Booting Linux: The History and the Future

Werner Almesberger
Werner.Almesberger@epfl.ch

June 25, 2000

Abstract

Booting an operating system means to mediate between a usually very basic, and frequently unreliable system environment (e.g. the PC BIOS), the functionality required by the operating system itself, and the sometimes rather sophisticated setups users wish to create.

From the humble beginnings of the floppy boot sector, the Linux boot process has grown rich functionality, with versatile boot loaders (LILO, LOADLIN, GRUB, etc.), several boot image formats, and an increasing variety of operations that can be done even before the system is fully booted, e.g. loading of driver modules before mounting the root file system.

The boot process is also becoming more difficult with time: new peripherals with interesting functionality and sometimes even more interesting problems get widely deployed and need to be supported, users create new and complicated system configurations and still need to be able to boot, and last but not least, new functionality is constantly added to the kernel, and some of it, e.g. new file systems, can also affect the boot process.

All the complications the boot process has to handle are even worse during system installation, because a large number of possible configurations must be considered, but storage space is limited. Frequently a single floppy disk has to suffice for the first steps.

This paper describes the boot process under Linux, the challenges it has to face, and how it evolved to meet them. Besides this historical overview, which also illustrates general design concepts, some more recent additions are discussed in detail.

1 Introduction

The boot process consists of two major phases: (1) loading the Linux kernel into memory and passing control to it, and (2) initializing the normal operation environment. Some of the possible ways to performs these steps are depicted in figure 1.

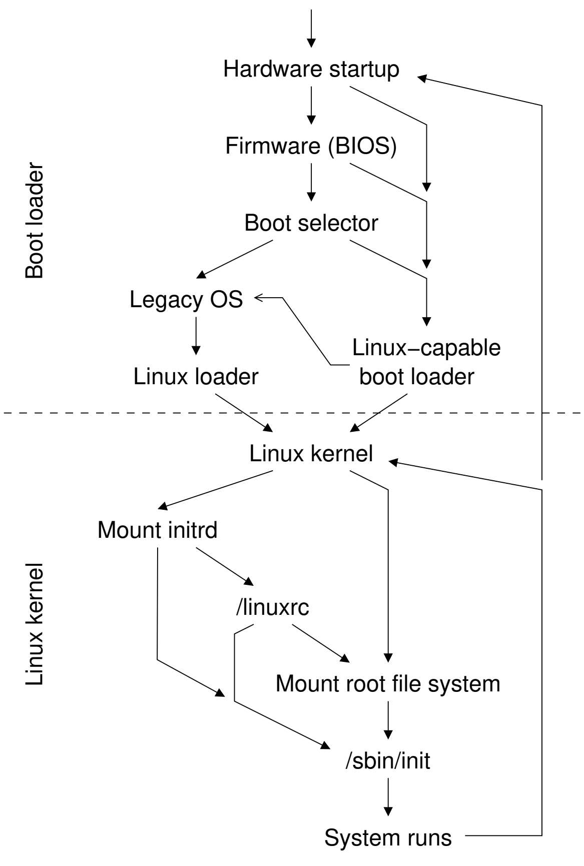

Figure 1: Boot process overview.

While this paper focuses mainly on the i386 architecture, many concepts also apply to other architectures supported by Linux.

1.1 Loading the kernel

The first phase is the domain of boot loaders. They have to retrieve the kernel executable and possibly additional data from some storage media, e.g. a disk, or from an external source, e.g. from a boot server on the network, load them at the right memory location, maybe change the execution mode of the processor, and start the kernel.

Boot loaders typically perform some additional tasks, like providing the kernel with parameters such as information retrieved from the firmware or the boot command line. Some boot loaders can also act as a boot selector and load other operating systems.

The duties of boot loaders and some common designs are discussed in more detail in section 2. An introduction to boot concepts on i386 in general can be found in [1].

1.2 Up and running

Once the kernel is running, it initializes its internal data structures, detects hardware components and activates the corresponding drivers, etc., until it eventually becomes ready to run user-space programs. Before it can start the user-space environment, it needs to provide it with a file system, so it has to mount the root file system first.

In order to mount the root file system, the kernel needs two things: it needs to know the media on which the root file system is located, and it needs drivers to access that media. In the most common configuration, when the root file system is simply an ext2 partition on an IDE disk, this is simple: the number of the root device is passed to the kernel as a parameter, and the IDE driver is typically compiled into the kernel.

1.3 Complications

Things get more complicated if the kernel has no driver for the device. This is quite common for the "generic" kernels that are used when installing a new Linux system, because a kernel with all available drivers would simply be far too big, and some drivers may also upset other hardware when probing for their devices.

This problem is solved by the initrd mechanism, which allows the use of a RAM disk before mounting the actual root file system. This RAM disk is loaded by the boot loader. initrd is described in section 3.

While initrd has proven to be very useful, the design of the mechanism used to mount the root file system after initrd has completed its work was never quite satisfactory. Also, other changes in the kernel made it increasingly difficult to use that mechanism in a "clean" way. Section 4 discusses those issues in more detail.

1.4 The future

Three new challenges await the boot process in the future: (1) the firmware and any hardware the boot loaders have to interface with will grow more functionality — and, if the past is any indication of the future, a richer set of bugs too. (2) file systems containing kernel images will become more complex, e.g. journaling file systems or RAID, and correctly interpreting their content will be very difficult for boot loaders. (3) people will want to load kernels from other exotic sources, e.g. from the network, using a secure connection.

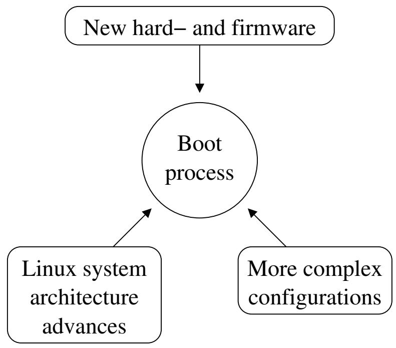

Figure 2: The boot process is facing new challenges from three directions.

While there is little choice but to teach the boot loaders to deal with their immediate firmware and hardware environment, loading the kernel from difficult to access media can be greatly simplified by leaving most of the work to a Linux kernel. Section 5 elaborates further on this topic.

