Copyright

by

Anish Chandrakant Trivedi

2010

The Report Committee for Anish Chandrakant Trivedi
Certifies that this is the approved version of the following report:

An Examination of Linux and Windows CE Embedded Operating Systems

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: ______________________________
Christine Julien

____
Paul Kline

An Examination of Linux and Windows CE Embedded Operating Systems

by

Anish Chandrakant Trivedi, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 2010

Dedication

To my wife for her unwavering patience and support; to our son for being the little bundle of excitement that he is; and to my parents, for everything.

Acknowledgements

Special thanks to my advisor, Dr. Christine Julien, for her guidance to significantly improve the quality of the report. Also, thanks to Dr. Paul Kline for his help and support for this report while I worked for him full time at Freescale.

July 26, 2010

Abstract

An Examination of Linux and Windows CE Embedded Operating Systems

Anish Chandrakant Trivedi, M.S.E

The University of Texas at Austin, 2010

Supervisor: Christine Julien

The software that operates mobile and embedded devices, the embedded operating system, has evolved to adapt from the traditional desktop environment, where processing horsepower and energy supply are abundant, to the challenging resource-starved embedded environment. The embedded environment presents the software with some difficult constraints when compared to the typical desktop environment: slower hardware, smaller memory size, and a limited battery life. Different embedded OSs tackle these constraints in different ways. We survey two of the more popular embedded OSs: Linux and Windows CE. To reveal their strengths and weaknesses, we examine and compare each of the OS's process management and scheduler, interrupt handling, memory management, synchronization mechanisms and interprocess communication, and power management.

Table of Contents

List of Tables viii

List of Figures ix

Chapter 1: *Introduction* 1

Chapter 2: *Operating System Overview* 3

Chapter 3: *Overview of Linux and Windows CE* 6

- Linux 2.6 6
- Windows embedded ce 6.0 8

Chapter 4: *Process Management and Scheduler* 11

- Discussion 15

Chapter 5: *Interrupt Handling* 18

- Discussion 20

Chapter 6: *Memory Management* 23

- Discussion 26

Chapter 7: *Synchronization and Interprocess Communication* 29

- Discussion 32

Chapter 8: *Power Management* 34

- Discussion 36

Chapter 9: *Conclusions* 38

References 41

Vita… 42

List of Tables

Table 1: Process Management and Scheduler Comparison. 15

Table 2: Interrupt Handling Comparison 20

Table 3: Memory Management Comparison 26

Table 4: Synchronization and Interprocess Communication Comparison...... 32

Table 5: Power Management Comparison 36

List of Figures

Figure 1: Linux 2.6 Architecture………………………………………………… 6

Figure 2: Windows CE 6.0 Architecture…………………………………………..9

Figure 3: Windows CE 6.0 Interrupt Handling…………………………………. 20

Chapter 1: *Introduction*

Embedded systems, generally speaking, represent a category of small to miniature computing devices with an ever-increasing number of applications, though even some large systems are characterized as an embedded system if they are part of a larger system: e.g., an in-car entertainment system. These diminutive but smart systems offer invaluable services in personal computing, with devices that offer on-the-go connectivity and multimedia experience, home automation, vehicular control and entertainment, industrial automation, and many other services. As a descendant of traditional computing systems, embedded systems are similarly designed around a microprocessor with memory and some peripherals that may include non-volatile storage and some Input/Output (I/O) devices. Common examples of I/O devices include a communication radio, serial port, audio, or even a keypad and display for the high end systems like a mobile phone. Unlike traditional desktop devices, however, the amount of each resource such as processor speed or memory size is much more limited. Owing to their small size, embedded systems find most use in mobile applications; therefore, the system is typically powered from a battery with a limited life.

Advancements in manufacturing technology, such as the System on Chip (SOC) -- which combines the processor with many specialized peripheral controllers into one package -- make it possible to add more of the resources to the embedded system while reducing the overall size and cost. The increase in the hardware capabilities has paved the way for an exponential growth of applications for embedded systems. Naturally, this had led to an equally explosive growth in the market for the software that runs the embedded system -- the operating system (OS). The OS plays a primary role in the overall success of the applications to be run on the system. While there are many competing embedded

OSs, Linux® and Windows® CE are two of the leaders in this market, as evidenced by the survey in [1]. In this paper, we examine how the two OSs handle the common challenges presented to an embedded operating system to reveal their similarities and differences.

The operating system software is an essential component of the embedded software stack. As the layer closest to the hardware, the OS's primary responsibilities lie in the area of hardware management to best suit the needs of the user applications. Among common tasks for operating systems are process management and scheduling, interrupt handling, memory management, synchronization and interprocess communication, and a particularly important one for an embedded OS: power management. Additionally, the OS provides an interface that exposes the hardware resources to the user applications. Resource allocation, therefore, is an inherent task of an operating system. The limited amount of resources in an embedded system presents some interesting challenges for the OS.

The two different 32-bit embedded operating systems under consideration in this paper are Linux 2.6, and Windows Embedded CE 6.0. We compare how each OS handles the common tasks enumerated above to reveal the different approaches taken to meet the challenges of an embedded environment. The rest of the paper is organized as follows: The next chapter introduces the concept of an operating system and presents a quick overview. Chapter 3 presents brief backgrounds behind both Linux and Windows CE, and covers the architecture of each from a high level perspective. Chapters 4, 5, 6, 7, and 8 examine the process management and scheduler, interrupt handling, memory management, synchronization and interprocess communication, and power management aspects of each OS. The paper is concluded in chapter 9 with some observations about the strengths and weaknesses of both operating systems.

