A Minimal Linux System from Scratch

These instructions will guide you through building a minimal Linux system from scratch using Qemu. To get your tiny and ultra customised system off the ground, you'll need the following:

- A Linux development machine with a standard set of tools (I am using Fedora 7)
- Qemu
- Some spare time

Contents

Creating a Virtual Disk Image

- First off, you'll want to create a virtual hard disk image using the following command:

```
qemu-img create -f raw test.img SIZEM
```

  Where SIZE is whatever you want, I used 12.
  (Raw should be the default)

- Attach the newly created image to the loop back device so you can play around with it like a normal hard disk with the command:

```
losetup /dev/loop0 test.img
```

- Create a partition to work with using fdisk:

```
fdisk /dev/loop0

        Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
        Building a new DOS disklabel. Changes will remain in memory only,
        until you decide to write them. After that, of course, the previous
        content won't be recoverable.

        Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)

        Command (m for help):
```

  You'll then need to execute the following commands in this order:
  n
  p
  choose 1 as it's the first partition
  First cylinder: Press enter to use the default of 1 i.e. the beginning
  Size: Press enter to use the default and have the partition use all the image space or specify a value e.g. +32M (this won't appear if the image is quite small)
  Note: If you specify anything less than 8MB, fdisk will complain that it can't determine the hard disk geometry by itself and you'll need to set the number of cylinders, heads and sectors (CHS) manually. More information is available on this blog.

- After you have verified everything, you'll want to make the partition bootable using the 'a' command. It's now time to commit all your hard work to disk so use the 'w' command and then sit back and relax as your shiny new virtual image gets created.

Setting up the Partitions Filesystem

To create the filesystem, you'll need to know how many blocks the partition is using i.e. how much space it uses in the disk image. This is especially important if it doesn't use all the available free space (one reason could be because if it did, it wouldn't end on a cylinder boundary and thus be unmountable/unusable). This will be explained further once other topics have been covered.

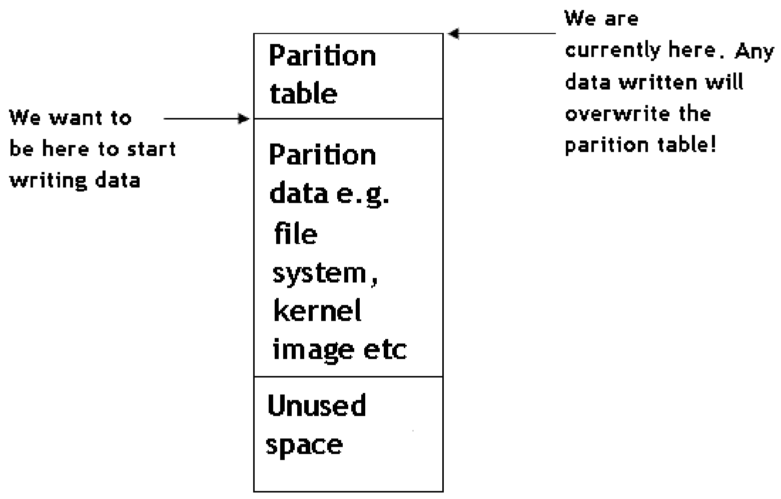
This information can be discovered by executing the following and making a note of the number in the blocks column:

```
fdisk -l -u /dev/loop0

        Disk /dev/loop0: 8 MB, 8388608 bytes
        255 heads, 63 sectors/track, 1 cylinders, total 16384 sectors
        Units = sectors of 1 * 512 = 512 bytes

              Device Boot      Start         End      Blocks   Id  System
        /dev/loop0p1   *          63       16064        8001   83  Linux
```

You now need to format the image but before you do that it must be reattached to the loop back device with an offset. This is to make sure the filesystem doesn't overwrite the partition table as you can see from the picture below.


Parition table
Parition data e.g. file system, kernel image etc
Unused space
We want to be here to start writing data
We are currently here. Any data written will overwrite the parition table!

As you may have noticed, the first partition starts at sector 63. Before that is the partition table and some other stuff which isn't important here. As each sector on a hard disk (which is what the virtual image represents) is 512 bytes, to get the offset in bytes you just calculate 63*512 giving 32256.

