Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Embedding Linux

David Woodhouse
Red Hat, Inc.
dwmw2@cambridge.redhat.com

Abstract

Linux is becoming widely accepted in the embedded systems arena. This paper will give a brief overview of the applications for which it is currently being used and new applications for which development is in progress, and will discuss the requirements and problems which are unique to such embedded applications.

Some discussion will also be given to situations in which Linux is *not* the most appropriate tool for the task at hand, and in which a smaller, more application-specific operating system such as eCos may be more useful. *[eCos]*

1 Introduction

Linux was developed as a general-purpose operating system. A single kernel is intended to scale usefully from handheld devices such as the Compaq iPAQ and Sharp Zaurus to "Big Iron" such as the IBM zSeries mainframes.

For many years, Linux has commonly been used on PC machines as a router. More recently, Linux has been used by many companies in embedded "black box" products intended for applications such as network routing and firewalling, file and print serving, web serving, and in one well-known case for recording to hard disk and playback of television programmes. These applications typically use PC-class or similarly powerful hardware, and make no particular requirements on the kernel that traditional desktop and server applications do not.

Linux is also becoming widespread on smaller hardware such as Personal Digital Assistants (PDAs), especially the Compaq iPAQ and now the Sharp Zaurus, which is one of the first PDAs to be shipped with Linux *instead* of Windows CE, rather shipping with Windows CE but having Linux available for installation. These devices have limited battery capacity, very limited amounts of flash memory available for storage, and small displays with touch screens. Therefore, the use of Linux on such devices has motivated much development in the areas of power management, flash storage and code size reduction, and user interfaces targetted at such displays — all of which are discussed later.

2 Scaling down

A significant criterion affecting decisions regarding embedded applications is the cost per unit. Significant up-front costs and development time will be borne in order to reduce the

per-unit cost of hardware and software by pennies. This is partly why Linux in the embedded space is so attractive to many developers of such products — the per-unit licensing cost of Linux and most Linux applications is zero.

The importance of per-unit cost means that hardware resources are often strictly limited. The cost of extra RAM and storage space which may be required is equally important when comparing Linux against alternative operating systems; any wastefulness of resources cannot be tolerated in mass-production. Therefore, there is significant effort required to ensure that Linux remains "lean and mean," without gratuitous increases in the demands made from hardware.

Although Linus is fairly good at guarding against the introduction of gratuitous bloat, this is still not a particularly easy task. Figure 1 shows the size of the Linux source tarball since the first releases[1]. Since Linux v0.01 with a gzipped tarball size of 73091 bytes, the Linux kernel has grown exponentially over time to reach roughly 32 MiB at the time of writing, with the 2.5.12 kernel.

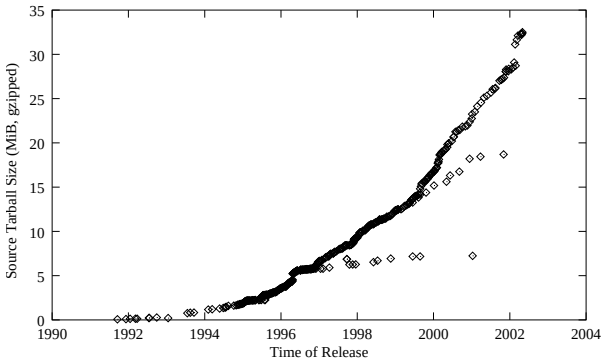

Figure 1: Growth of the Linux kernel.

Thankfully, this exponential increase in source code size does not translate linearly to an increase in object code size. Much of the increase in the source code size is optional features and new architectures and subsystems.

Nevertheless, the kernel *is* growing steadily. The transition of the μClinux code base from the 2.0 series of kernels to 2.4 was delayed significantly when it was realised that the resulting kernel images would *double* in size. This is largely due to expansion in code which has not been made conditional. Especially in the networking parts of the kernel, new features are often added unconditionally, without much consideration for situations where they will be unrequired.

