LINUX MOBILE
A Platform for Advanced Future Mobile Services

Frode Sivertsen
Dept. of Telematics, Norwegian University of Science and Technology
O.S. Bragstads Plass 2E, N-7491 Trondheim, Norway

Ivar Jørstad
Ubisafe, Bjølsengata 15, N-0468 Oslo, Norway

Do van Thanh
Telenor R&D, Snarøyveien 30, N-1331 Fornebu, Norway

Keywords: Linux Mobile, Embedded Systems, Soft Real-Time kernel, Application Development Platforms.

Abstract: Linux has for some time been the operating system of choice for many types of embedded devices (e.g. network devices like routers, as well as multimedia devices like set-top-boxes). Currently, Linux is also gaining momentum as an operating system for mobile phones. This paper studies what it takes to make Linux "go mobile", i.e., what adaptations are necessary to make the Linux kernel fit as a mobile operating system, what is the architecture of such a platform, and what are the major benefits.

1 INTRODUCTION

Linux already exists in several commercial distributions targeted for embedded platforms and currently has about 23% of the world market share on mobile phones, even though this number provided by The Diffusion Group can be disputed. (The Diffusion Group, 2006) (Blandford, 2006) With the development of the handheld device hardware, Linux is of particularly interest. It has been ported to several hardware architectures for years, it has one of the most stable kernels, and the functionalities of the handheld devices are growing to be more and more similar to that of a "regular" PC. Major embedded Linux vendors such as MontaVista, and Trolltech are serving more and more customers with development environments partially based on proprietary software every day.

During the first half of 2007 one of the most anticipated releases of a Linux driven mobile phone will be ready for shipping, the Neo1973 from First International Computing, FIC. Linux is nothing new as a mobile phone operating system, but this is the first mobile phone which will be shipped with completely open source software based on the OpenMoko platform. (Cheap, hackable Linux smartphone due soon, 2006) (OpenMoko: The World's First Integrated Open Source Mobile Communications Platform (n.d.).)

Many in the handheld operating system community favours Linux as the de-facto operating system for handheld devices to be, because of its openness, flexibility, broad developer base, and its modularity. They predict a new value added feature in the next generation of mobile phones where the applications may become the ringing tones of today (Purdy, 2007).

With the release of the 2.6 kernel of Linux, it has gone further in providing real-time services but yet keeping the advances features compared to regular real-time operating systems. Linux positions itself with the advantages from both the real-time operating systems and the microkernel operating systems. Compared to its major competitors, being Symbian and Windows, it has its already mentioned advantages, but the performance is just as good as that of the mobile targeted operating system of Symbian (Benchmark clocks OMAP2420 graphics on Linux, Symbian, 2006).

Sivertsen F., Jørstad I. and van Thanh D. (2007).
LINUX MOBILE - A Platform for Advanced Future Mobile Services.
In *Proceedings of the Second International Conference on Wireless Information Networks and Systems*, pages 230-237
DOI: 10.5220/0002149302300237

These are just some of the reasons why it is believed that Linux actually has the potential to become the de-facto mobile operating system of the future phones.

2 INTRODUCING LINUX

The components that form Linux do not change much whether they run on a server, a workstation, or a mobile phone. The Linux kernel is what is referred to as a monolithic kernel. Basically it consists of an architecture-dependent low-level interface that interacts with the hardware. However, it provides a hardware-independent API to the higher layers (i.e application layer and libraries) through high-level abstractions which can have a constant code base. The high-level abstractions are processes, files, sockets, signals etc.

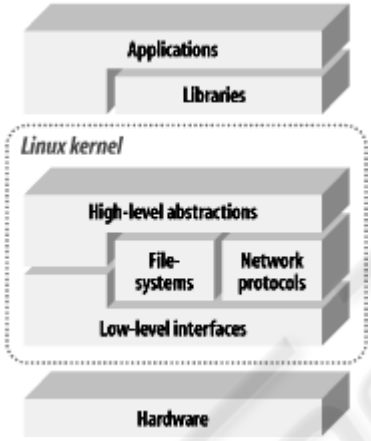

Figure 1: The architecture of a generic Linux system. (Yaghmour, 2003).