To detach, execute the following command:

```
losetup -d /dev/loop0
```

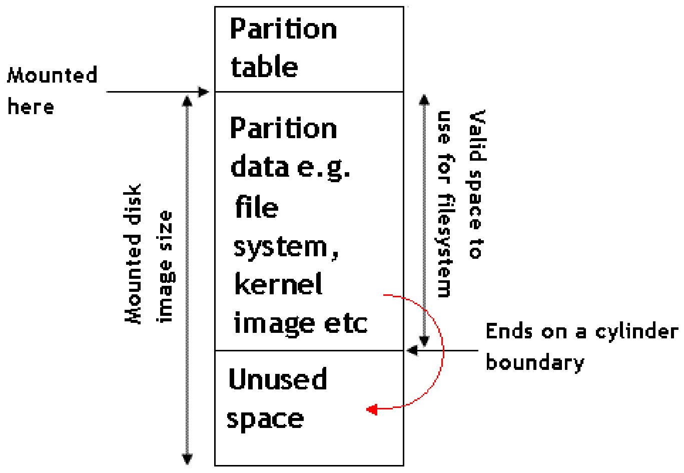
To reattach at the offset, run the following:

```
losetup -o 32256 /dev/loop0 test.img
```

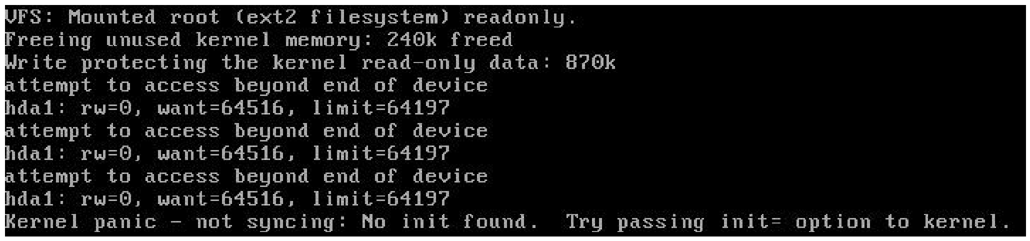
(-o stands for offset)

Why is the Partition Block Size Needed?

As mentioned above, it's very important to specify the block size when creating the filesystem. This may seem strange as you wouldn't normally have to if you were formatting a standard hard disk. The reason (hopefully) becomes somewhat clear when you look at the following:


Parition table
Mounted here
Parition data e.g. file system, kernel image etc
Mounted disk image size
Valid space to use for filesystem
Ends on a cylinder boundary
Unused space

The mounted image's partition table is ignored so as far as any tool like mke2fs is concerned, the partition is as big as the image size minus the partition table size. If the partition had to be smaller than the total available space so that it would end on a cylinder boundary, or if you specified a smaller value for its size then mke2fs is wrong and there will be unused space which must not be used. The red arrow highlights what could happen if it is used and the screenshot below shows the result, a file could be scattered across the image so that one of its blocks is located in the unused/invalid space. When this disk image is booted in Qemu and the block accessed, Linux will detect that it's outside the partitions limit and will throw up nasty errors such as "attempt to access beyond end of device" as shown here:



```
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 240k freed
Write protecting the kernel read-only data: 870k
attempt to access beyond end of device
hda1: rw=0, want=64516, limit=64197
attempt to access beyond end of device
hda1: rw=0, want=64516, limit=64197
attempt to access beyond end of device
hda1: rw=0, want=64516, limit=64197
Kernel panic - not syncing: No init found.  Try passing init= option to kernel.
```

(This can also happen when using GRUB and manifests itself as the dreaded "error 24: Attempt to access block outside partition" message)

The worst part in all this is that you may be lucky and your initial system may have all its files located completely within the valid partition only to fail later when you start to add more data.

Finally you can create the filesystem, here I've chosen to use ext2 which involves executing the following command:

```
mke2fs -b 1024 /dev/loop0 NUM_BLOCKS
```

in this example, NUM_BLOCKS would be 8000.
I've made use of the -b command to specify block size, as the image is quite small it seems to make sense to use the smallest block size to make the most efficient use of the available disk space.

As you probably know, files are made up of at least 1 block and because the block size has been set to 1024, the smallest file that could possibly exist is now 1024 bytes. This is the smallest block size that the filesystem will allow.

The reason for using 8000 rather than 8001 is for efficiency and you should probably make sure that the size is a multiple of 4 unless you've changed the block size and/or page size or aren't using a normal PC. If you want to know how mke2fs deals with this internally then read the next section, otherwise you can skip ahead to detaching the image.

Under the Hood of mke2fs

In case you want to take a look at the code yourself, you can download version 1.40.2 of e2fsprogs here which includes mke2fs.