There has been a great deal of work recently on "scalability" of Linux, with a lot of publicity and large companies getting involved. Mostly, attention has been concentrated on scaling *up* to large multi-processor and NUMA boxes; reducing lock contention and bouncing of shared cache lines, dealing efficiently with large amounts of memory and storage space, etc.

However, there has also been less visible work on scaling Linux *down* — reducing the storage and RAM footprint of both userspace and the kernel, and to a certain extent keeping the other developers honest by ensuring that optimisations made for larger boxes are not pessimisations for the embedded targets.

Inevitably, there have been some trade-offs required to accommodate the vast range of target hardware supported by Linux, but the most important such choices have been made configurable by the user at compilation time, so features can be included or omitted at will.

One area which is currently receiving attention is the support for block devices. Many embedded boxes have no block devices of any kind, and do not require any of the support for block I/O which is currently built into each ker-

[1] Graph thanks to James Smaby.
http://virgo.umeche.maine.edu/misc/kernel_size/

nel image. Significant size savings could be made by stripping this code out. However, the task is complicated; the block I/O subsystem has always been present in the Linux kernel and a large amount of code is written with the assumption that this will remain true. Large amounts of file system core (VFS) code need attention, but mostly to separate the block-related functions from generic file system code to allow conditional compilation. More complex is the virtual memory subsystem, which is littered with assumptions about the presence of block I/O, which is required for paging.

3 μClinux

μClinux, mentioned above, is a port of Linux to microcontrollers without memory management units. In embedded single-user and single-application boxes, a memory management unit serves little purpose. If the whole point of the box is to run a single application, it matters little whether a crash of that application can scribble over kernel memory and kill the kernel too, or whether the application will take a page fault and be killed — if the application dies, the game is over already. Hardware watchdogs to reset the board in the event of a malfunction are just as effective as a script to restart an individual application if it crashes; and of course neither are a suitable substitute for ensuring that the application doesn't crash in the first place.

The MMU, therefore, is a prime candidate for removal when counting the pennies which are being spent for each unit shipped.

Surprisingly, the Linux kernel code is not particularly intrusive. Most driver, networking and file system code does not need modification to accomodate the lack of MMU support. The memory management code in the `mm/` directory is replaced with equivalent routines in a parallel `mmnommu/` directory, and the architecture-specific code is also replaced with a new version.

Of course, most kernel code is designed to run in a single address space with no protection on memory accesses anyway. In userspace, the distinction is far more important. Because applications must *also* share a single address space, a drastic rearrangement of how user space programs are loaded was required. μClinux uses a new type of binary, known as 'flat' format. μClinux executables contain Position Independent Code (PIC) and hence the text segment containing the program instructions can be loaded anywhere inside the available address space, as can the data segment.

Although μClinux is not merged with the mainstream Linux kernel, the unintrusive nature of the port means it remains a possibility, and key kernel developers have repeatedly expressed a desire to do so.

μClinux supports a wide range of platforms and CPUs including Motorola m68k-based CPUs, ARM, Axis' ETRAX and the Intel i960. It is used in a number of products using μClinux including MP3 players, voice-over-IP telephones, Network Cameras, routers, etc.

4 Low-fat libraries and utilities

After the kernel, the next obvious large object in a Linux system is usually the C libraries. The GNU C library, glibc, is a multi-megabyte monster which can account for a large proportion of the available storage and RAM space on a small device. The maintainer of GNU libc, Ulrich Drepper, has clearly stated that

GNU libc is not targetted at embedded systems: *"...glibc is not the right thing for [an embedded OS]. It is designed as a native library (as opposed to embedded). Many functions (e.g., printf) contain functionality which is not wanted in embedded systems."* *[Drepper]*

Thankfully, there are alternatives to glibc. The older Linux libc5, the endpoint of the branch which was originally taken from GNU libc 1.07.4 to add Linux support, is still maintained and is significantly smaller than glibc. Also, there at least two C library implementations specifically designed for a small footprint.