2 Boot loaders

A boot loader performs the following tasks:

- decide what to load, e.g. by prompting the user
- load the kernel and possibly additional data, such as an initrd or parameters for the kernel
- set up an execution environment suitable for the kernel, e.g. put the CPU in privileged mode
- run the kernel

2.1 Taxonomy

Boot loaders come in many sizes and shapes. As shown in figure 3, we will distinguish the following four types of them:

- specialized loaders, e.g. the floppy boot sector LinuxBIOS [2], SYSLINUX [3], Netboot [4]
- general loaders running under another operating system, e.g. LOADLIN [5], ArLo [6]
- file system aware general loaders running on the firmware, e.g. Shoelace, GRUB [7], SILO
- file system unaware general loaders running on the firmware, e.g. LILO [8]

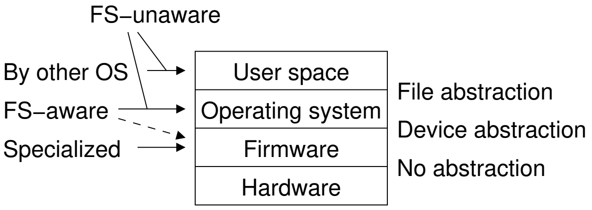

Figure 3: Layers at which boot loaders interact with the underlying services.

Specialized loaders typically know only one storage device, e.g. flash memory or the floppy disk, on which a small number of kernels is stored in some format specific to the boot loader.

Boot loaders that run under another operating system normally use the services provided by the host operating system for reading the kernel image and additional data. This frees them from having to know the structure of the underlying file system or any properties of the actual store devices. One of their disadvantages is that they have to take special precautions when loading the kernel, in order to keep the host operating system operational until they are ready to run the Linux kernel, e.g. they must not overwrite memory locations occupied by the host operating system. Another disadvantage is that the entire boot process takes longer than with other boot loaders, because the host operating system needs some time to boot too.

File system aware boot loaders are almost little operating systems by themselves: they know the structure of one or more file systems, they access devices via the services provided by the firmware, and sometimes, they may even have their own drivers to access hardware directly.

File system unaware boot loaders rely on a third party to map the on-disk data structures to a more general and more convenient representation. E.g. in the case of LILO, the so-called map installer (`/sbin/lilo`) uses the file system drivers already contained in the Linux kernel to perform this mapping, and simply writes the list of data sector locations in its map file. A description of LILO internals can be found in [9].

2.2 File system awareness

The lack of file system awareness is a common complaint about LILO, and competing boot loaders advertize their ability to read file systems without prior mapping as one of their main features. It is therefore interesting to compare the two approaches.

Figure 4 shows what a file system aware boot loader does when using the Second Extended file system: first, the file is written to disk, via the ext2 file system driver. The file system driver adds a bit of meta information. At boot time, the boot loader interprets the ext2 meta information and loads the corresponding data sectors into memory. In order to do so,it has to contain a simplified version of the file system driver.

A file system unaware boot loader (figure 5) requires an additional step after writing the file: the mapping, during which the generalized meta information is written. The boot loader uses this meta informa-

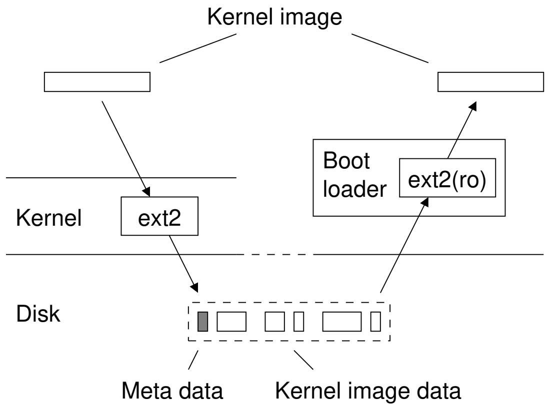

Figure 4: Data flow with file system aware boot loader.

tion to retrieve the actual data. The meta data generated by the file system driver is not needed.

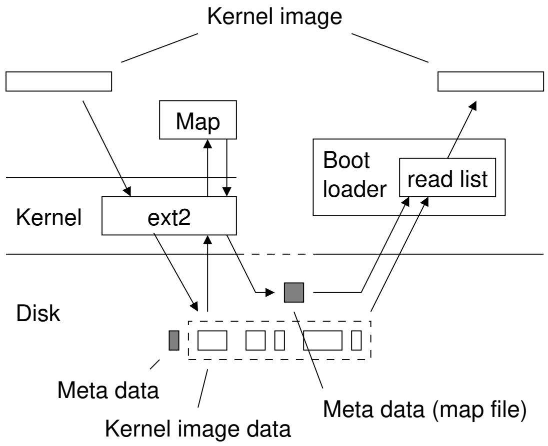

Figure 5: Data flow with file system unaware boot loader.

File system unaware boot loaders have the main disadvantage that the map installer has to be run after adding new kernel images and after an already mapped kernel image changes its on-disk location for some reason.

However, they have one big advantage: if a file system is supported by the Linux kernel and if it fulfills some fairly basic properties, they can load a kernel from it without requiring any change to the boot loader or the map installer. And this is the main reason why LILO was designed to be file system unaware.

2.3 File system history and LILO

In the early days of Linux, the only boot loaders available were the floppy boot sector and Shoelace, a file system aware boot loader inherited from Minix. Shoelace only supported the Minix file system. Since also Linux supported only the Minix file system back then, this was no limitation. However, it became soon clear that the Minix file system, lacking some functionality traditionally found in Unix file systems, e.g. distinct creation, modification, and access time for files, and also restricting file names to 14 characters, was not good enough as the primary file system for Linux.

In order to allow for the implementation of other file systems, the VFS (Virtual File System) interface was added, which quickly led to the creation of a wide variety of new file systems, among them the Extended file system, Xiafs (named after its author), and also a "big" variant of the Minix file system that raised the file name length limit to a whole thirty characters. There was fierce competition among the file systems, and it was quite uncertain which design would eventually prevail, or if there would actually be a single "winner".

In all this confusion, one thing was clear: no matter what file system one favoured, in order to boot from the hard disk, the root file system had to be Minix, because Shoelace did not support anything else. LILO was written to fill this gap. Since implementing and maintaining support for a large number of different file systems (at that time there were already Minix, Extended (ext), and Xiafs in the mainstream kernel, some people had ported BSD FFS, and there was no end in sight) appeared hardly desirable, and the boot loader should not prevent people from experimenting with new file system proposals, a file system unaware design was chosen.

This approach turned out to be very successful. Even today, LILO can boot from most disk file systems supported by the Linux kernel. However, since ext2 has become the de facto standard, and has been so for many years, file system aware boot loader designs have been successfully tried again, and some of them have already gained a certain popularity.