In this example run through, I am presuming mke2fs has been executed without a block count specified. I'll start by looking at misc/mke2fs.c at line 1238 which calls ext2fs_get_device_size. This gets the devices size in blocks, so for an 8MB image (8388608 bytes) it will first discover that its size in bytes is 8356352 which is the total size minus the partition table size. It then gets the number of blocks by dividing this value by the block size (which was specified with the -b option as 1024) giving 8160.5. Because of integer rounding, this becomes 8160. For those that prefer code, a simplified version of the function looks like the following:

```
errcode_t ext2fs_get_device_size(
                            const char *file, /* = /dev/loop0 */
                            int blocksize,    /* = 1024 because of -b */
                            blk_t *retblocks) /* = 0 at the moment */
{

    int             fd, rc = 0;
    unsigned long long size64;

    /* opens /dev/loop0 and stores the descriptor in fd */
    fd = open( file, O_RDONLY );

    /* Gets the size of the disk image minus partition table size*/
    ioctl( fd, BLKGETSIZE64, &size64 );
    /* size64 is now 8356352 */

    *retblocks = size64 / blocksize;
    /* *retblocks = 8160 */

    close( fd );
    return rc;
}
```

(This code was created using CodeSourceAsHTML)

Once this returns, nothing else relevant happens until line 1276 which looks like the following:

```
if (sys_page_size > EXT2_BLOCK_SIZE(&fs_param))
        /* Currently, fs_param.s_blocks_count = 8160 */
        fs_param.s_blocks_count &= ~((sys_page_size /
                                    EXT2_BLOCK_SIZE(&fs_param))-1);
```

The first test will compare 4096 (this is the default page size used in the kernel on most home computers) represented by sys_page_size with 1024. 1024 comes from following macros from lib/ext2_fs.h:

```
#define EXT2_MIN_BLOCK_LOG_SIZE    10

#define EXT2_MIN_BLOCK_SIZE        (1 << EXT2_MIN_BLOCK_LOG_SIZE) /*This gives you
1024*/

#define EXT2_BLOCK_SIZE(s)        (EXT2_MIN_BLOCK_SIZE << (s)->s_log_block_size)
/*Because its ext2, s_log_block_size is 0 thus EXT2_MIN_BLOCK_SIZE is unaffected and
remains at 1024*/
```

Obviously 4096 is greater than 1024 so on the next line you can see fs_param.s_blocks_count (currently set to 8160) which gets manipulated so that the blocks on the filesystem will fit efficiently into memory. Translating the code to numbers gives you:

```
fs_param.s_blocks_count &= ~((sys_page_size /
                    EXT2_BLOCK_SIZE(&fs_param))-1);

    → 8160 &= ~((4096 / 1024)-1);
    → 8160 &= ~3;
    → 1111111100000 (8160) &
      1111111111100 (~3)
       =  1111111100000 (8160)
```

In this case nothing happens because 8160 is already a multiple of 4.

Breaking "~((4096 / 1024)-1);" down, you can ignore the -1 and logical not (~) as all that does is help to convert the number (in this case 8160) to a multiple of whatever it needs to be. The important part is the 4096 / 1024 which determines what the multiple is. Now, if you don't know what a page size is you can basically think of it as a value that divides up memory into chunks. In this case the chunks would be 4096 bytes. This calculation ensures that if the entire disk was loaded into RAM, every chunk used would be completely filled making the most efficient use of memory. You can see this by doing the following calculation 8160 * 512 = 4177920 bytes. Loaded into RAM this would take up exactly 4177920/4096 = 1020 chunks

- You can now detach the disk image from the loop back device using the following command:

```
losetup -d /dev/loop0
```

Getting a Boot Loader Installed

Although there are other boot loaders, these days GRUB is almost the standard for UNIX based systems so if you haven't already got it, download version 0.97 from here and configure, make and install it. If you have an old distribution it might be useful to get an updated version and install it to a different directory using the -prefix=/DIRECTORY switch on configure.