As mentioned above, the μClinux kernel required drastic changes to userspace libraries. A new C library, μClibc, was developed for use with μClinux. After it became apparent that there was a need for a bloat-free C library in full MMU-capable Linux systems too, support for such systems was added to μClibc. μClibc supports most target architectures on which embedded Linux is found, including ARM, MIPS, PowerPC and Hitachi Super-H. μClibc is licensed under the GNU Lesser General Public License, and is available at http://www.uclibc.org/.

There is also diet libc, which claims to achieve even better code size reduction than μClibc by rewriting far more routines rather than copying them intact from other sources. The diet libc is licensed under the GNU General Public License, not the LGPL, and is available at http://www.fefe.de/dietlibc/.

Both μClibc and diet libc have multifunction binaries associated with them which can replace a large number of standard utilities such as `cat`, `mv`, `cp`, `ln`, etc. By using a single multifunction binary such as these to replace a multitude of overly feature-laden GNU utilities, further dramatic improvements in required space can be achieved.

BusyBox works with both μClibc and glibc, and is available at http://www.busybox.net/.

The set of utilities which works with diet libc is called "embutils" and is available at http://www.fefe.de/embutils/.

5 Power management

Another area which has received much attention in Linux, and still requires further development, is power management. Traditionally, Linux would power up and initialise devices at boot time or when the driver module was loaded, and would keep them powered at all times thereafter, often not even powering them down when a driver module was unloaded. This is extremely wasteful of power, which is extremely important on battery-powered computers such as laptops and handheld devices.

Obvious improvements are achieved by modifying drivers to remove power from unused circuits while devices are inactive. Often, this precise control over the application of power is very platform-specific, but hooks are required in generic code such as UART drivers so that the platform-specific code can be called at appropriate times when the port becomes active or inactive.

A great deal of work has therefore been done on extending the device driver APIs to accommodate power management facilities.

Suspend modes

Many battery-powered systems support a mode where all circuits except the RAM can be disabled and even the CPU can be placed into a low-power state until woken by an interrupt.

In order to enter this state and correctly return from it, it is necessary to maintain information about bus connectivity so that devices can be powered down before the busses which connect them, and the resumption of power can be performed in the opposite order.

Once all devices have been powered down, the entire CPU state can be stored in memory, the RAM can be switched into a self-refresh mode and even the CPU can be placed into an extremely low-powered state, to be woken only by a specially-configured interrupt. Often, only a single 32-bit register is retained over such a sleep state, and the CPU will start to execute the boot loader from the reset vector when it wakes just as it would after a normal power up cycle. The boot loader must then check the contents of the register and behave appropriately if it detects that it's waking from a sleep state, not a hard reset. Usually, the value in the register is a return address, and the boot loader will switch the RAM back to its normal state and jump back to the kernel code at the specified address.

This mechanism is used on PDA devices to implement the "instant-on" power mode which is reached by pressing the power button. A complete reset and reboot is rare, and usually requires pushing or switching a recessed reset or battery disconnect switch.

Frequency scaling

A CPU will consume less power when running at slower speeds, and many current CPUs can dynamically scale their clock speed under software control.

In addition to removing power to individual devices and circuits and shutting down the CPU completely, it is also possible to achieve power savings by utilising this facility to reduce the speed at which the CPU runs to match the current requirements of the running system.

Scaling CPU speed dynamically requires careful changes to timing-related functions and CPU-external bus timing. Basic support for management of CPU clock scaling is being developed and is present in the 2.4 version of the kernel for the ARM architecture. CPUs which are supported include ARM Integrator, SA1100 and SA1110. The various Intel IA32 clone manufacturers each have their own method of clock scaling, and CPUFreq contains support for AMD PowerNOW and VIA Cyrix Longhaul technologies.

Support for the Intel SpeedStep method of clock scaling is lagging far behind the rest, because Intel have so far refused to give sufficient documentation; preferring to push ACPI as their preferred method of accessing such functionality. Essentially, ACPI provides control methods in a form of interpreted bytecode similar in concept to Java, which must be trusted by the Linux kernel and run in privileged mode. This is no substitute for true GPL'd Linux drivers for the hardware in question.