While ext2 was handling everybody's daily work, file system designers have been busy with the next generation of file systems, whose key feature is support for journaling. Considering that there are now

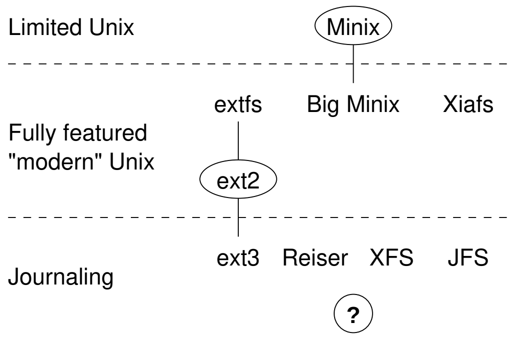

Figure 6: Evolution of the "standard" Linux file system.

(again) several competing proposals (figure 6), it seems likely that the need for the flexibility offered by a file system unaware boot loader will again become strong.

2.4 Other things to load

A Linux boot loader does not only load the kernel image, but it has to give further data to the kernel, e.g. the initial RAM disk, which allows the kernel to set up a fully functional user space without accessing any peripherals. This is discussed in section 3.

Other additional data is a parameter block used during kernel initialization. It typically contains things like the number of the device with the root partition, the desired video mode for the system console, the boot command line, etc. The type of information and its layout are architecture-specific. It is also quite common that the parameter block is merged from multiple sources, e.g. LILO can selectively overwrite the default VGA mode.

2.5 i386 details

One problem that is constantly plaguing the authors of boot loaders, particularly on the i386 platform, are the various disk size limits imposed by hardware or, more frequently, firmware. A good discussion of most known limits can be found in [10]. The usual effect of using a hard disk that exceeds such a limit is that the part of the disk beyond the limit is only accessible under some circumstances.

One such limit that has earned particular fame in the Linux world is the 1024 cylinder limit commonly encountered when using LILO. It originates from the BIOS, which only supports a maximum of 1024 cylinders in the traditional functions for accessing hard disks. This limit is exceeded on all hard disks larger than 8 GB, and sometimes even with smaller ones. Since LILO uses the BIOS for all disk operations, all files accessed by it had to be within the first 1024 cylinders of the hard disk. In 1995, an extension called "Enhanced Disk Drive Specification" [11] raised the limits of the BIOS interface by a factor of roughly 2^{40} to a more reasonable 2^{73} bytes. Unfortunately, it took some more years until one could be reasonably sure that correct implementations of EDD were widely deployed. Support for EDD has been added to a development version of LILO in 1999, and later versions released for general use and maintained by John Coffman also support EDD.

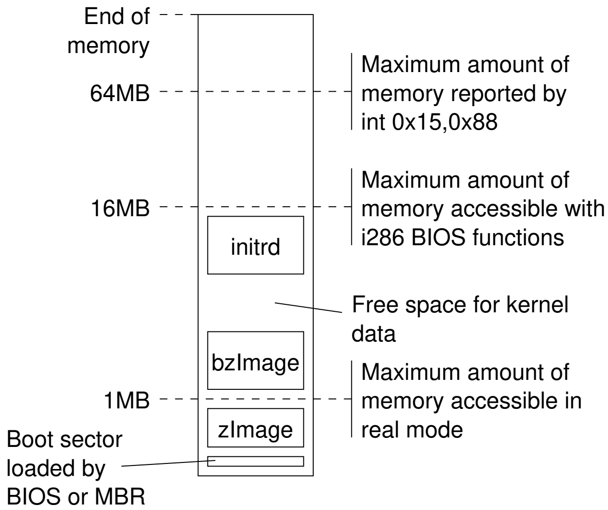

Figure 7: Simplified memory layout at boot time on i386.

Another interesting problem on i386 are the various memory size limits (figure 7). First of all, in the so-called real mode, the CPU has a 4+16 bit address space giving it access to only 1 MB. Since the CPU is in real mode when the boot sector is started, early boot loaders were not able to load kernels (called "Image") larger than several hundred kilobytes.[1]

This was soon found to be too confining, and compressed kernel images were introduced. Compressed kernels (called "zImage") were still limited to 512 kB, but once started, they uncompressed themselves

[1] Some of the lower address space is reserved for the BIOS and video memory, and some space is also claimed by the boot loader. This leaves 512 kB for loading the kernel.

to higher memory locations. This increased the maximum kernel size to approximately 1 MB.

After a few years, also this became a problem, and a mechanism was added to load bigger kernels, called “bzImage”. A bzImage is loaded above the 1 MB barrier, then uncompresses itself, and moves the resulting uncompressed kernel down to 1 MB. The parameter block contained in the floppy boot sector and the real mode setup code are still loaded at their original addresses below 1 MB. This is described in more detail at the end of this section.

Because zImage is inferior to bzImage in almost all respects, support for it is likely to be phased out in the near future.

In order to load the bzImage above 1 MB, the boot loader either switches to a CPU mode giving access to the full address space, or it runs still in real mode but uses special BIOS functions for the copy. Unfortunately, those BIOS functions originate from the i286 era and may still use the so-called protected mode of the i286 with a 8+16 bit address space, giving access only to 16 MB. While 15 MB[2] should be more than sufficient for compressed kernels alone, it also limits the maximum size of initrds, which use the space not occupied by the kernel. Since the 16 MB limit comes from the boot loader but does not exist in the kernel, it is likely to disappear in the future. Some boot loaders are already using copy mechanisms that do not have this restriction.

The next barrier is 64 MB, which is the amount of memory that can be traditionally reported by the BIOS. All newer BIOSes support mechanisms that can report larger memory sizes, and kernels have recently started using them. It is not clear if the 64 MB limit is likely to ever become a serious problem for boot loaders.

The maximum kernel size is also limited by the page tables the kernel sets up prior to its own initialization. For a long time, only 4 MB were mapped. Since kernels started to exceed this limit, it was recently raised to 8 MB.

It should be noted that all these restrictions only apply to the kernel image loaded at boot time. Any additional code loaded by modules can use all of the memory the kernel is willing to provide.

[2]The lower megabyte is reserved for BIOS, boot loader, video memory, etc.

The loading of a bzImage is a fairly intricate procedure, as shown in figure 8. First, the boot loader loads the kernel setup sectors (1) and the compressed kernel (2), and jumps to the setup code (3). The bzImage consists of the compressed kernel code (“text”) and data, and a small piece of uncompressed code for extracting the kernel. Once finished, the setup code jumps to the extractor (4). Then, the kernel is uncompressed into a low memory region below 1 MB (5), and a high memory region after the end of the loaded bzImage (6). By using the low memory region, the extraction process reduces its peak memory usage by 568 kB.

