending on whether we want to load modules automatically, we may also include **modprobe** and **depmod**.

In order to boot a compressed ext2 filesystem, we must have ramdisk and ext2 support built-in. *They cannot be supplied as modules.*

Some final details

Some system programs, such as **login**, complain if the file `/var/run/utmp` and the directory `/var/log` do not exist.
So:

```
mkdir -p /mnt/var/{log,run}
touch /mnt/var/run/utmp
```

Finally, after we have set up all the libraries we need, run **ldconfig** to remake `/etc/ld.so.cache` on the root filesystem.
The cache tells the loader where to find the libraries.
We can do this with:

```
ldconfig -r /mnt
```

Wrapping it up

When we have finished constructing the root filesystem, unmount it, copy it to a file and compress it:

```
umount /mnt
dd if=DEVICE bs=1k | gzip -v9 > rootfs.gz
```

When this finishes we will have a file `rootfs.gz`. This is our compressed root filesystem. We should check its size to make sure it will fit within the available space; if it doesn't we will have to go back and remove some files.

Root File Size

Lets take a look at a common embedded Linux system, the Linux Router Project(www.linuxrouter.org). The Linux Router Project continues to be a very commonly used embedded Linux system.

Its self-described tagline reads "A networking-centric micro-distribution of Linux" which is "small enough to fit on a single 1.44MB floppy disk, and makes building and maintaining routers, access servers, thin servers, thin clients, network appliances, and typically embedded systems next to trivial."

If we download a copy of one of the Linux Router Project's "idiotimages" (idiot-image_1440KB_FAT_2.9.8_Linux_2.0.gz). Opening up the idiot-image we can see a few very interesting facts:

```
[root@sage /tmp]# mount idiot-image_1440KB_FAT_2.9.8_Linux_2.2 /mnt -o
loop
[root@slag /tmp]# mkdir test
[root@slag /tmp]# cd test
[root@slag test]# tar -xzf /mnt/ROOT.LRP
[root@slag test]# du -ch bin root sbin usr var
460k    bin
8.0k    root
264k    sbin
12k     usr/bin
304k    usr/sbin
36k     usr/lib/ipmasqadm
40k     usr/lib
360k    usr
56k     var/lib/lrpkg
60k     var/lib
4.0k    var/spool/cron/crontabs
8.0k    var/spool/cron
12k     var/spool
76k     var
1.2M    total
[root@slag test]# du -ch lib
24k     lib/POSIXness
1.1M    lib
1.1M    total
```

Reducing root filesystem size

One of the main problems with building bootdisks is getting everything to fit into one (or even two) diskette.

Even when files are compressed this can be very difficult, because Linux system components keep growing. We discuss some common techniques used to make everything fit.

Increase the diskette density

By default, floppy diskettes are formatted at 1440K, but higher density formats are possible. Whether we can boot from higher density disks depends mostly on our BIOS.

`fdformat` will format disks for the following sizes: 1600K, 1680K, 1722K, 1743K, 1760K, 1840K, and 1920K. See the **fdformat** man page and

`/usr/src/linux/Documentation/devices.txt`.

uClibc - a Small C Library for Linux

Erik Andersen <andersen@codepoet.org>

•uClibc (aka µClibc/pronounced yew-see-lib-see) is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc.

•Porting applications from glibc to uClibc typically involves just recompiling the source code.

•uClibc even supports shared libraries and threading. It currently runs on standard Linux and MMU-less (also known as µClinux) systems with support for alpha, ARM, cris, e1, h8300, i386, i960, m68k, microblaze, mips/mipsel, PowerPC, SH, SPARC, and v850 processors.

If you are building an embedded Linux system and you find that glibc is eating up too much space, you should consider using uClibc.

If you are building a huge fileserver with 12 Terabytes of storage, then using glibc may make more sense. Unless, for example, that 12 Terabytes will be Network Attached Storage and you plan to burn Linux into the system's firmware...

uClibc and Glibc are not the same -- there are a number of differences which we should be aware of. (Taken from uClibc Documents)

1) uClibc is smaller than glibc. We attempt to maintain a glibc compatible interface, allowing applications that compile with glibc to easily compile with uClibc. However, we do not include _everything_ that glibc includes, and therefore some applications may not compile.

2) uClibc is much more configurable then glibc. This means that a developer may have compiled uClibc in such a way that significant amounts of functionality have been omitted.

3) uClibc does not even attempt to ensure binary compatibility across releases. When a new version of uClibc is released, you may or may not need to recompile all your binaries.

Compiling programs with uClibc

We will need to have our own uClibc toolchain (i.e. GNU binutils and gcc configured to produce binaries linked with uClibc).

We can build our own native uClibc toolchain using the uClibc toolchain builder from uClibc toolchain builder, or the uClibc buildroot system from uClibc buildroot system.

Simply adjust the Makefile settings to match your target system, and then run 'make'.

Pre-compiled uClibc development system

If we want to start using uClibc right away without needing to compile our own toolchain or anything, we can grab a copy of the uClibc development systems, currently available for i386, powerpc, arm, mips, mipsel, and sh4.

These are pre-built uClibc only development systems (created using buildroot), and provide an easy way to get started.

These are bzip2 compressed ext2 filesystems containing all the development software we need to build your own uClibc applications. With bash, awk, make, gcc, g++, autoconf, automake, ncurses, zlib, openssl, openssh, gdb, strace, busybox, GNU coreutils, GNU tar, GNU grep, etc, these should have pretty much everything we need to get started building our own applications linked against uClibc. We can boot into them, loop mount them.

Pre-compiled uClibc development system

The quickest way to get started using a root_fs image (using the i386 platform as an example) is:

Download root_fs_i386.bz2 from kernel.org

bunzip2 root_fs_i386.bz2

mkdir root_fs

su root

mount -o loop root_fs_i386 root_fs

chroot root_fs /bin/sh

Type "exit" to end the chroot session and return to the host system.

Replace common utilities with BusyBox

Much root filesystem space is consumed by common GNU system utilities such as `cat, chmod, cp, dd, df`, etc.

The *BusyBox* project was designed to provide minimal replacements for these common system utilities.

BusyBox supplies one single monolithic executable file, `/bin/busybox`, about 150K, which implements the functions of these utilities.

We then create symlinks from different utilities to this executable; busybox sees how it was called and invokes the correct code.

BusyBox even includes a basic shell. BusyBox is available in binary packages for many distributions, and source code is available from the BusyBox site.

Busybox is a multi-call binary that combines many common Unix utilities into a single executable. When it is run, Busybox checks if it was executed by running a symlink, and if the symlink name matches an applet compiled into Busybox, it runs that applet.

If it was run as "busybox", then Busybox will read the command line and try to execute any applet passed as the first argument. For example:

```
[root@slag busybox]$ ./busybox date

Wed Feb 28 17:17:52 MST 2001

[root@slag busybox]$ ./busybox echo "hello there"

hello there

[root@slag busybox]$ ln -s ./busybox uname

[root@slag busybox]$ ./uname

Linux
```

Busybox is designed such that the developer compiling it for his embedded system can select exactly which applets he wants to include in the final binary.

Thus, it is possible to strip out support for unneeded applets, resulting in a smaller binary with a carefully selected set of commands.

The customization granularity for Busybox even goes one step further: each applet can contain multiple features that can be turned on or off.

Thus, for example, if we don't want to include command-line completion in the Busybox shell(lash), or we do not need to mount NFS filesystems, we can simply turn these features off, further reducing the size of the final Busybox binary.