Another power-saving feature which is not yet implemented but which is planned is the possibility of removing the system timer interrupt. Currently, Linux systems have a fixed-frequency interrupt, often at a frequency of 100 Hz, which is used for keeping system time and for running timers. If the CPU is entering a low-power state during idle periods, it must wake up and run the interrupt service routine every 10 ms — usually to find it has nothing to do but go immediately back to sleep again. This causes a significant power drain which should be unnecessary. Therefore, it is planned to develop code which allows Linux to abolish the fixed-frequency timer interrupt and instead

use a one-shot timer to set a wake-up time each time the low-power idle state is entered. The CPU will be woken either by the first pending timed event or by interrupts from other sources such as I/O devices.

This improvement will be useful not only for embedded devices where power consumption is paramount, but also at the opposite end of the spectrum; on mainframe hardware where many hundreds of Linux kernels may run inside virtualised machines, and the overhead of a timer interrupt on *every* virtual Linux machine each few milliseconds quickly starts to take a significant proportion of the available CPU time.

6 Hotplug capabilities

Linux has for a long time supported PCMCIA and CardBus peripherals; 16-bit PCMCIA being significantly more common on handheld devices than CardBus. The Linux PCMCIA code is based heavily on the architecture laid out in the PCMCIA specification, which seems to be overly complicated and designed for legacy drivers and MS-DOS. This level of complexity appears to be overkill for Linux, and work has started on a rewrite of the PCMCIA support based solely on the reality of PCMCIA hardware rather than the intricacies of the PCMCIA specification. This work has yet to reach a state in which it can be announced to the public for further development.

In networking and control applications, Linux is also often required to support hot-swapping of PCI and CompactPCI peripherals. Basic support for dynamic addition and removal of PCI devices is present in the Linux kernel — each device driver must be individually upgraded to the new PCI driver API to be capable of supporting hot-swappable devices, and this has not yet happened for all drivers.

Support for physical insertion and removal of devices, probing of new devices and notification of drivers is implemented. Recently, some support for correct handling of the CompactPCI procedures for device insertion and removal, involving notification of the opening of the removal handle, lighting of the appropriate LED to signal that the system is ready for device removal etc.

One severe problem currently faced by the existing PCI hotswap code is the assignment of address space resources to newly-inserted cards. There is a limited amount of physical address space which may be assigned to BARs of PCI devices, and this space is further subdivided by having to configure ranges for each PCI bridge in the system, with each bus getting a single range of each type of address space. The current approach is to reserve some address space for each PCI bus which may accept hot-swap cards, in the hope that it will be enough. Yet if multiple PCI busses are present, then by repeatedly inserting and removing cards on different busses it is possible to fragment the allocation of resources to the extent that a newly-inserted card cannot be assigned a range of address space on the bus into which it has been introduced. Therefore, it is being proposed that another addition to the Linux PCI driver API be considered, which allows supporting drivers to have the BARs of their devices moved by the core PCI code to make room for other devices in the address space.

In many cases, it should be sufficient for the driver to momentarily quiesce the card, to prevent interrupts from occurring during the relocation, change the BARs to the new address range given by the PCI code, and reenable the device. Virtual mappings of memory BARs will need to either torn down and set up again

for the new location, hence the need to quiesce the hardware rather than simply disabling interrupts while performing the move.

If accepted and implemented, this enhancement will allow for more reliable management of resources, assuming that the drivers for all hot-swapped cards provide support for this method of relocation.

7 Storage

The storage requirements of embedded devices differ significantly from traditional Linux installations. Often, the only storage available will be flash memory. Flash is a form of solid-state storage which provides persistant storage with low power requirements and relatively low cost.

The most common form of flash is NOR flash. This can be connected directly to the CPU's address and data busses and for reading is treated as ROM. As with ROM, each bit of storage can be in one of two states — either it contains a zero or a one.

Each bit of storage in NOR flash chip will start containing a one, and by a predefined sequence of writes of magic numbers to magic addresses, the contents of each bit can be individually changed to zeroes.