Chapter 2: *Operating System Overview*

Before delving into the specifics of Linux and Windows CE, a basic overview of an operating system is presented in this chapter. An operating system is the set of software that controls the hardware and enables user applications to execute. Without a central piece of software to manage the hardware, supporting multiple applications on the system becomes virtually impossible. How conveniently applications can execute and how efficiently the hardware resources are utilized depends on the design of the OS [2]. Naturally, for an embedded OS, efficient resource utilization is a point of emphasis.

Given the considerable complexity of hardware management as well as application support, an OS typically comprises many components. The core piece of the OS is the kernel. The kernel invokes device drivers to perform I/O on a particular peripheral such as keyboard or display, as needed by applications. The kernel performs various other essential tasks such as process management, interrupt handling, interprocess communication, file system management, power management, resource allocation and accounting, error detection, as well as protection. [2]

A process refers to a program or execution unit. Process management involves loading an application or user space process into memory and executing it, while enforcing a certain scheduling policy. The scheduling policy is a key factor in the performance of applications in the system. Events in a computer system, whether originating in the hardware or software, can cause interrupt signals to fire, serving as a fundamental communication method. For example, a press of a key on a keyboard can indicate an interrupt to the processor, which suspends the current process and executes the interrupt handler for the key press. While processing the interrupt, whether another interrupt event is handled or not depends upon the interrupt handling policy of the OS.

Different strategies can be employed to enable different system behavior based on requirements for the system.

Whenever an application is to be executed for the first time, the OS is required to allocate memory for the program. Memory is limited in an embedded system, and it must be shared with other applications, so this task must be performed judiciously. The higher level responsibility of resource allocation and accounting encompasses the task of memory management. Accounting involves tracking the amount of the resource available and allocated. An I/O device is also treated as a resource, so the OS needs to track which applications have opened which devices. Files are a basic form of data storage that most applications use; therefore, file system management is a key function of an OS. This involves creation, deletion, reading, and writing of files by any number of applications. The OS must also be capable of providing a mechanism for processes to share information with each other, also known as interprocess communication. Given concurrent execution of processes, access to shared resources is managed via synchronization. [2]

Errors are not an uncommon occurrence in most computer systems, and embedded systems are no exceptions. Error detection is also the responsibility of the OS. Whether originating in the software or hardware, errors must be handled properly so that normal execution can continue after the occurrence of an error. The degree of error handling offered by the OS is, once again, a design decision that is driven by the requirements of the applications. Furthermore, given a multi-application environment, the OS may also be required to implement some degree of protection or security. A basic example of protection is an OS restricting access for each application to its assigned memory space. [2]

In order to implement all the services expected from it, an OS typically includes, besides the core kernel, several other programs, utilities, and libraries, which may include, for example, the C library for applications. Additionally, an OS may include middleware such as the TCP/IP stack, an essential service for any application that needs to communicate over the Internet. [2]

Chapter 3: *Overview of Linux and Windows CE*

The two operating systems examined in this paper have emerged from two very different histories. A high level overview of each is provided below.

LINUX 2.6

Linux 2.6 is the most recent stable kernel version of the Linux operating system, which was originally developed by Linus Torvalds, with assistance from several other people in the early 1990s. Since then, the online community of Linux developers (http://www.kernel.org) has continued to collaborate to evolve the OS. Although the internals of the Linux OS were developed from scratch, the APIs are very much like UNIX. Furthermore, many UNIX concepts are designed into Linux, creating a simple yet refined OS. The Linux OS is targeted to run on desktops, servers, and -- as relevant to this paper -- embedded devices. Figure 1, which was borrowed from [3], depicts the architecture of the Linux OS.

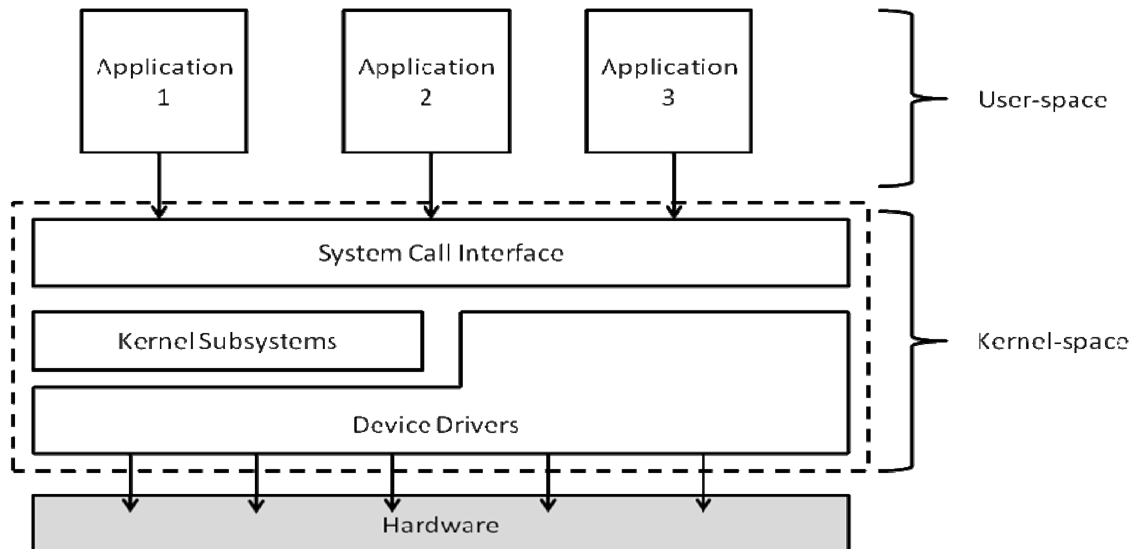

Figure 1: Linux 2.6 Architecture.