- Mount the disk image using the command:

```
mount -o loop,offset=32256 test.img /mnt/SOMETHING
```

- Change to your /mnt/SOMETHING directory and make some of the directories that grub requires:

```
mkdir boot
mkdir boot/grub
```

- Create a placeholder configuration file in the grub directory:

```
Touch boot/grub/grub.conf
```

- Create a symbolic link so that grub can find the config file:

```
ln -s boot/grub/grub.conf boot/grub/menu.lst
```

- Copy the following files to the disk image (if you have a 64bit machine or other architecture then the i386-pc directory will be named something else):

```
cp GRUB_INSTALLATION/grub/i386-pc/stage1 boot/grub
        cp GRUB_INSTALLATION/grub/i386-pc/stage2 boot/grub
        cp GRUB_INSTALLATION/grub/i386-pc/e2fs_stage1_5 boot/grub
```

(GRUB_INSTALLATION is normally /usr/lib when using the version that comes with a distribution)

e2fs_stage1_5 is copied because an ext2 filesystem is used, if you're using something else then a different module will be required and you'll need to look in GRUB_INSTALLATION PATH/grub/i386-pc directory to see if it's supported. For more information on how this all fits together see "GRUB boot process".

- In order to install GRUB on a virtual disk, you'll need to create a floppy disk image which can be done with the following command:

```
dd if=/dev/zero of=floppy.img count=1440 bs=1024
        losetup /dev/loop0 floppy.img
```

  This creates an output file (of) called floppy.img containing the contents of the input file (if) which in this case is all zeros. The size in bytes of floppy.img is determined by count * block size (bs) which in this case is 1440 * 1024 = 1474560.

- To make the disk bootable, follow the instructions here replacing /dev/fd0 with /dev/loop0
- Ensure the floppy and hard disk images are detached and unmounted then proceed to boot Qemu using the following command:

```
qemu -L . -fda "floppy.img" -hda test.img -boot a
```

  Once booted follow the instructions here

  GRUB is now installed on the disk image so in future you can boot Qemu with the following:

```
qemu -L . -hda test.img
```

Booting the Kernel

GRUB is now installed, all it needs is something to boot i.e. a kernel so grab one from your favourite source (or use the download link below) and then copy it into the hard disk images boot directory (don't forget to remount your image with the -o parameter).

IMPORTANT: Unless you're using an initrd image, you must make sure that your kernel has compiled in support for the filesystem on your root partition (in this case ext2). You must also make sure you have the appropriate IDE drivers compiled in which includes the following options in the kernel configuration menus:

```
→ Device Drivers
             → ATA/ATAPI/MFM/RLL support
                  → Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support
               → Include IDE/ATA-2 DISK support
                       → Generic/default IDE chipset support
```

In this case I've copied a kernel image called vmlinux-2.6.21-1 which you can get from here.

GRUB now needs to be told to boot this image so fire up your favourite text editor and open the grub.conf config file which you created earlier (found in boot/grub) and enter the following:

```
default 0
        timeout 30

        title Linux-2.6.21-1
        root (hd0,0)
        kernel /boot/vmlinux-2.6.21-1 root=/dev/hda1 rw
```

If you want more information on the configuration file syntax, have a look here

Now that the kernel is setup and booting, it's time to remount the image and add the minimum amount of user space applications necessary to get a basic system up and running.

Creating a Linux Filesystem

No minimal Linux system would be complete without busybox. It provides all the standard tools you might need e.g. listing directories, copying files etc all in one file reducing the executable file header overhead.

- The compilation process is very similar to configuring a kernel as you can use:

```
make menuconfig
```

- As in the kernel, this allows you to select only the features you want but if you're happy with a standard configuration, you can go with the default settings using the following:

```
make defconfig
```

- To compile, execute:

```
make
```

- Installing is slightly more complicated as you may not want busybox on your development environment interfering with your start up files etc so I recommend using the following:

  ```
  make CONFIG_PREFIX=/INSTALL_PATH install
  ```

  For more information, see the INSTALL file in the root of the busybox source.

- You can now create the key directories on the root of your virtual disk which will be populated in the next few steps. These include:

Directory	Description
bin	binaries available to all users (in this case busybox and symbolic links)
lib	libraries for userspace applications e.g. the standard C library.
dev	device files i.e. files that represent your console, hard disk etc

- The only real binary that needs to go in the bin directory is busybox because as mentioned earlier, although it only consists of one executable, it contains a huge number of standard Linux tools. To access them requires either executing busybox with the desired command as a parameter e.g. (busybox ls) or by using symbolic links e.g. ls → busybox. As the symbolic links have already been created during the busybox installation process, it makes sense to use that method and also saves a lot of typing.

  At the absolute minimum you need to copy the busybox binary and the sh symbolic link (both from the INSTALL_PATH/bin directory) to your virtual disks bin directory. Sh is the first command started by the kernel after it's booted (see http://lxr.linux.no/source/init/main.c?v=2.6.18#L769 for the code that executes it) and obviously the busybox binary is where the actual executable code is stored. I also copied the ln symbolic link so I can easily create more symbolic links to other commands as needed.
  Note: Don't forget to use the -P switch when copying to ensure symbolic links are copied rather than followed and the binary copied multiple times.