However, bits which have been cleared cannot be individually reset to contain ones again. Bits can only be reset to ones, or "erased," in large blocks of typically 64 or 128 KiB in size, known as "erase blocks." Furthermore, the lifetime of a flash chip is measured in erase cycles; typically each block can be erased 100,000 times before it is expected to fail.

It is important to note that the lifetime of flash chips is measured in erase cycles *per block*, not total erase cycles. Individual erase blocks can be erased to the point of destruction without affecting other erase blocks in the chip.

Therefore, by repeatedly erasing a few blocks it is possible to destroy them while the remainder of the chip is still usable — however, even with appropriate detection of bad blocks this reduces the storage capacity of the device, and as the storage available is unlikely to have exceeded the *required* amount by any significant margin, would quickly lead to the device being unusable.

Therefore, it is necessary to perform "wear levelling" on flash devices, to ensure that the block erases are evenly distributed over the entire range of the chips rather than concentrated in particular areas. This is particularly important because the normal use of permanent storage will be precisely the opposite of what is required — typically a device with 16 MiB of available flash would have 14 MiB of static data, programs and libraries, 1 MiB of dynamic data and 1 MiB of space; without wear levelling the 14 MiB of static data would never be moved and the remaining 2 MiB of the chip would be destroyed very quickly.

In addition to the need for wear levelling, the large block size of flash means that traditional file systems cannot easily be used, as they rely on being able to replace data blocks in-place, which is not possible on flash without also erasing and replacing the surrounding data in the rest of the same block. There is an extremely naïve driver available for Linux which does present a flash device to file systems as a block device with 512-byte sectors, then on writes will read the whole erase block, modify the contents as desired and then write back the new version. This is obviously extremely unsafe, but can be useful for setting up file sys-

tems which are going to be read-only in production.

The traditional approach to using flash has been to use a form of pseudo-filesystem on the raw flash to emulate a normal block device with smaller sectors. This solution evolved in the days of DOS, where providing an INT 13h disk service interrupt was sufficient.

In practice, this is very suboptimal. To ensure reliability, the pseudo-filesystem used must be a journalling one - it must be able to revert to a consistent state if power is lost or a crash occurs during a write. Furthermore, the traditional file system used on the emulated block device must *also* be a journalling file system, for precisely the same reasons. The result is a journalling file system running atop another journalling file system, which is inefficient in terms of both speed and wear on the flash devices.

A better approach is that taken by the Journalling Flash File Systems, which are designed to operate directly on the underlying flash device rather than through an intermediate emulation layer. *[JFFS]*

These file systems are log-structured, writing packets of data to the flash describing each *change* to the file system, and requiring a complete playback of those logs on remounting of the file system to recreate the current contents of the file system. As the log progresses, older log entries (or "nodes") will be obsoleted by newer entries which overwrite the old data, delete files, etc.

When the medium becomes close to full, the system must perform garbage collection to reclaim the space taken by such obsoleted nodes. An erase block is selected for garbage collection and the nodes which are still relevant are copied into the remaining empty space, before the victim block is erased. More details of the operation of these file systems are given in the referenced paper.

Since its development in the first quarter of 2001, JFFS2 has rapidly become extremely common in the deployment of embedded Linux devices with flash storage.

In addition to the common NOR flash, support has recently been added to JFFS2 for NAND flash. NAND differs from NOR flash in that it is not directly accessible as if it were ROM; instead data, addresses and commands are exchanged a byte at a time over a single 8-bit bus. NAND flash is smaller erase block sizes than NOR, typically around 8 KiB, and is further subdivided into "pages" of typ. 512 bytes, each of which is associated with a further range of "out-of-band" data, used for ECC and metadata. NAND flash chips are cheaper than NOR flash, and tend to have higher production tolerances, leading to higher incidence of bit errors and bad erase blocks.

Execute in place

A feature which is not implemented in Linux is "execute in place" (XIP). This refers to the arrangement where data are not copied from the flash medium into RAM, but are used directly by entering pages of the flash chip directly into the page tables of user space processes.