In Linux, the software system is split into two categories – user space and kernel space. Kernel space is compiled into one monolithic binary file consisting of device drivers that control the hardware and the kernel with subsystems that include code for important functions such as process management, memory management, file system management, and networking. Besides the device drivers, which control a specific peripheral on the hardware, the kernel space also contains hardware specific code, or the machine layer, that facilitates the portability of the Linux OS to different architectures, which is one of the reasons why the OS can run on a wide range of hardware – a small embedded device to a powerful server [3]. This code manages the processor, memory subsystem, and includes a device tree that contains all the peripherals on a particular hardware platform. The device tree is used to match up each peripheral with its driver when the driver registers with the kernel during initialization. Linux offers some flexibility in loading modules, which may contain one or more drivers. Modules can be built into the kernel so that the modules will load upon OS boot, or a module can be loadable, which means that it can be dynamically loaded and unloaded after the OS has completed booting. Loadable modules can also be part of the kernel space.

User space consists of user applications as well as any OS code that does not require execution in the privileged kernel mode. One example of such a piece of code is the GNU C library. One of the key requirements of an OS is to facilitate application execution on the hardware. Applications can request resources or services from the Linux OS through the system call interface. For example, when an application requires access to a storage disk, it can make a system call to the kernel space, which carries out the read or write on the application's behalf. Once the kernel space code completes the disk access via the device driver in privileged mode, the execution is returned to the application in

user space. Since kernel space code is trusted, it is ensured that precious system resources are not used in an illegal or malignant manner.

The memory footprint of a Linux kernel varies depending upon the configuration selected by the developer. Many modules can be optionally compiled into the image, allowing Linux to function in a system with 4 MB or 4 GB of storage space – a demonstration of its flexibility. [2]

WINDOWS EMBEDDED CE 6.0

Microsoft has long enjoyed a dominant share in the desktop computing market with the Windows operating system. In the mid 1990s, they introduced the first version of Windows CE that was designed specifically for embedded devices. While there are many different flavors of embedded Windows operating systems today, in this paper, we focus on Windows Embedded CE 6.0 OS, which is the flagship embedded OS from Microsoft. The familiar Windows interface not only appeals to users of embedded systems but also enables desktop application developers to more easily write applications for the embedded world due to the fact that application support libraries in Windows CE are similar to the desktop version of Windows. Over the years, many features that users expect from desktop Windows systems have been made available in CE. Windows CE also offers the advantage of a large commercial ecosystem created by Microsoft. Figure 2, which was reproduced from [4], depicts the Windows Embedded CE 6.0 architecture.

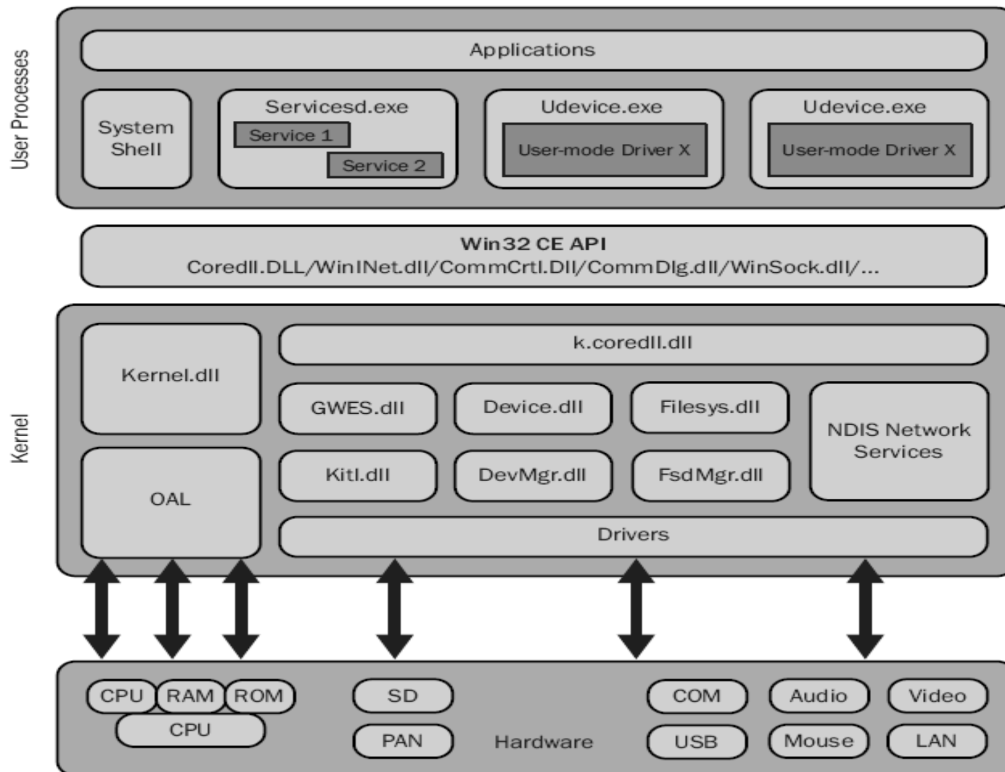

Figure 2: Windows CE 6.0 Architecture.

Similar to Linux, there are two categories of software – kernel mode and user mode. The core kernel, Nk.exe, is an executable process into which the kernel mode Dynamic Link Libraries (DLL) are loaded. The kernel.dll and Original Equipment Manufacturer (OEM) Adaptation Layer (OAL) are linked together to form the Nk.exe process. The OAL contains hardware architecture specific code that facilitates portability of the OS to different hardware platforms. All device drivers, which control a specific peripheral, are kernel mode DLLs. Additionally, numerous other functionalities, such as file system management (filesys.dll and fsdmgr.dll), graphics, and user interface support (GWES.dll), driver and resource management (device.dll and devmgr.dll), as well as debugging (kitl.dll) and networking support, are part of kernel space code.