The interpretation components such as file systems and network protocols are used to understand how to interact with the devices present on the platform. Many standards have been developed throughout the years, and because of its many portings, Linux supports more than its competing operating systems.

On top of the high-level abstractions one find the libraries that acts as standardized API's for the application layer, since the services exported by the kernel are often unfit to be used directly by the applications. (Yaghmour, 2003) This is, as already mentioned, one of the areas where Linux has its strength. C, C++, Perl, Java etc. are languages easily supported by the Linux kernel through various libraries. This can be custom fit, regulating the size of the operating system footprint.

For the graphical user interface Linux supports several window managers and graphical libraries. The X Window System, X11, which usually runs on most desktop distributions is quite large, requires 8 MB of RAM and was originally made as a client/server application. The most used open source window managers for handheld devices are Nanowindows, formerly known as Microwindows, and Matchbox. In contrast to their "big brother" X11, they have reduced resource requirements. Other window managers intended for embedded devices exist as well. (Embedded Linux Graphics Quick Reference Guide, (n.d.).)

The window managers usually use graphical libraries such as Nano-X, Qt/Embedded, and GTK+ possibly with GTK-DFB and GTK-X, to provide the GUI. Trolltech, the makers of Qt, have a rather complex license model while GTK+ is completely GPL licensed.

3 LINUX AS A SOFT REAL-TIME OPERATING SYSTEM

Regular real-time operating systems are mainly made for MMU-less processors with a flat address space with no memory protection between the kernel and its running applications. This means that the kernel, the kernel subsystems, and the applications share the same address space and must therefore be made foolproof to avoid crashing the system. This makes adding new software difficult. The system must also be brought down to do this.

A microkernel provides a very small operating system footprint which offers only the most basic services such as scheduling, interrupt handling, and message passing. The rest of the operating system, such as file systems, device drivers, and networking stack, runs as applications with their own private address space. The microkernel is dependent on well defined APIs for communication with the operating system and robust message-passing schemes between processes. Only that way might real-time services and modularity be ensured.

Linux is built up by several subsystems that can be dynamically loaded into the kernel, such as the file systems. This, however, do not make it a microkernel-based operating system. The kernel still interacts with the drivers using direct system calls, and not through message passing between processes. Message passing between processes can be very resource consuming and is regarded as one of the major drawbacks of microkernel operating systems. The dynamically loadable kernel module are pieces of kernel code that are not directly included or

linked in the kernel, but can be inserted and removed from the running kernel at almost any time.

Any new code intended for the Linux kernel goes through a great deal of testing regarding design, functionality, and performance before it gets accepted into the mainline kernel releases. Hence, this trying process has looked after the advantages of "regular" real-time operating systems and made it one of the most stable pieces of software. At the same time it has kept the advantage of the memory protection to individual kernel subsystems provided in microkernels, but avoided the resource consuming message passing. These are some the reasons why Linux have become so popular (Raghavan, Lad and Neelakandan, 2005).

3.1 User Mode and Kernel Mode

The monolithic kernel of Linux has a distinction between kernel and user mode execution states to secure the memory protection. A process in User mode can not enter kernel programs or kernel data structures directly. The User mode programs issue system calls to enter Kernel mode. The time before a system call is being served depend on the interrupt signal sent from the process to the CPU and its actions according to the interrupt (Bowet and Cesati 2001:1-34).

3.2 Re-entrancy

The Linux kernel is re-entrant, meaning that several processes may be executing in Kernel Mode at the same time. Only one process can progress at the time in a uniprocessor system, but others may be waiting for the completion of some I/O request or the CPU. To provide re-entrancy, the functions must only modify local variables, not global ones.

The kernel may also include non-re-entrant functions that use locking to ensure that only one process can execute that function at a time. These processes may then modify global variables. If an interrupt occurs, the kernel is able to suspend the running process even if it is in Kernel Mode. This ensures a higher throughput for the device controllers that issue interrupts. While the kernel handles the interrupt, the device controller may perform other tasks.