In many situations, XIP is not desirable. For obvious reasons, XIP and compression are mutually exclusive — if data are compressed, they cannot simply be used in-place. In terms of cost per byte, flash is generally more expensive than DRAM, hence the cost savings from using compression and reducing flash requirements are more than the cost savings from using XIP and reducing RAM.

However, XIP becomes a more sensible option in situations where low power consumption is an extremely important criterion. In this situation, static RAM may be used instead of DRAM, and this is normally more expensive than flash.

The implementation of XIP presents some interesting problems which have yet to be properly solved. Obviously, it can only work with NOR flash technology, as NAND flash cannot be directly accessed. The problem with NOR flash is that the write and erase commands are performed by writing magic numbers to magic addresses within the chip and then reading status words back from the chip. When the flash chip is in a command mode, the values returned on read accesses are not necessarily valid data. The flash drivers handle this by keeping a state machine and ensuring that the chip is always in the correct mode by sending the appropriate commands before performing any operation, including any reads from the device.

However, if pages of the flash chip are simply mapped into user space processes using the MMU, it is not possible to ensure that the proper sequences are followed; either scheduling must be disabled during the entirety of each period for which the flash chip is placed in a mode other than read mode, or every mapping of a flash page to user space must be found and torn down before each such access. As erase and write operations may take an extremely long time, the former option is not particularly feasible. Until recently, the latter was not possible either — only with the advent of the memory managment code based on reverse mappings of physical addresses to virtual was it possible to find all the mappings of a given page without scanning the entire address space of every process in the system.

Now that the rmap VM has made XIP at least technically feasible, there are plans to adapt the flash drivers to accommodate this form of mapping. However, there remains the problem of designing a file system which can make use of this facility. In order for XIP to be used, each page of file data must be page-aligned in the flash chip, because no common MMU hardware allows for the remapping of arbitrary byte ranges. This effectively means that the JFFS2 mode of operation, writing a node header followed immediately by a payload, is extremely suboptimal. An ideal file system designed for XIP would separate metadata from pages of data, yet still perform in a broadly similar manner to JFFS2. However, designs for such a file system have yet to come any closer to completion than the above.

Removable storage

There are a multitude of forms of removable solid state storage. The best supported by Linux is CompactFlash, which presents itself to the system as an IDE device. It uses a translation layer as described above to emulate a block device, but this is performed internally to the device, and the host computer treats it exactly as if it were a normal IDE drive. Some CompactFlash devices perform wear levelling internally, but some do not. It is often not easy to tell whether a particular device performs wear levelling or not.

Another common form of removal storage is SmartMedia. SmartMedia is effectively just a NAND flash device, and the host computer must implement a similar translation layer in software. Linux does not currently support the SmartMedia translation layer, although there are drivers under development. However, there exist USB devices which perform the necessary transformations in their own firmware, presenting an interface to the host computer which is a simple USB storage device.

8 User Interface

With the advent of handheld devices running Linux, the user interface has become extremely important. Once the initial excitement of reaching a shell prompt over the serial console has passed, it rapidly becomes apparent that traditional Linux graphical user interfaces based on X Windows are not ideally suited for use on a 320x200 pixel display.

9 Alternatives

10 Conclusion

Linux has come a long way since Linus first played with multitasking by making a kernel which would interleave his two processes printing ‘AAAAAA’ and ‘BBBBBB’. Linux is becoming widely accepted in the embedded market, and there has been a great deal of good work on improving its applicability to these targets.

Still more development is currently under way and being planned, and it is clear from the discussions above that there remains a lot more work to be done in these areas in the future; certainly there’s plenty to keep us from getting bored.

References

[eCos] Red Hat, Inc., *eCos — Embedded Configurable Operating System.* http://sources.redhat.com/ecos/

[Drepper] Ulrich Drepper <drepper@cygnus.com>, posting to bug-glibc@gnu.org mailing list, 24 May 1999. http://sources.redhat.com/ml/bug-glibc/1999-05/msg00039.html

[JFFS] David Woodhouse, Red Hat, Inc. *JFFS: The Journalling Flash File System*, May 2001. http://sources.redhat.com/jffs2/jffs2.pdf