The user mode software consists of all the user applications, the shell, user mode services, as well as user mode drivers that run in udevice.exe process. All executables in the system, including user applications, must link to the system API library – coredll.dll, which provides many functionalities to the process in the areas of graphics, time, synchronization, and system calls, to name a few. Coredll performs a system call into kernel mode from user mode that serves as a mechanism to access some kernel functionality from the applications. Calls to coredll in a kernel mode process are routed at run time to k.coredll.dll, which is the kernel mode version of the library [4]. Servicesd.exe is responsible for loading such services as HTTP and FTP. The Win32 CE API is the embedded equivalent of the desktop Win32 APIs used by applications. The APIs enable a rich set of features made available for application development in Windows CE that are touted by Microsoft as a major advantage of CE. Similar to Linux, the Windows CE developer has the power to optionally include or exclude many components during compile time. This flexibility results in memory footprints that can range from kilobytes to megabytes. [4]

Chapter 4: *Process Management and Scheduler*

To load and run a process, which is called an application if it is user space, is one of the primary reasons of existence of the operating system. In a multi-process environment, in order to prevent one process from monopolizing cycle time on the processor, the OS may choose to suspend the currently running process in favor of another process. Or the running process may make a system call that blocks execution until the request is serviced by the OS. In any case, when switching execution to a new process, the current process's context -- such as the register values and currently held resources -- is saved, and the new process's state is restored based on data structures that the OS maintains for each process. This scenario is referred to as a context switch. Process management involves the creation, maintenance, and deletion (when the process exits) of processes and related data structures. The scheduler is the part of the OS that decides which process gets to execute based on scheduling policy, which can greatly affect overall system performance [2].

In Linux 2.6, when a process is to be created, an existing process calls the *fork* command, which duplicates the calling process to create a child process. Thereafter, if a new piece of code is to be executed in the child process, the *exec* command is run, which creates a new address space with the new program specified with the *exec* command. In contrast, each Windows Embedded CE 6.0 process contains the finer grained thread as the basic unit of execution. Each process has one designated primary thread and can have other threads that are similar to child processes in Linux. Therefore, unlike Linux, processes in Windows CE do not share address spaces, threads within a process do. Windows CE supports an even finer grained unit of execution, called a fiber, which can be created within a thread. Because the scheduling policy of fibers within a thread can be

controlled by the developer, the use of fibers can empower the developer to define a custom scheduling policy within a given thread. Such control over scheduling between a parent and child processes is not available under Linux.

When a Linux process calls *fork*, the kernel creates the child process's data structure, the process descriptor. This data structure is referred to as the thread context in Windows CE. In Linux, the child process's memory pages are not created by the *fork* command, but they are the same as the parent's until a page is written to. In effect, the child process is like a Windows CE thread within the parent process because it is sharing the same program and memory as the parent process. Only when *exec* is called does the program change and a new address space created. This delays or prevents needless copying of the parent's address space and enables quick execution of the child process with very little overhead [3]. In Windows CE, process creation is not as fast because the entire address space has to be created and assigned at the time of creation, but thread creation is fast.

In Linux, the process state identifies whether a process is executing or executable, blocked waiting for a signal, blocked for a definite period of time, stopped, or waiting to be terminated [3]. Similarly, Windows CE uses the following thread states: running, runnable but not currently executing, blocked waiting for a shared resource, suspended until resume is explicitly called, and sleeping [5]. These states are used by the scheduler of both OSs when switching or deciding to switch execution between processes for Linux, or threads for Windows CE.

The Linux OS uses preemptive multitasking, where process execution can be stopped by the scheduler based on the scheduling policy. Two primary metrics are used by the standard Linux scheduler to make scheduling decisions: the process priority and timeslice. A process with a higher priority is allowed to execute for a specific amount of

time, the process's timeslice, before another process with a lower priority. Users can set the priority of a process, via system calls, just as the kernel can [3]. Windows CE uses preemptive multitasking, as well, using timeslices and priorities when decision points are reached in scheduling [4].

Where they are quite different, however, is that Linux, in addition to preemptive multitasking, employs dynamic priority based scheduling, where heuristics about the process's behavior are used to raise or lower the priority and, thus, the timeslice increased or decreased. Windows CE does not re-calculate priorities like Linux does. Timeslices are fixed at 100 ms unless modified by the user or the device manufacturer. Dynamic priority recalculation is justified based on the following theory: The more a process blocks, the higher probability that it relies on I/O; therefore, its response to any event must be fast, requiring a higher priority. Conversely, the less a process blocks, the higher chance that it is a background process that is less interactive; therefore, its priority is lowered. The timeslice is also changed dynamically based on the priority: more interactive (higher in priority) processes are allotted a greater time slice (up to 200 ms). Every time a process runs, its timeslice is reduced by the amount of time it ran. Since an interactive program will spend most of its time blocked, it will retain its timeslice that will be needed when reacting to events. Once the timeslice reaches 0, the process is no longer active but it is considered expired, and the next process in the active queue with the highest priority is selected for execution. The timeslice of an expired process is recalculated, based on heuristics, before it is moved to the expired array. Expired processes cannot run again as long as there are processes available in the active queue. This policy tries to implement some level of fairness by ensuring that lower priority processes will always get a chance to execute since higher priority processes will eventually exhaust their timeslices. When all active processes have exhausted their

timeslices, a simple swap of the active and expired arrays is performed that is very efficient compared to the alternative of calculating timeslices for all expired processes at one time. This design ensures a constant-time scheduling task regardless of the number of processes in the system [3].