The re-entrancy influences the organization of the kernel and its *kernel control path* which denotes the sequence of instructions executed by the kernel, being an interruption, a system call or an exception. Normally the kernel would execute these tasks one by one, from the first to the last. However, during handling interrupts and exceptions, the kernel can interleave one process in Kernel Mode to run a process required by the first one or run another process until the first one can be continued due to waiting on an I/O operation. Re-entrancy requires the implementation of interprocess communication, which will be described shortly (Bowet, D P. and Cesati, M., 2001:1-34).

3.3 Process Address Space

Each process runs in its private address space. When a process is running in User Mode it has its own private stack, data, and code areas. When operating in Kernel Mode, those are different.

Since the kernel is re-entrant, several different processes may be executed in turn, each with its own kernel control path. These paths have their own stack. But processes may also share address space. This is done automatically by the kernel to save memory. For instance, when two different users use the same editor, the program is only loaded into memory once. The data are not shared in this case, so it must not be confused with shared memory, which will be described later (Bowet and Cesati 2001:1-34).

3.4 The Soft Real-Time 2.6 Kernel

It is possible to categorize Real-Time operating systems into two camps; those which support Soft Real-Time responsiveness and those which support Hard Real-Time responsiveness. Real-Time responsiveness can be defined as "the ability of a system to respond to external or clock events within a bounded period of time."(Singh, 2004) The 2.6 kernel of Linux is regarded as a Soft Real-Time operating system, where determinism is not critical. That is, a fast response is desirable, but an occasional delay does not cause malfunction. This is the contrary to a Hard Real-Time operating system, such as a flight control system, where a deadline never may be missed.

Soft Real-Time responsiveness is a requirement to mobile phones. Even though there are requirements to multiprocessing, it is still a mobile phone and the phone specific services such as calls and messages will have to be prioritized before other applications and events. Before the 2.6 kernel release, special patches were necessary to achieve sufficient responsiveness. The improved responsiveness of the 2.6 kernel is mostly due to three significant improvements: a preemptible kernel, a new efficient scheduler, and enhanced synchronization. These improvements have contributed to make Linux an even better suited operating system for mobile phones.

3.4.1 The Pre-emptive 2.6 Kernel

Even though most UNIX kernels used to implement non-pre-emptive kernels as a solution to synchronization problems, the Linux 2.6 kernel implements pre-emption. In earlier releases of the Linux kernel, and like most general-purpose operating systems, the task scheduler was prohibited from running when a process were executing in a system call. The task would control the processor until the return of the system call, no matter how long that would take. Hence, the kernel could not interrupt a process to handle a phone call within an acceptable time limit. The 2.6 kernel is to some degree preemptive, meaning that a kernel task may be preempted with a low interruption latency to allow the execution of an important user application. The preemtion is triggered by the use of interruptions. A microprocessor typically has a limited number of interrupts, but an interrupt controller allows the multiplexing of interruptions over a single interrupt line. There also exist priorities among the interrupts. (Bowet, and Cesati, 2001)

This means that a process that is executing in Kernel Mode can be suspended and substituted by another process because it has higher priority. The operating system must be able to handle multiple applications and processes. For a mobile phone with soft Real-Time requirements such functionality is essential, as it must be able to handle important tasks such as an incoming phone call while the user is filming a video etc.

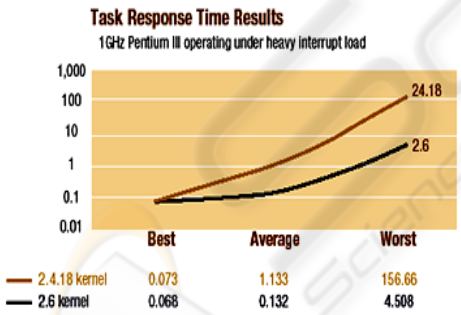

Figure 2: A comparison between the task response time of the 2.4.18 Linux kernel and the 2.6 kernel. (Singh, 2004).

Compared to a PC, the processing power is reduced, but the requirements to responsiveness are higher. The kernel code is laced with preemption points allowing the scheduler to run and possibly block a running process so as to schedule a higher priority process. Linux is still not a true Real-Time operating system, but it is certainly less jumpy than before and considerable faster than its predecessors, as seen in figure 2.