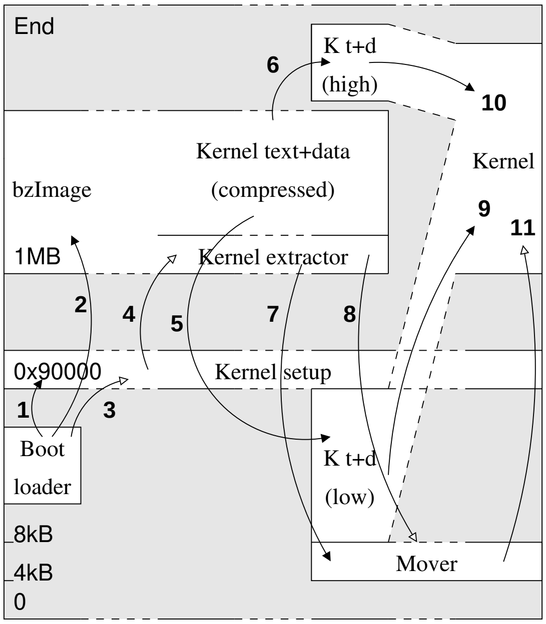

Figure 8: Loading a bzImage.

When the kernel is extracted, it needs to be moved to 1 MB. This is done by a mover function which is copied to a low address (7 and 8). After moving the uncompressed kernel to its destination (9 and 10), the mover jumps to the kernel entry point (11).

2.6 Adding new features

When adding new functionality to the boot process, frequently the question arises where it should be implemented – in the boot loader or in the kernel ? Figure 9 illustrates this choice.

With a large number of different architectures and possibly a large number of boot loaders per archi-

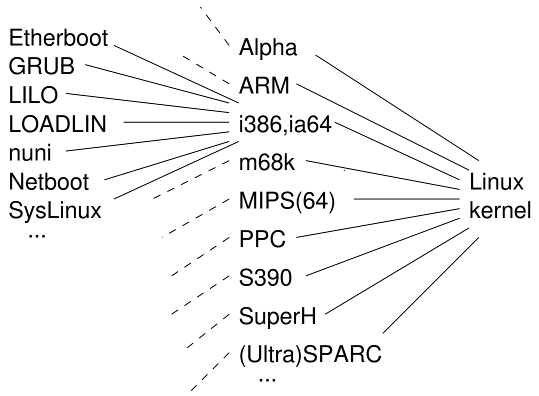

Figure 9: Where to add a new feature ?

tecture, it is clear that additions requiring major changes in boot loaders are not likely to be met with much enthusiasm. With the number of supported architectures increasing, even architecture-dependent changes should not be considered lightly. The introduction of the initial RAM disk was the last time a change affecting all architectures and most boot loaders was made. Fortunately, most authors agreed on the usefulness of initrd, and it is well supported today.

More recent extensions of the boot process try to stay within the kernel, e.g. the mechanisms to boot Linux kernels from Linux combine an architecture-specific part with a more general framework, and recent improvements of mechanisms related to initrd (see section 4) are completely architecture-independent.

Section 3.5 continues this discussion, examining the choice between kernel and user space.

3 Loading drivers

Only loading the kernel is sometimes not enough, because the driver(s) needed to access the root file system may not be included in the kernel. This section describes the reasons for this seemingly paradoxical situation and the solution adopted for it.

3.1 Conflicting drivers

Very early, many Linux distributions encountered the problem that some of the drivers they needed to access any further storage medium, e.g. the CD-ROM, were conflicting with the drivers they needed in other cases.

This can happen quite easily with ISA cards, because the only way to probe for their presence used to be to blindly write to registers at well-known addresses and to check if the card showed whatever reaction was expected in this case. If two cards happened to have some well-known addresses in common and did not respond gracefully to incorrect accesses, e.g. by entering a state that could only be left by following a complicated reset procedure or, in extreme cases, only by a hardware reset, one could not probe for one card without upsetting systems that used the other one.

In order to avoid such conflicts, distributions started to use large numbers of pre-compiled kernels containing only a small number of drivers each. Such a distribution then either had to ship with several floppy disks for all those kernels, or the user had to pick the right kernel from the distribution medium and make their own boot disk before installation. This was hardly a satisfying situation.

The readily available solution to such problems was the use of kernel modules, which can be loaded after either performing a more detailed hardware configuration analysis than done by the kernel, or simply after asking the user for advice.

3.2 Dynamic kernel composition

Loading modules before the kernel mounts the root file system is also desirable after installation, when a customized kernel containing only the components required on the respective system should be used.

Ideally, one would go through regular kernel configuration and compile the kernel from scratch for this, but most users would be rather unpleasantly surprised by the daunting task of having to pick the right set from more than a thousand configuration options, particularly since many mistakes would lead to an unbootable system. Also, there are usually some dependencies among options that are not caught by the kernel configuration system, so certain choices could lead to obscure build failures. Last but not least, building the kernel requires several tools (compiler, etc.), which are not necessarily installed on every system, and the build process may also take a long time on slower machines.

Linking a pre-compiled monolithic kernel would only offer partial relief, because it still requires almost all of the tools needed for compilation, and any conflicts would make the entire linking process fail.

Again, the most reasonable choice is to use modules. The modules framework is regularly used by many people and is therefore quite reliable. If there are conflicts among modules (e.g. missing or duplicate symbols), the respective module and any modules depending on it cannot be loaded, but this is still safer than failing the entire build process.

In principle, a simplified linker could be built on the basis of modules, offering all the advantages of a modular system, while avoiding the slight overhead introduced by modules. For some reason, such a linker was never implemented.

3.3 Chicken and modular eggs

The use of modules requires the presence of a file system.[3] While an installation floppy disk can contain a file system, this does not help for other media, e.g. a CD-ROM or the scenario described in the previous section. Also, every once in a while, floppy disk drives appear that can be accessed via the BIOS, but that are not properly handled by the regular floppy driver.

Fortunately, there is already a program that – by definition – knows how to read data from the boot medium under all circumstances: the boot loader. The logical conclusion was therefore to let the boot loader load the modules too. In order to keep the concept as flexible as possible, and the work of the boot loader simple, it loads a single file that is presented to the kernel as a linear block of memory. The kernel then uses it as a RAM disk. Therefore, the mechanism is called "initial RAM disk" or short "initrd". As a pleasant synergy effect, the RAM disk driver automatically detects if the RAM disk is compressed, and uncompresses it if necessary.