Without dynamic priority recalculation, it is possible for the highest priority thread in Windows CE to continue execution for as many timeslices as needed, provided it does not block, over lower priority threads, which may result in a less responsive system. Nevertheless, dynamic priority recalculation does add scheduling overhead in comparison with Windows CE. The lone situation when Windows CE does reassign priorities dynamically is when a lower priority thread is holding a resource that causes a higher priority thread to block. In this case, the scheduler inverts the priorities of the 2 threads. With priority inversion, the thread holding the resource is allowed to execute, which results in a release of the resource eventually, at which point the priorities are set back to the original values. The thread waiting for the resource is then able execute and obtain the resource. [4]

Many embedded applications require real time capabilities from the OS. Real time means meeting certain timing requirements for particular tasks. Here is an example of a real time requirement: in an automobile, a computer usually controls the deployment of air bags when a collision is detected. The software running on the processor must start the air bag deployment mechanism within fractions of a second after the collision. Owing partially to the scheduling overhead described above, Linux can only exhibit soft real time capabilities, which means that timing deadlines can be met most of the time, but it is not guaranteed. At a minimum, however, separate queues for real time processes in Linux ensure that they are guaranteed to run over non-real time processes [3]. Windows CE, on the other hand, is a hard real time OS, which means that timing deadlines are guaranteed

to be met. The scheduler is written such that it can be preempted by an event requiring real time processing. However, the real time support is dependent upon avoidance of priority inversion and long unbounded operations performed by interrupt handlers that may create long latencies, which would result in failure to meet the hard real time deadline [6].

DISCUSSION

Process Management or Scheduler Property	**Linux 2.6**	**Windows Embedded CE 6.0**
Basic unit of execution	Process	Thread
Address space created at time of process creation	No	Yes
Ability to schedule execution of code within a unit of execution	No	Yes, via fibers
Preemptive multitasking support	Yes	Yes
Dynamic priority based scheduling	Yes	No
High priority process/thread can lock out lower priority processes/threads	No	Yes
Hard real time capable scheduler	No	Yes

Table 1: Process Management and Scheduler Comparison.

Table 1 summarizes the key process management and scheduler characteristics examined in this chapter for both Linux and Windows CE. While Linux uses a process as the basic unit of execution, Windows CE designates a thread as a basic unit of execution. Linux does not create address space for the newly created process until the shared memory pages are modified by either the parent or the child process. Windows CE, on the other hand, does create the process address space at the time of creation, which results

in significant up-front overhead when compared to Linux. Accounting for a smaller unit of execution, thread, within a process also adds to the Windows CE process creation time. Linux, therefore, is better suited for applications that require newly spawned processes that quickly start executing.

While Linux and Windows CE both support preemptive multitasking, only Linux supports dynamic priority based scheduling. In that regard, Linux increases the priority of I/O bound processes and decreases the priority of processor bound processes since they do not block as much and, consequently, may not allow an I/O bound process, which has to respond to events much faster, to execute as much as it needs to. It is a fine tuning of the overall system performance on the fly to alleviate any sluggishness perceived by the user. Windows CE, on the contrary, relies on the developer to select the proper thread priority and fine tune the overall system performance. From another perspective, the developer created priorities in Windows CE are not mutated by the scheduler, empowering her to customize the system performance. It is difficult, however, to predict the behavior of processes that require user I/O as opposed to processes that can run in the background. Therefore, developers of an embedded system can better engineer a system for performance by using Windows CE rather than Linux, but only for applications that do not require a lot of user interactivity. The fact that the Windows CE scheduler is simple and quick helps it to qualify as a hard real time OS, whereas Linux does not qualify due to the fact that its scheduler collects some metrics on all processes.

From a scheduler fairness perspective, however, Linux has the advantage because the scheduler ensures that lower priority processes will eventually get a chance to execute, when the higher priority process is moved into the expired array after its timeslice is exhausted. In Windows CE, when the timeslice of the higher priority process is exhausted, it will be selected for execution again over a lower priority thread until it

completes or blocks, It is easy to imagine that if a higher priority thread in Windows CE does not block much and does not complete for a long time, lower priority threads will not be able to make any progress. Therefore, if a high degree of multitasking is required in an embedded system, Linux is a better choice.

Chapter 5: *Interrupt Handling*

Another of the operating system's primary responsibilities is to manage the hardware on which the applications are to be run. The hardware periphery surrounding the processor typically runs at a much slower rate than the processor itself. Therefore, it is imperative that the processor be free to do some other useful work, such as to run a different application, while a particular application is blocked waiting for a peripheral (e.g., a key press). When the peripheral wants to indicate an event to the processor, it enables a specific hardware interrupt line. Based on the interrupt line, the kernel can suspend the currently executing process, and service the interrupt in an interrupt handler that is part of the device driver. Upon returning from the interrupt handler, the kernel may resume the originally executing code or schedule a new piece of code for execution. The longer an interrupt handler takes to process the interrupt, the more delay is induced into the previously executing process. It can lead to a degradation of performance in a system; therefore, it is highly desirable for interrupt handlers to return as quickly as possible. [2]

In Linux 2.6, interrupt handlers for each peripheral are supplied as part of the device driver for that peripheral. When the driver is loaded, it registers the interrupt handler function with the kernel identifying the hardware interrupt number, or IRQ, with which the handler is associated. In Windows Embedded CE 6.0, each IRQ is mapped to a software interrupt (SYSINTR). The device driver registers an Interrupt Service Thread (IST) against the SYSINTR. Upon receiving an interrupt, each OS runs an architecture-specific function, known as *do_IRQ* in Linux and the Interrupt Service Routine (ISR) in Windows CE. This is where the similarities in the two OSs interrupt handling scheme end. In Linux, *do_IRQ* identifies the interrupt handlers registered against the IRQ, and disables all interrupts if the handler has registered as a fast interrupt handler. Disabling all

interrupts for the duration of the interrupt handler adversely affects the real time capabilities of Linux, which cannot meet hard real time deadlines. Interrupt handlers, therefore, are required to be simple and fast, performing only the most essential tasks, such as moving received data from peripheral to memory, and leaving non-essential tasks, such as processing received data, for a later time. Postponing non-essential tasks ensures that the task is preemptible by other higher priority tasks [3].