3.4.2 The New O(1) Scheduler

The 2.6 kernel has a totally new process scheduler that replaces the slow algorithms of earlier kernels. Earlier, the scheduler would have to look at each ready task and score its relative importance to decide which task to run next. The new scheduler no longer scans every task every time, but uses two queues. When a task is ready to run, it will be sorted and placed in a queue, called the *current queue*. The scheduler then chooses the most favourable one in this queue to run next, giving each process a specified time to occupy the processor. Opposite to earlier, this is done in a constant amount of time, and not relative to the number of processes. After its time in the processor expires, the process is placed in the other queue, called the *expired queue*. The process is then again placed according to its priority. When all the tasks in the current queue are done, the scheduler once again starts its simple algorithm of picking tasks from the expired queue, which now is called the current queue. This new scheduler works substantially faster than the previous scheduler, and it works just as fast with many tasks as with few (Deshpande, 2004).

3.4.3 Synchronization

By implementing a re-entrant kernel, one also introduces the need for synchronization among kernel control paths. One must ensure that while acting on a kernel data structure, no other kernel control path is allowed to act on the same data structure, even if the first one suspend the data structure. The data structure must be put back into a consistent state.

Let's say that we have one global variable V representing available items of some system resource. If a first kernel control path reads V, it sees that it is 1. Another kernel control path reads the same variable, and decreases it to 0. When A resumes its action, it has already read V and decreases it. As a result, the value of V is now -1. The two kernel control paths are using the same resource, which could result in serious errors.

When the outcome of a computation depends on how the processes are scheduled, the code will be incorrect and we have a *race condition.* Safe access to global variables is ensured by using *atomic operations*, which refers to combining the operations from two or more kernel control paths so they appear as one to the rest of the system. Any section of code that can not be entered by a process before another one has finished it is called a *critical region*.

The 2.6 kernel implements something that is referred to as *futex* – fast user-space mutexes. It is a new implementation of the mutex previously

implemented as system calls to check that only one task is using a shared resource at a time. This time-consuming system call to the kernel to see whether block or allow a thread to continue was often unwarranted and unnecessary. Futex checks user-space to see whether a blocking is necessary, and only issues the system call when blocking the thread is required. This saves time. The function also uses the scheduling priority to decide which thread is allowed to execute in case of a conflict (Singh, 2004), (Deshpande, 2004).

4 COMPUTER VERSUS MOBILE PHONE

Adapting Linux for mobile phones first requires a thorough study of the similarities and differences between the two hardware platforms, i.e. between the ordinary computer and the mobile phone. The most significant difference is usually the processor architecture, where x86 is the most common on regular PCs and ARM is the most common on mobile phones. The ARM architecture is generally better on performance, power, and integration for mobile phones. But the choice of a non-x86 architecture, which Linux was originally built for, first of all results in necessary porting of some low-level drivers.

4.1 Necessary Subsystems

There are certain subsystems that are required for Linux to work on all systems. Generally the kernel can be split into these following subsystems:

- Hardware Abstraction Layer
- Memory Manager
- Scheduler
- File System
- IO subsystem
- Networking subsystem

The scheduler has already been discussed, but the Hardware Abstraction Layer, Memory Manager, File Systems, and IO subsystem will be described briefly.

4.1.1 Hardware Abstraction Layer

A Hardware Abstraction Layer (HAL) is a more concrete name of the underlying low-level interfaces that are supposed to give higher level languages the ability to communicate with lower level components, such as directly with hardware.

Its function is to hide differences in hardware from most of the operating system kernel, so that most of the kernel-mode code does not need to be changed to run on systems with different hardware. The HAL supports these hardware components, which are usual on both platforms:

- Processor, cache, and MMU
- Setting up the memory map
- Exception and interrupt handling support
- DMA
- Timers
- System Console
- Bus Management
- Power Management

4.1.2 Memory Manager

The task of the memory manager is to control memory access to the hardware memory resources. In Linux the memory manager implements a logical layer for as the Memory Manager Unit being able to provide virtual memory to kernel subsystems such as drivers, file systems, and networking stack. But also it provides virtual memory to user applications. The advantages of virtual memory can be summarized with these points:

- Several processes can be executed concurrently
- It is possible to run applications whose memory need are larger than the available physical memory.
- Processes can execute a program whose code is only partially loaded in the memory.
- Each process is allowed to access a subset of the available physical memory.
- Processes can share a single memory image of a library or a program.
- Programs can be relocatable – that is, they can be placed anywhere in physical memory.
- Programmers can write machine-independent code, since they do not need to be concerned about physical memory allocation.