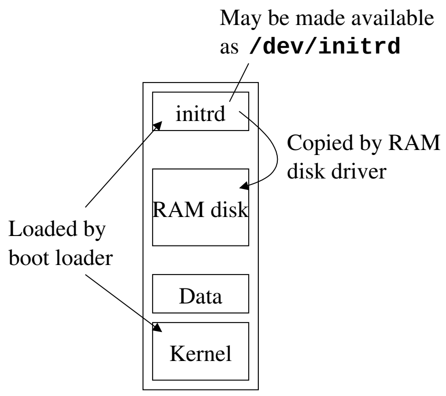

Figure 10: Loading an initial RAM disk.

For debugging or for using the initrd mechanism for other purposes than the initial RAM disk, the boot command line option `noinitrd` can be used to prevent automatic use of the memory block as a RAM disk. Instead, its content is made available via the block device `/dev/initrd`.

3.4 Using the initrd

Once the RAM disk is loaded, any regular Linux programs can be run from it. Initrd can be used in two modes: either for the regular root file system, so the program run is the usual `/sbin/init`, or as an intermediate environment in which the system is prepared for mounting the real file system.

In the latter case, a program called `/linuxrc` is invoked to perform the necessary initialization. When `/linuxrc` finishes, the "real" root file system is mounted and it replaces the initial RAM disk. After this, `/sbin/init` commences with the usual startup procedures. The process of changing the root file system is described in section 4.

3.5 Size matters

The main limitation of an initial RAM disk is that there has to be enough memory for the kernel, the initrd file as loaded by the boot loader, the RAM disk extracted from it by the RAM disk driver, and

[3] An alternative approach that is proposed every once in a while is to teach the boot loader to link modules into the kernel at boot time. The problems of this approach have been discussed in section 2.6.

any other data the kernel needs at that time. This limits the size of compressed initrds to roughly a third of the memory not occupied by the kernel itself.

One obvious improvement is to free memory containing the original initrd data immediately after it has been read when building the RAM disk. This will be implemented in the near future.

By the way, it is a common misconception that the use of initrd automatically implies that many megabytes of precious memory will be wasted. This misconception comes from the fact that most programs are linked with the shared C library (libc), and that some versions of libc are fairly large – typically up to around 4 MB. Even linking with the static version of libc, which yields a program containing only the library functions which are really used, does not result in the desired size reduction. E.g. a program that does nothing at all (`main(){}`) still gets larger than 200 kB.

One reason for this is that libc has many internal dependencies, which require the inclusion of auxiliary components. When some of those dependencies are removed, program sizes become more reasonable, e.g. the example above shrinks to a mere 3 kB. More work is needed in this direction.

Another possibility is simply to refrain from using any library at all. This is feasible for reasonably simple programs. The micro-shell [12] is an example for this.

4 Changing the root file system

Changing the root file system is similar to the task of changing a carpet while still standing on it. Most people would probably suggest to jump up while trying to throw the new carpet under one's feet, and to smooth any wrinkles afterwards. The first implemented solution, called `change_root`, is actually remarkably similar to this approach. It is described in section 4.2.

A much lazier possibility is to roll out the new carpet next to the old one and to just walk over. This much more elegant approach, recently implemented in a mechanism called `pivot_root`, is described in section 4.3. A similar solution, involving layering of the new root file system on top of the old one, is currently being worked on. Its current design is described in section 4.4.

4.1 What's keeping it busy

Changing the root file system is tricky, because the design of Unix makes sure there is always something accessing it. In particular, at least the following items are "busy" if any process is running:

Mapped files	The executable of the process and any shared libraries used by it.
Terminal	Standard input, output, and error of that process. Typically `/dev/console`
Directories	The current directory and the current root directory of the process.

Furthermore, the root file system can also be busy because of:

Mount points	Mounted file systems (e.g. `/proc` or any auxiliary file systems)
Demons	Demon processes or kernel threads.

4.2 Feet in the air

Figure 11 illustrates the approach of awkwardly jumping up while rearranging things underneath one's feet. It works as follows:

- Kernel prepares initrd and starts `/linuxrc`
- `/linuxrc` makes everything ready for mounting the root file system and writes the number of the new root file system device to `/proc/sys/kernel/real-root-dev`
- When `/linuxrc` terminates, the kernel tries to unmount the old root file system and to mount the file system on the device described in `/proc/sys/kernel/real-root-dev` instead
- Kernel runs `/sbin/init`

One of the design goals for change_root was to make its use easy for shell scripts, in order to simplify the transition to initrd.

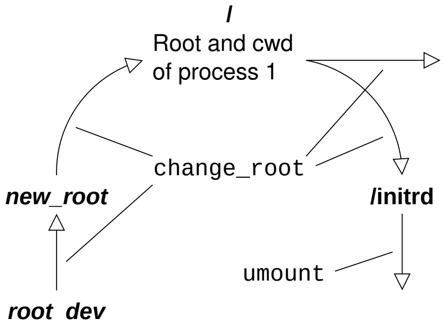

Figure 11: Changing the root file system with change_root.

The following table shows how well this approach handles things keeping the root file system busy:

Mapped files	Disappear at process termination.
Terminal	Closed at process termination.
Directories	Not accessed after process termination.
Mount points	Unaffected.
Demons	Unaffected.

Mount points and demons are still a problem. Mount points can be avoided by simply unmounting everything before /linuxrc terminates. Demon processes can be more difficult to avoid, and kernel threads may refuse to disappear at all.

If change_root fails to unmount the old root file system (because it is kept busy by something), it prints a warning and tries to mount it on a mount point called /initrd on the new root file system instead. Once all accesses to the old root file system have been removed, it can the be unmounted like any other mounted file system. If no directory called /initrd exists, change_root gives up and leaves the old root file system mounted but inaccessible.

4.3 Towards a general solution

While change_root is good enough for most purposes, it has a few undesirable restrictions:

- It can only mount objects which exist as a block device, which precludes NFS,[4] SMB, etc.[5]
- Kernel threads have become quite popular and some of them keep the root file system busy.
- change_root can only be used once, which makes it hard to debug initialization procedures.
- If change_root fails to mount the new root file system, the system hangs.

Besides, all the device number magic and the hard-coded names of change_root are just plain ugly.

Already at the time when change_root was introduced, an alternative design based loosely on the chroot system call was discussed. Recent improvements in VFS have made it comparably easy to implement, so this was finally done.

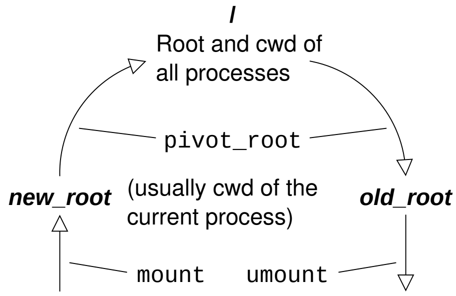

Figure 12: Changing the root file system with pivot_root.