Windows CE, in contrast, is a hard real time OS. Initially, while the OS identifies which ISR to run[1], all interrupts are disabled. When the ISR for the particular IRQ is executing, all higher priority interrupts are enabled, allowing preemption of the current ISR. The ISR finds the SYSINTR mapped to the IRQ and sets the event on which the device driver's IST is blocked. At the end of the ISR, all previously enabled interrupts, except the one being handled, are re-enabled. Interestingly, the Windows CE scheduler has to get involved to run the IST since an event has been set by the ISR. If the device driver developer is not careful to assign a high enough priority to the IST, it may not be selected by the scheduler to run next, causing a delay in servicing the peripheral that has interrupted [4]. It is in sharp contrast to the fast interrupt handling protocol in Linux, which blocks other higher priority interrupts in order to process the current one. The more complex Windows CE interrupt handling protocol is illustrated in Figure 3, reproduced here from [4].

[1]On ARM processors, there is only one hardware interrupt, so there is only 1 ISR in the system that identifies the interrupt source(s).

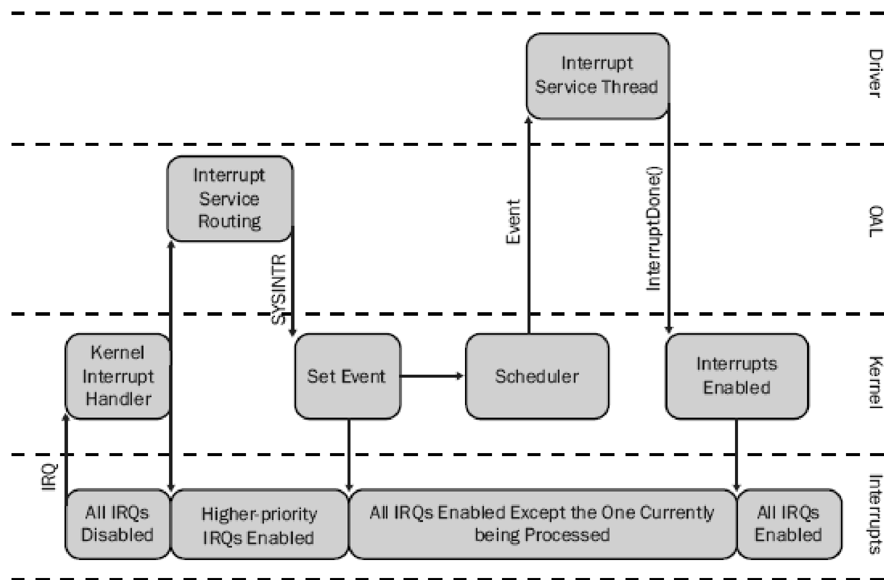

Figure 3: Windows CE 6.0 Interrupt Handling.

DISCUSSION

Interrupt Handling Property	**Linux 2.6**	**Windows Embedded CE 6.0**
Main interrupt handling routine	Interrupt handler	IST
Interrupts disabled during handling	All if registered as fast interrupt	Lower priority disabled during ISR, no interrupts disabled after ISR
Scheduler involved in interrupt handling	No	Yes
Hard real time capable interrupt handling mechanism	No	Yes

Table 2: Interrupt Handling Comparison

Table 2 summarizes the key interrupt handling characteristics examined in this chapter for both Linux and Windows CE. Linux employs a more straightforward interrupt handling strategy than Windows CE: for an interrupt handler registered as a fast interrupt,

which is common, all interrupts are disabled while it is executing. In the time that the the interrupt handler is running, other interrupts that may fire during this time, but they will remain unhandled until the interrupt handler completes, regardless of the fact that it may be higher in priority than the currently handled interrupt. Now, if the unhandled interrupt is something critical like the indication of a collision in an automobile, the system will fail with disastrous consequences for the occupants of the vehicle if the resulting reaction of deploying air bags does not start within a handful of milliseconds, which is a likely scenario if the currently executing interrupt handler does not complete rapidly.

Windows CE, on the other hand, employs a layered interrupt handling mechanism. At each stage, except the initial one when the firing IRQ and its corresponding ISR that is to be executed are identified, all higher priority interrupts are allowed to preempt the interrupt that is currently being handled. The IRQ and its registered ISR identification consumes a much shorter windows of time compared to the rest of the stages, therefore it cannot significantly delay any higher priority interrupts that fire during that initial stage. The ISR is also relatively short and simple since its job is only to find the SYSINTR value that corresponds to the IRQ and to notify the kernel to set any events registered against that SYSINTR value. After that stage, all interrupts of lower and higher priority are re-enabled. The IST, which performs the bulk of the work required by the interrupt, does not run until the scheduler selects it for execution, but then only if there is no higher priority thread currently running. Critical interrupts are guaranteed to be handled without delays caused by lower priority threads and interrupts. Therefore, Windows CE, with its hard real time qualification, is better suited than Linux for applications having strict timing requirement. Windows CE also places the onus on

the developers of the system to fine tune the priorities in the system to ensure that a critical interrupt or IST is not mistakenly assigned a low priority.