All this is solved by the use of a virtual address space, which is representation of physical locations located by the MMU and the kernel. The virtual address space is also referred to as a linear address space. The virtual addresses are divided by the kernel into *page frames* with a size of 4 or 8 KB, which result in that a request for contiguous virtual address space can be satisfied by allocating a group of page frames that do not necessarily have contiguous physical addresses. All the pages are

accessible by the kernel, but only some of them get used by the kernel. The paging process only involves the applications, which get pulled into main memory on request. By using virtual addresses a running process will not be able to corrupt neither another process's nor the operating system's memory. This means that any pointer corruptions within a process are localized to the process itself, and will not bring down the system. This is important for system reliability.

On the other hand, the 2.6 kernel allows the system to be built without a virtual memory system. This is often to meet real-time requirements. Slow handling of *page faults* can ruin responsiveness. A page fault is when a demanded page is not in physical memory and an interruption has to be raised. Of course, a no virtual memory solution removes the advantages previously mentioned, and it becomes the software designer's responsibility to ensure there will always be enough real memory available to meet the applications demands. The issue of whether to use virtual memory or not is left to the programmer.

4.1.3 File Systems

There are many file systems that can run on Linux. Ext2, CRAMFS, ROMFS, RAMFS, NFS, DEVFS, and JFFS2 are often used on embedded systems. As a general point, the hardware memory/storage technology used on the device may set limitations to the choice of file systems. The kernel supports them all through a concept called the Virtual File System (VFS). VFS handles all the system calls related to the file systems. The file systems must translate their physical organization into a *common file model* which can represent all the supported file systems. In that way, to interact with the different file systems the kernel (i.e. the VFS) has only one interface to relate to.

It is necessary for every Linux system to have a *root file system.* This is the master file system which gets mounted during start-up. In Linux, everything is a file, even the directories and the I/O devices. UNIX systems also implement a *current working directory* for every process.

The PROCFS or /proc file system, is a special file system as it is a pseudo file system that resides in memory and is created every time the system is rebooted. The /proc directory reveals important data on the running processes and the state of the system itself. It is readable by the owner of the processes and the root. This openness and access to devices is very useful for programming.

4.1.4 I/O Subsystem

The most difficult part of porting Linux to a mobile phone is not the main configuration of the kernel, but the programming of the low-level interfaces which are special for this kind of embedded devices. For the programmer, the IO subsystem provides a simple and uniform interface to onboard devices. Special or not, on a mobile phone I/O devices will typically involve devices such as keypad, camera, Bluetooth, LCD screen, and non-volatile storage in some form, but also the drivers for the GSM/GPRS Digital Baseband Subsystem related functions. Those are often provided by the board manufacturers, such as Texas Instruments, or by the operating system vendors, such as MontaVista. These must be custom made to the hardware architecture and this is a process that may be troublesome (Raghavan, Lad and Neelakandan, 2005).

The I/O subsystem supports three kinds of devices:

- Character devices for supporting sequential devices
- Block devices for supporting randomly accessible devices. Block devices are essential for implementing file systems.
- Network devices that support a variety of link layer devices.

4.2 The MTD Subsystem

In Linux, memory technology devices are all kinds of memory devices: RAM, ROM, and Flash in different technological solutions. The Memory Technology Devices (MTD) subsystem is a module of the Linux kernel. The MTD subsystem intends to provide a uniform and unified access to memory devices for the VFS. In that way it avoids having different tools for different technologies. The MTD subsystem consists of low-level chip drivers and high-level interfaces called *MTD user modules.* The user modules are software modules in the kernel that enables access to the chip drivers through recognizable interfaces and abstractions, which in turn are provided to the higher levels of the kernel and in some cases to user space.