The new mechanism is called pivot_root and figure 12 shows how it works:

- The new root file system is mounted like any other file system.
- A directory is selected as the location for the old (now current) root file system.
- pivot_root is called with the name of the directory containing the new root file system and

[4] change_root was originally able to mount NFS root file systems using the "NFS root" mechanism built into the kernel. Support for this disappeared after a while during a reorganization of the NFS code. Note that the new pivot_root mechanism can be used to cleanly replace and even generalize the NFS root mechanism. It is therefore likely that the latter will be phased out in future kernels.

[5] Recent changes in VFS may allow mounting of such file systems even via their "anonymous" block device. However, this would still be a fairly messy operation.

the name of the directory for the old root file system.

- pivot_root moves the current root file system to the directory for the old file system and makes the new root file system the current root.

The most important differences to change_root are:

- An arbitrary file system can become the new root, including NFS, SMB, etc.
- pivot_root does not attempt to unmount the old root file system, yielding more predictable behaviour than change_root with its two fallback levels.
- pivot_root can be invoked any number of times, which allows cascading of root file system transitions, and makes it easier to debug initialization scripts.
- pivot_root can be retried and is even reversible, which also helps debugging.

Unfortunately, this does not yet help against demons and kernel threads keeping the old root file system busy. The solution chosen is based on the observation that most demons and kernel threads are actually not interested in the file system. They just keep it busy because they, like any other process, reference their current directory and their current root directory.[6] pivot_root therefore scans all processes and changes their current directory and their current root directory if they point to the old root.

This operation is admittedly rather ugly, and the documented behaviour of pivot_root leaves it open to change only root and current directory of the process executing pivot_root. The implications of this are described in the pivot_root man pages included in [13, 14].

Unlike change_root, which makes all changes in a single step after /linuxrc exits, pivot_root allows for a gradual switch to the new root file system. This requires a bit more cooperation from user space for releasing any remaining references to the old root file system. The running executable and shared libraries accessed by it can be closed simply by exec'ing an executable on the new root file system. At the same time, the console can be conveniently closed and re-opened with the device file on the new root file system.[7]

Although all those operations can in principle be done before or after the call to pivot_root, it is usually more convenient to change the root file system first, because this avoids accidental use of items on the old root file systems, e.g. shared libraries.

To summarize, with pivot_root, the situation is now as follows:

Mapped files	Changed by exec.
Terminal	Closed and re-opened.
Directories	Changed with chdir and chroot.
Mount points	Unaffected (except for new root, which is handled directly by pivot_root)
Demons	Current and root directory are forcibly changed.

4.4 Union mounts

The need to forcibly change the current and root directories of processes is the only remaining ugly hack with pivot_root.

Alexander Viro is currently designing so-called "union mounts", an extension of VFS that allows multiple file systems to be stacked at a single mount point. The file systems are accessed only when trying to look up items on that mount point.

To return to the carpet analogy, this gives us a tiny patch of flying carpet that we can use to avoid stepping on the real carpet while replacing it.

Although this work has not yet finished at the time of writing, one can already speculate on how it may allow for a cleaner use of the concepts introduced by pivot_root.

Figure 13 illustrates how this concept may work. The file systems can be either directly mounted and unmounted at the root, or they can be moved from or to other directories.

[6] Kernel threads can release their references to these two directories. Unfortunately, only very few kernel threads make use of this possibility.

[7] When using devfs, a second instance of it should be mounted on the new root file system for this purpose.

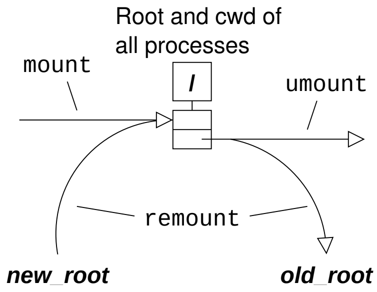

Figure 13: Changing the root file system with union mounts.

So the final situation is as follows:

Mapped files	Changed by `exec`.
Terminal	Closed and re-opened.
Directories	Directories change is transparent.
Mount points	Unaffected (except if moving mount points to root)
Demons	Directories change is transparent.

The mechanism described in this section is likely to be added to the mainstream kernel in the very near future.

5 Linux boots Linux

With the infrastructure discussed so far, we can use any file system the kernel can mount as the root file system. Now wouldn't it be nice if we could also use any file the kernel can read as kernel or initrd ?

File system unaware boot loaders reach their limits when files are no longer stored in sequences of data sectors on the disk, e.g. in the case of software RAID, there may be multiple instances of the same data block, and a RAID5 array in reconstruction mode needs to perform calculations over multiple data blocks in order to obtain the content of a block on a defective volume. Worse yet, the files may not even be on a local disk, but maybe on an NFS or HTTP server.

In principle, any boot loader can of course access any resources the kernel can access too. The only problem is that all the necessary functionality needs to be rebuilt in the boot loader. And once half a dozen file systems, RAID, a TCP/IP stack, NFS, SMB, DHCP, HTTP, etc. are added to a boot loader, it probably looks like a complete operating system …

5.1 The ultimate boot loader

… which brings us right to a very convenient solution: there is already a program that can access everything the kernel can access – it's the kernel itself. And all the other tools that might be needed (e.g. DHCP and such) are conveniently available too.

The only missing element is a means to boot a Linux kernel from within Linux. The concept is basically the same as for boot loaders running under some other host operating system. However, some requirements are slightly higher, because it is desirable to have a solution that can be easily adapted for all platform supported by Linux, and also the range of possible system configurations is wider than for most other such boot loaders, e.g. it seems quite unlikely that LOADLIN is ever used on multiprocessor systems. On the other hand, the work can be simplified by making small changes to the kernel.

Another requirement is to pass on data obtained from the firmware from kernel to kernel. E.g. on i386, video mode, memory layout, SMP configuration, etc. are retrieved either directly from the BIOS or from memory areas initialized by the BIOS. Since these memory areas may be overwritten by the kernel in normal operation, they either need to be protected if booting kernels from Linux is desired, or the information contained in them needs to be extracted and passed on to the next kernel.

Finally, some operations done during initialization, e.g. SCSI or IDE bus scans, may take a significant amount of time. It would be desirable to pass this information from kernel to kernel in order to speed up the boot process.

There are currently at least three different implementations that allow booting a Linux kernel from Linux: bootimg, LOBOS, and Two Kernel Monte. The last two are described in [15] and [16], respectively. Bootimg is described in section 5.3 of this paper.