Similar to fairness in scheduling, Linux also is fairer in interrupt handling compared to Windows CE. It ensures that the currently handled interrupt is not preempted by a higher priority one. Preemption may also be less desirable when interrupt priorities are not far apart in a system. Suspending a currently executing interrupt service routine in order to service another interrupt that is only slightly higher in priority may be costlier in the long run since it requires a context switch, which means additional processing and, therefore, additional power. Consequently, Linux is a better choice for an embedded system without much distance between the interrupt priorities.

Chapter 6: *Memory Management*

Like processor time, the system's main memory is a precious resource in an embedded system. Via process management and the scheduler, the OS manages the sharing of time on the processor among the applications. Similarly, the OS has to manage the sharing of main memory. Any code that is to be executed and any data that is to be accessed by the code needs to reside in main memory. The size of the memory on the hardware is usually too small to fit all the applications, system libraries, and the kernel. Yet, the application developers are not burdened with any concerns about how the application's memory needs might be satisfied. A primary reason for that is to keep the applications portable to platforms with different sized physical memory. From the application's perspective, it looks as though the entire system memory is available for its sole use. It is responsibility of the OS to allocate or free memory when requested by applications. An approach employed by most operating systems, including Linux 2.6 and Windows Embedded CE 6.0, is the use of virtual memory. [2]

Virtual memory space is the maximum possible addressable space, which for 32-bit systems is 4 GB – typically much larger than physical memory. The OS maps the virtual memory to the physical memory on the hardware as needed and is responsible for divvying up the virtual memory address space among the kernel and the processes. The use of virtual memory requires swapping out a memory page, if the memory is full, whenever a new page is requested that is not resident in the physical memory. A page is the smallest unit of memory that is used during memory management; it is typically 4 KB for 32-bit systems. When a page is swapped out, a page of physical memory containing data is replaced with another page of memory that is read in from storage, such as a disk. The page to be swapped is written out to disk, but it can be brought back into main

memory if it is accessed. The selection of the page to be replaced is a pageout policy decision that depends upon the OS. [2]

The Linux 2.6 kernel, as most modern operating systems, uses virtual memory. Each process is permitted to have its own virtual memory space, which means that it is possible for the same virtual address to exist in multiple processes. The kernel maintains a set of page tables for each process, which maps each virtual address of that process to a unique physical memory address [3]. The Windows Embedded CE 6.0 virtual memory implementation is very similar. Also similar is the memory protection implemented for user space process in both operating systems: For a particular user process, the kernel restricts access to the kernel address space as well to other processes' address spaces to prevent malicious or unintended corruption of another process's memory. When accessing user memory, both kernels have to map the buffer into its address space [4]. How the virtual memory space is divided between the kernel and user space code is one of the key differences between the two operating systems' memory management domain.

For 32-bit architectures, of the maximum possible 4 GB virtual address space, the Linux kernel consumes 1 GB, leaving 3 GB for each process (although this configuration can be changed to a split such as 2 GB / 2 GB or 3 GB / 1 GB). Windows CE, however, has a fixed 2 GB / 2 GB split between kernel and user space [4]. In both OSs, the kernel directly maps 1 GB of the address space to the physical memory on the system for improved performance as fewer page faults are encountered. Of the 1 GB of kernel address space, the Linux kernel directly maps 896 MB to the physical memory on the system, which means a page table lookup is not required to translate the virtual address to its corresponding physical address [3]. The Windows CE kernel only directly maps 512 MB, in comparison. Another non-cached version of the same 512 MB is also mapped by the Windows CE kernel to allow direct access to hardware registers for memory mapped

peripherals [4]. Part of the 1GB kernel space in Linux contains user memory that is mapped into kernel space (e.g., a device driver wants to dump data from peripheral memory directly into user memory), and so it requires page table access and is not direct mapped. Windows CE allocates some memory from the remaining 1 GB of the kernel's address space for a similar mapping of user memory. Typically, embedded systems have less than 1 GB of physical memory, so although the kernel region is direct mapped, the kernel may still encounter a page fault, which occurs when the page is not in physical memory and must be read in from the disk.

From a user process perspective, Linux provides 3 GB of address space, while Windows CE provides 2 GB. The user process's address space has to fit code and data pages along with mapping of shared libraries, which would otherwise require a copy of the library to be loaded in each process's address space. Some portions of the process's address space are also reserved for interprocess communication and memory mapped files, covered in the next chapter. [3] [4]

Memory allocation in support of executing programs is a key responsibility of the operating system. A process requires memory for the variables declared in the code. Variables declared statically are allocated from the process or thread stack. Whereas, dynamically allocated variables consume memory from the process heap, rather than entire pages, typically 4 KB, for small amounts of memory needed for the variables. Heaps permit more efficient use of memory in the system, but they are not immune to fragmentation, when repeated allocations and deallocations by the process leave gaps of free memory in the contiguous heap space. In Linux, a user process's stack is virtually unlimited, while it defaults to 64 KB on Windows CE. But, user thread stack size in Windows CE can be changed by the developer through the use of a linker flag. The same is true for kernel thread stacks. In contrast, the Linux kernel process stack is strictly

limited to 8 KB for 32-bit architectures. In Windows CE, a process heap, which is shared by all the threads in the process can suffer from fragmentation. The Linux heap includes a layer known as the Slab that prevents fragmentation by rearranging the allocated and freed blocks to make the heap contiguous again. Keeping the heap contiguous not only enables more efficient use of memory but also improves performance during memory allocation of a process since the search for free memory blocks completes faster for a contiguous heap. [3] [4]