- As for the lib directory, it contains BusyBox's dependencies. For the non programmers among you, a single binary doesn't have to contain all the code needed to execute it and may pull in the contents of other files (known as libraries) into its memory space at run time. This is known as dynamic linking. Alternatively, binaries can also be statically linked so that they are self sufficient but take up more disk space.

  BusyBox can be configured to either be statically or dynamically linked using the build options in the configuration menu before compiling. Static linking requires less work on your part because you don't have to worry about copying across library files but in the current state of things this isn't advised due to a static linking bug in glibc (the standard C library).

  To determine which libraries are required, you will need to use ldd which "prints shared library dependancies" by inspecting the provided executable:

  ```
  ldd /INSTALL_PATH/bin/busybox
  ```

  This should display something like:

  ```
  linux-gate.so.1 =>  (0xb7f5b000)
                  libcrypt.so.1 => /lib/libcrypt.so.1 (0x4c52d000)
                  libm.so.6 => /lib/libm.so.6 (0x4b9a1000)
                  libc.so.6 => /lib/libc.so.6 (0x4b84b000)
                  /lib/ld-linux.so.2 (0x4ae7a000)
  ```

  You can ignore linux-gate.so.1, it's a shared kernel object and you won't find any file directly related to it (see http://www.trilithium.com/johan/2005/08/linux-gate/ for more information). As for the rest, you can either compile your own versions or do it the easy way and copy them from your existing distributions /lib directory. The only point to note when copying is that the filenames given are actually symbolic links to the real libraries so you'll need to take both the real binary and link. The reason for this is so that Linux executables can find these shared libraries on any system regardless of version as the symbolic links abstract away the exact filenames that include version information.

- To discover the actual binaries on the system that these symbolic links point to, execute the following command and copy everything across to the virtual disks lib directory:

  ```
  ls -l /lib/{libcrypt.so.1,libm.so.6,libc.so.6,ld-linux.so.2}
  ```

- The final part to the puzzle is to create a console for the command prompt which can be done by executing the following from your virtual disks dev directory:

  ```
  mknod -m 0600 console c 5 1
  ```

  This creates a character device node called console with read and write access for the owner (-m 0600). The 5 and 1 represent the "major and minor" numbers that are used by the kernel to access the actual device. In this case 5 and 1 are reserved numbers that are always used for the console. For a list of device reservations see Documentation\devices.txt in the kernel source.

- Once everything has been created, compiled and/or copied across, the system is ready for booting. If you've set everything up correctly, you'll be greeted with a command prompt designated by a '#'. The first thing you'll probably want to do is list the contents of directories so create a symbolic link to ls by executing the following command:

  ```
  cd /bin
  ln -s busybox ls
  ```

You can easily make more symbolic links to the rest of the applications busybox has to offer through a similar process, see the busybox documentation for more info.

Now you have the smallest bootable Linux system that could possibly exist (aside from cutting stuff out of the kernel) up and running. It's not secure and can't do much but its certainly lean and will only get as fat as you want it to.

So go forth and create your own distribution and while you're at it, let me know if you found this useful. Enjoy!

Further Reading

http://www.linuxfromscratch.org - setting up a complete Linux system from scratch
http://free-electrons.com/articles/elfs - more technical information on setting up a minimal system
Embedded Linux Primer: A Practical, Real-World Approach - provides a good overview of many aspects for a minimal Linux system
http://axiom.anu.edu.au/~okeefe/p2b/buildMin/buildMin.html - another old but useful minimal Linux setup

Troubleshooting

Q: On boot up I get the message "No init found"
A: Make sure your kernel has compiled in support for IDE and your root filesystem (unless you're using initrd), check that the permissions are set correctly on all executables and libraries or just run chmod -c 755 FILENAME on everything other than the console device file, check for any other errors further up in the kernel log and also make sure that you set up your filesystem properly (see below).

Q: I get error 24 when trying to use grub and/or the kernel gives me the error "attempt to access beyond end of device"
A: Something must have gone wrong when you set up your hard disk image, unfortunately this means you'll have to go through all the steps again. You probably want to pay particular attention to the steps entitled Creating a virtual disk image and Setting up the partitions filesystem.

[Advertisement]

Comments

Commenting disabled due to a network error. Please reload the page.

You do not have permission to add comments.