5.2 What a waste ?

The concept of using a fully featured Unix kernel as a boot loader may look like the perfect waste of resources. In the section, we will consider the implications on time, memory, and disk space.

Note that these calculations may not apply to special environments like embedded systems or small battery-powered devices, which may have very little memory or use a slow CPU. Fortunately, the flexibility offered by the ability of booting a kernel from Linux is hardly necessary in those cases, so an optimized specific solution can be chosen.

First time: loading a kernel and an initrd takes time. Since the kernel is probably compressed, some more time is spent for uncompressing. If we assume that any expensive bus scans are not repeated, and that the hardware is not overly slow or obsolete, we obtain:

1-2 sec	Loading 1-2 MB (kernel and initrd)
1-2 sec	Uncompressing kernel and initrd
1 sec	Other overhead
3-5 sec	

Considering that a normal reboot typically takes 20-60 seconds, this is a reasonably small increase. Also, reboots for configuration changes or kernel updates are much faster now, because the old kernel can directly load the new one, without going through BIOS or boot loader.

The peak memory utilization occurs when the kernel acting as boot loader has loaded the next kernel along with its compressed initrd. Assuming fairly large kernels and initrds, we obtain:

1-2 MB	Boot kernel (running)
2-4 MB	Kernel data
1-2 MB	initrd (mounted)
0.5-2 MB	Compressed kernel
0.5-2 MB	Compressed initrd
5-12 MB	

Since 5 MB is probably the minimum amount of memory required for any usable Linux system, these memory requirements can only become a significant problem if using very large kernels or feature-laden initrds, which are of little use on systems with tight memory constraints.

Finally, the disk space requirements:

1 MB	Compressed boot kernel
1 MB	Compressed initrd
2 MB	

This is hardly noticeable. Developers who frequently change their boot kernel may wish to keep an additional kernel build tree for this purpose. This takes about 100-120 MB.

5.3 Case study: bootimg on i386

This section gives a rough overview of how bootimg [17] currently loads a Linux kernel. Note that this is still work in progress, and major changes are quite probable. Bootimg consists of two parts: a user space program that loads the necessary files and prepares a load map, and kernel code that moves the memory pages to the right locations and starts the new kernel.

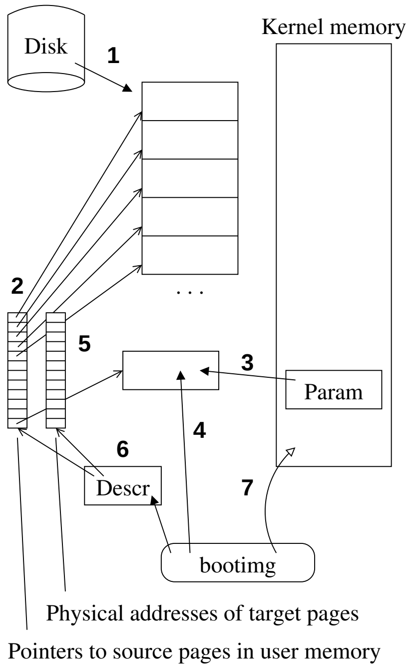

Figure 14: Bootimg: set up from user space.

As shown in figure 14, the user space program first loads the kernel image and, optionally, an initrd file into its address space (1). It registers the addresses of these memory pages in an array of pointers (2). Note that the data does not necessarily have to come from a disk, but it may as well be loaded over the network, or bootimg could even generate it on the fly, e.g. from object modules. Next, bootimg copies the parameter block from the running kernel (3) and adds the new boot command line and the initrd parameters (4). By copying the current parameter block, all other values set by the BIOS, e.g. the memory configuration, are preserved. Along with the pointer array to the source data, bootimg also maintains a second array (5) that contains the target addresses in physical memory for all pages. Once all this is done, bootimg sets up a descriptor containing pointers to the two arrays and some additional information (6), and invokes the `bootimg` system call (7).

As shown in figure 15, the `bootimg` system call first copies the source pages to kernel memory (1). This is done mainly in order to check access permissions and to ensure proper alignment of the pages, but it also makes it easier to implement the crash dump utility described in the next section. When copying, `bootimg` also updates the source pointers (2) to point to the new pages in kernel memory.[8] Since the pages have been allocated at arbitrary locations, they must be moved to the right place before the kernel can be started. This is done by a little position-independent function that is copied to its own memory page (3). This function moves all pages to the location indicated in the target address array (4). If a target address happens to coincide with a page that is still needed, the function copies the content of the target page first to a free page. Note that this may also include the page containing the function itself. Once all pages have reached their destination, the startup code of the new kernel is called (6).

Two likely future changes are the addition of support for references to physical pages in the source pointer array in order to support copying of data that may change after the call to `bootimg` (i.e. the kernel message buffer), and a split of the bootimg user-space program into a set of library functions and a simple utility calling them, in order to make it easier to use bootimg in other programs.

[8] It actually does this in two steps: first, it uses addresses in the kernel address space. Then, immediately before reordering the memory pages, it changes them to addresses in physical memory. This way, the addresses are still available if any operation fails before the reordering, and the pages can be freed before the system call returns. This would be more difficult if the addresses were already translated to physical memory addresses, because the latter can not generally be converted back to kernel address space.

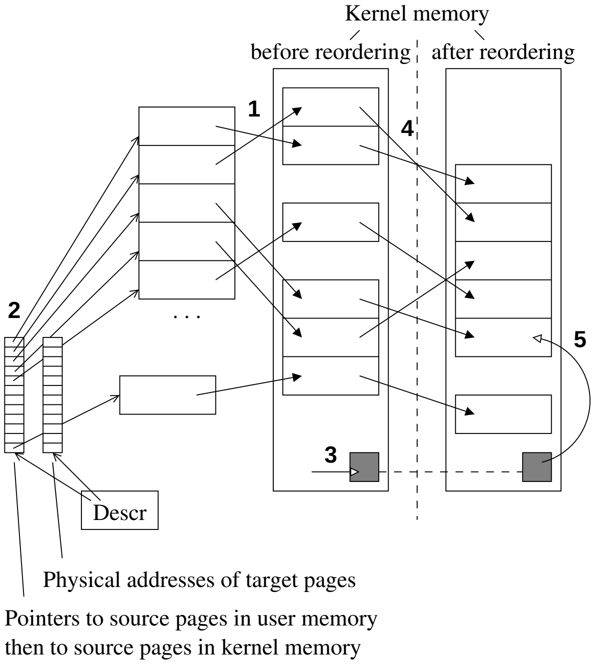

Figure 15: Bootimg: memory reordering in the kernel.