The typical operations the MTD subsystem has to carry out is erase, read, write, and sync. The system works in a manner where the chip drivers register sets of predefines call-backs and properties with the MTD subsystem. The call-backs and properties are defined in an mtd_info structure, which is provided to the add_mtd_device() function. These call-backs are then called through this function.

There is no "standard" physical address location for the MTD devices, and therefore they need a customized *mapping driver*. In addition, some systems and development boards have known MTD device configurations. The kernel therefore contains a number of specific drivers for these systems. The drivers are found in the *drivers/mtd/maps/* directory of the kernel sources.

On a mobile phone a combination of the CRAMFS and the JFFS2 file systems is a well known working combination. CRAMFS for the non changing boot image which is extended into RAM on start-up and JFFS2 for the writable persistent file system (Yaghmour, 2003).

5 SERVICE DEVELOPMENT FOR LINUX MOBILE

5.1 Trolltech

Trolltech is a Norwegian company with two product lines; Qt (pronounced cute) and Qtopia. They were one of the first companies in the world to use a dual licensing model. The business model allows software companies to provide their products for two distinct uses - both commercial and open source software development. This type of licensing is based on Quid Pro Quo – Something for something. Either the customers of Trolltech may release their software under the GNU Public License, GPL, or they may purchase the appropriate number of commercial licenses from Trolltech and release the software under a license of choice.

Trolltech means that this strategy will make them able to provide the best cross-platform development tools in the world. The commercial license makes the money, and the open source licenses ensure quality and stability of the products delivered by Trolltech.

5.1.1 Qtopia Core

Qt is a cross-platform application development platform. Qt includes the Qt Class Libraries, which is a collection of over 400 C++ classes. Further it includes Qt Designer for rapid GUI and forms development, and other tools as well. "The Qt class libraries aim to provide a near-complete set of cross-platform application infrastructure classes for all types of C++ applications." Qtopia Core is the application framework for single-application devices powered by embedded Linux. It provides the same API and tools as other versions of Qt, but it also includes classes and tools to control an embedded environment.

5.1.2 Qtopia Phone Edition and Greenphone

Qtopia Phone Edition is the phone intended version of Qtopia Core. It is an application platform and user interface for Linux-based mobile phones. Trolltech claims that Qtopia Phone Edition is the de-facto standard application development platform and user interface for Linux-based mobile phones.

Also, Trolltech have developed a dual licensed hardware component of the Greenphone SDK. This SDK provides a complete environment for developing and modifying application software for Qtopia Phone Edition on the Greenphone (Trolltech: Code less – Create More (n.d.).).

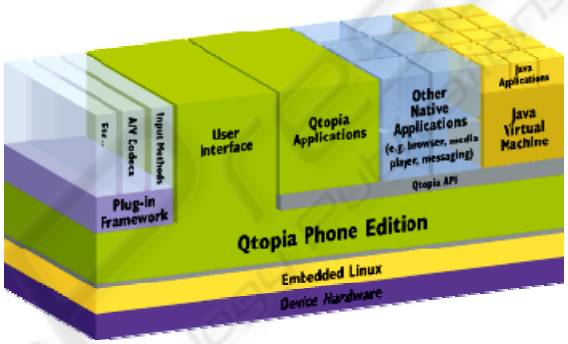

Figure 3: Qtopia Phone Edition diagram (Trolltech: Code less – Create More (n.d.).).

5.2 MontaVista

MontaVista offers an optimized Linux operating system and development environment for both wireless handsets and mobile phones with requirements for power management, hard real time performance, fast start-up, and small footprint, called *Mobilinux*.

5.2.1 Mobilinux

The current version of Mobilinux is based on the Linux 2.6 kernel. It uses the reduced C library uClibc, DirectFB on top of the Linux Framebuffer Device, and SquashFS as the compressed read-only file system to be able to provide a reduced footprint. The Linux framebuffer, *fbdev*, is a graphic hardware-independent abstraction layer to show graphics on a console without relying on system-specific libraries. Further, Mobilinux has improved Real-Time support and implements a fully preemptible kernel through MontaVista's enhancements. (Montavista Software – Platform to innovate (n.d.).).