5.4 Other interesting applications

Besides just booting Linux kernels from odd sources, two other possible applications for such a mechanism have been proposed recently.

LinuxBIOS [2] takes the reduction of boot loader functionality to the logical extreme and simply puts a Linux kernel in the Flash EPROM that normally holds the PC BIOS. This kernel can then act as a very feature-rich boot loader.

Another interesting use is the creation of crash dumps. Many traditional Unix systems can write the memory content to disk when a kernel panic occurs. A crash dump can later be analyzed to determine what has caused it. Since a kernel panic should only occur in cases where the kernel has detected a serious defect, it is not safe to assume that the normal drivers can be used for writing that crash dump. Even if the drivers still work, using them may change the system state such that the problem leading to the kernel panic can no longer be discovered.

It is therefore desirable to use an subsystem that is independent from the regular kernel for this task.

With a mechanism like bootimg, this is quite simple: a small kernel for taking the crash dump is pre-loaded along with a suitable initial RAM disk, and when a panic occurs, the pre-loaded pages are checksummed (they may have been damaged as a result of the problem leading to the kernel panic), and the kernel is launched. It can then set up a new clean environment and write the dump.

An implementation of such a crash dumper, based on bootimg, can be found at [18].

6 Acknowledgements

Many people have contributed to LILO over the years by reporting bugs and suggesting improvements. Development has stalled in the last years, but John Coffman is now carrying on the torch with fresh energy.

The architectures for the initial RAM disk and for bzImage are a joint work with Hans Lermen.

The design of `pivot_root` was strongly influenced by discussions in the linux-kernel mailing list. In particular, comments from H. Peter Anvin, Linus Torvalds, and Matthew Wilcox helped to shape the current design, and Alexander Viro is currently refining the concept.

The basic idea for bootimg comes from an implementation for SVR4 written by Markus Wild in the early nineties. The memory reordering algorithm of bootimg was strongly inspired by FiPaBoL, designed mainly by Otfried Cheong and Roger Gammans, and implemented by the latter.

7 Conclusion

Table 1 shows the evolution of boot concepts in the history of Linux. Items still under development are shown in *italics*. Also, boot loaders for other architectures than i386 have been omitted.

The first boot loaders plainly got the kernel loaded, without much convenience beyond this. The second generation of boot loaders overcame the file system type constraints and added many useful features, such as the boot command line or the ability to boot other operating systems. Almost all boot loaders in use today are of the second generation.

The ability to use arbitrary file systems as the root file system evolved slowly since the beginning of Linux. Since the introduction of `pivot_root`, a completely generic solution is available.

Finally, the ability to load kernels from other sources than floppy or hard disks is comparably recent. Since the three current approaches to boot Linux from Linux are already quite generic, convergence will probably be reached soon.

As has been shown, the apparently simple act of booting a Linux system is full of interesting problems. Modern Linux systems offer a rich set of features to handle those problems, and even more exciting improvements continue to be developed.

References

[1] Almesberger, Werner. *LILO User's guide*, `ftp://metalab.unc.edu/pub/Linux/system/boot/lilo/`

[2] Minnich, Ron; Hendricks, James; Webster, Dale. *The Linux BIOS Home Page*, `http://www.acl.lanl.gov/linuxbios/`

[3] Anvin, H. Peter. *SYSLINUX*, `http://www.kernel.org/pub/linux/utils/boot/syslinux/`

[4] Kuhlmann, Gero. *Netboot*, `ftp://metalab.unc.edu/pub/Linux/system/boot/ethernet/netboot-0.8.1.tar.gz`

[5] Lermen, Hans. *LOADLIN*, `ftp://metalab.unc.edu/pub/Linux/system/boot/loaders/lodlin16.tgz`

[6] Cheong, Otfried. *Arlo – Arm boot loader*, `ftp://ftp.calcaria.net/pub/arlo051.tgz`

[7] Boleyn, Erich; et al. *GNU GRUB*, `http://www.gnu.org/software/grub/grub.html`

[8] Almesberger, Werner; Coffman, John. *LILO – Generic boot loader for Linux*, `ftp://metalab.unc.edu/pub/Linux/system/boot/lilo/`

The humble beginnings	
1991	Linux boots stand-alone from floppy.
	Shoelace is used to boot from Minix file system on hard disk.
Beyond Minix	
1992	LILO allows booting from (almost) arbitrary file systems and of other operating systems.
	BOOTLIN allows booting from DOS.
1994	LOADLIN replaces BOOTLIN.
	SYSLINUX reads FAT (MS-DOS) floppies.
1995	GRUB, a modern file system aware boot loader.
Root file system abstraction	
1991	Root file system device can be set in kernel image.
1995	NFS root mounts root file system from NFS server.
1996	Initial RAM disk support added to kernel.
	`change_root` mechanism.
2000	`pivot_root` mechanism.
	Union root mount.
	Early freeing of initrd memory pages.
Kernel image abstraction	
1996	Netboot boots from Ethernet, using TFTP.
1999	GRUB supports TFTP boot too.
2000	*Linux boots Linux.*
	LinuxBIOS.

Table 1: Evolution of the boot process. (Work in progress is shown in *italics*.)

[9] Almesberger, Werner. *LILO Technical overview*, ftp://metalab.unc.edu/pub/Linux/system/boot/lilo/

[10] Brouwer, Andries. *Large Disk HOWTO*, http://www.win.tue.nl/~aeb/linux/Large-Disk.html

[11] Phoenix Technologies Ltd. *Enhanced Disk Drive Specification Ver 1.1*, http://www.phoenix.com/products/specs-edd11.pdf

[12] Almesberger, Werner. *ush – micro shell*, ftp://icaftp.epfl.ch/pub/people/almesber/psion/ush-2.tar.gz

[13] Brouwer, Andries. *util-linux: Miscellaneous utilities for Linux*, ftp://ftp.win.tue.nl/pub/linux-local/utils/util-linux/

[14] Brouwer, Andries. *man pages for Linux*, ftp://ftp.win.tue.nl/pub/linux-local/manpages/

[15] Minnich, Ron. *LOBOS: (Linux OS Boots OS) Booting a kernel in 32-bit mode*, http://www.acl.lanl.gov/linuxbios/papers/lobos.ps

[16] Hendriks, Erik. *Two Kernel Monte (Linux loading Linux on x86)*, http://www.scyld.com/software/monte.html

[17] Almesberger, Werner. *bootimg* ftp://icaftp.epfl.ch/pub/people/almesber/misc/bootimg-current.tar.gz

[18] Mission Critical Linux. *Kernel Core Dump*, http://www.missioncriticallinux.com/technology/coredump/