5.3 The OpenMoko Strategy

The OpenMoko camp, with its NEO1973, has taken on another business strategy than Trolltech and MontaVista. They favour a complete open strategy, as any regular PC intended open source Linux distribution.

The software of the mobile phone is based on the 2.6.20 kernel. It runs on a Samsung board with 64 MB NAND flash and 128 MB RAM. At the moment it has GSM/GPRS, USB, and Bluetooth support. It is equipped with a touch screen, and only two buttons for power and for auxiliary devices. It uses U-boot as boot loader.

Further this open source project provides a development framework, namely the Open Mobile Communications Platform (OpenMoko). The project intends to provide a completely open standard framework for developing mobile phone applications, much like Trolltech. The phone comes shipped with a package manager to be able to take full advantage of the all ready large Linux application community.

The key of OpenMoko's business strategy is to trigger the open source community first. With them they will be able to ensure increased revenues for both carriers and handset developers. The idea is to let the users control their own environment of applications. The handset manufacturers can get a reduced time to market and the carriers will experience a large increase in data traffic. Applications may form the next generation of multi billion industry similar to that of ringing tones. It's a win-win situation for all three parts; users, carriers, and handset manufacturers (OpenMoko: The World's First Integrated Open Source Mobile Communications Platform (n.d.).).

6 CONCLUSION

What have been presented in this paper are the fundamental mechanisms of Linux, with a focus on the latest major kernel release. The most important, main components that must be included in an embedded environment are discussed, and the paper clearly describes how a standard Linux kernel may be adapted to fit the mobile device.

Further, some of the important differences between the hardware platform of the computer and the hardware platform of a typical mobile phone are shown.

In addition, the paper has elaborated the challenges and opportunities of employing Linux as an enabler for advanced services on mobile phones.

REFERENCES

The Diffusion Group, 2006, February 7. *Windows & Linux to Displace Symbian as Dominant Force in Advanced Mobile Operating Systems.* http://www.tdgresearch.com/press066.htm

Blandford, R., 2006, February 8. *TDG claim Symbian will be behind Linux and Microsoft by 2010.* Retrieved March 28, 2007, from All About Symbian Web site: http://www.allaboutsymbian.com/news/item/TDG_claim_Symbian_will_be_behind_Linux_and_Microsoft_by_2010.php

Cheap, hackable Linux smartphone due soon, 2006, November 7. http://www.linuxdevices.com/news/NS2986976174.html

OpenMoko: The World's First Integrated Open Source Mobile Communications Platform, (n.d.). Retrieved March 28, 2007, from www.openmoko.org

Purdy, J. G., January, 2007. *Mobile Linux: Why it will become the dominant mobile OS.* http://www.fiercewireless.com/story/feature-mobile-linux-why-it-will-become-the-dominant-mobile-os/2007-01-03

Benchmark clocks OMAP2420 graphics on Linux, Symbian. February 2, 2006. http://linuxdevices.com/news/NS6023095418.html

Yaghmour, K., 2003. *Building Embedded Linux Systems,* Sebastopol, CA: O'Reilley

Embedded Linux Graphics Quick Reference Guide, (n.d.). http://linuxdevices.com/news/NS6023095418.html

Raghavan P., Lad A. and Neelakandan S., 2005. *Embedded Linux system design and development*, Boca Raton, FL: Auerbach Publications

Bowet, D P. and Cesati, M., 2001, *Understanding the Linux Kernel,* 1st edition, Bejing: O'Reilly.

Singh, I. M., 2004, *Embedded Linux: The 2.6 kernel is ideal for specialized devices of all sizes*, http://www.lynuxworks.com/corporate/news/2004/linux-kernel-2.6.php

Deshpande A. R., 2004, March 4, Linux *Kernel 2.6: the Future of Embedded Computing, Part I.* Retrieved March 28, 2007, from the Linux Journal Web site: http://www.linuxjournal.com/article/7477

The OMAP 730 Digital Baseband (n.d.). Retrieved March 28, 2007, from http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12003&contentId=4676

Trolltech: Code less – Create More. (n.d.). Retrieved April 3, 2007, from www.trolltech.com

Montavista Software – Platform to innovate. (n.d.). Retrieved April 3, 2007, from www.mvista.com