DISCUSSION

Memory Management Property	**Linux 2.6**	**Windows Embedded CE 6.0**
Use of virtual memory	Yes	Yes
Kernel/User memory split	1 GB / 3 GB	2 GB / 2GB
Memory split configurable	Yes	No
Kernel direct mapped memory	896 MB	512 MB, cached and uncached copies
Memory access and protection scheme	User process cannot access kernel or other user process memory; kernel can access entire memory map	Same as Linux
User stack size / allowed to grow dynamically	As big as needed / yes	64 KB or developer defined / no
Kernel stack size / allowed to grow dynamically	8 KB / no	64 KB or developer defined / no
Heap can suffer from fragmentation	No	Yes

Table 3: Memory Management Comparison

Table 3 summarizes the key properties related to memory management in Linux and Windows CE. While both use virtual memory and employ a similar memory protection scheme, they greatly differ in many areas. Linux provisions a bigger chunk of

the virtual address space for user processes: 3 GB as opposed to 2 GB in Windows CE. This means that user processes in Linux are allowed to grow much bigger than in Windows CE. In the embedded systems space, however, user processes hardly ever grow this big; therefore, this seems to be a negligible difference. It does mean, however, that the kernel space is bigger in Windows CE than Linux, so the kernel has more room to include objects like the Windows registry, which can be helpful for user processes for storing settings. Therefore, Windows CE offers more flexibility with the bigger kernel space. With a smaller direct mapped region and the bigger virtual address space, the Windows CE kernel can potentially suffer from more page faults, leading to slower kernel performance. However, that disadvantage is tempered for embedded systems since the physical memory size is typically small enough to be entirely direct-mapped by both the Windows CE and Linux kernels. Linux offers more flexibility for the developer of the system through configuration of the user-kernel memory split.

Of the memory management properties examined, by far the most important one, from the perspective of an embedded system, which has limited memory, is how each OS handles memory allocation for processes. At first glance, it seems that Windows CE kernel is better because of the bigger default stack size and the ability for the developer to choose a particular stack size. However, the fixed 8 KB stack size for Linux kernel processes forces the developers to make more use of the heap with dynamic memory allocation. The heap in Linux is managed to a much higher degree than Windows CE in an effort to keep memory contiguous and prevent fragmentation. Less fragmented memory means more of it is available and memory requests from processes can be serviced faster. Therefore, Linux is the better choice for embedded systems, from the perspective of efficient memory allocation.

Windows CE, on the other hand, empowers the developer to fine tune the memory usage for her process since the stack size is definable during compile time. The developer of the system can collect some empirical data on her system to determine optimal stack size for the kernel and user space processes. However, when not carefully selected, stack sizes that are too large can quickly deplete the memory pool in an embedded system. Linux may encounter similar problems due to poorly written user processes, whose stack is allowed to grow as much as needed during run time.

Chapter 7: *Synchronization and Interprocess Communication*

In a modern embedded system, it is typical to have multiple applications executing concurrently, which are managed through scheduling. Many applications require synchronization with other processes or the kernel. The degree of synchronization may vary; it could be as little as requiring notification that an event occurred or as much as transferring huge chunks of data back and forth with another process. The system hardware typically possesses one or two instances of a peripheral, such as a serial port or Ethernet port. If one process is currently using a resource, the other process has to wait its turn. But how will the other process realize when the resource is free? Concurrent execution means preemption by the scheduler at any time. How would the newly scheduled process, which accesses some shared critical region of code, be sure that there no process is currently executing in the critical region? Synchronization is the solution to these problems. Synchronization can be viewed as a subset of interprocess communication, which includes data exchanges between processes. [2]

The Linux 2.6 OS provides many forms of synchronization, both for the kernel and for applications. One of the basic synchronization tools available in the kernel is atomic operations, such as test-and-set, at the integer and bit levels. Atomicity ensures that no other process can interrupt that particular operation. This is useful when sharing a small piece of shared data, which can be employed as a method to obtain a shared resource [3]. Windows Embedded CE 6.0 provides similar atomic operations known as interlocked functions [4]. For critical sections of code that must be executed exclusively, and for protecting shared data structures from concurrent access, which could put the data in an inconsistent state, both OSs provide similar synchronization mechanisms in kernel mode: spin lock in Linux and Critical Section in Windows CE. When a process in Linux

tries to acquire a spin lock, it will either succeed if no other process is holding the spin lock or it keeps executing a wait loop until the other process releases the lock. However, in Windows CE, the thread trying to acquire the Critical Section blocks until the thread holding the Critical Section releases it. Processes holding a spin lock are not premptable and are not allowed to sleep; whereas, no such restrictions exist for threads holding a Critical Section in Windows CE. Therefore, it is possible for threads holding a Critical Section to sleep which could negatively impact the overall system performance if there is a lot of contention for the Critical Section. In Linux, a process holding a spin lock is also expected to complete its work as quickly as possible to avoid affecting the overall system performance. [3] [4]

A Linux semaphore is a version of the spin lock that enters a queue and sleeps if the semaphore is not free. When the semaphore is released, a process from the queue is awakened. Semaphores can be configured to allow more than one process access to the shared resource by simply setting the count appropriately. Windows CE also provides an equivalent Semaphore object to permit one or more threads in the kernel to access a shared resource concurrently. Completion variables are another form of synchronization between processes in the Linux kernel. Each process that is interested in a particular event can wait on a completion variable; when the event occurs, all (or one) waiting processes can be woken up. Windows CE's equivalent synchronization object is known as an Event. The Windows CE kernel and user address spaces also provide for a one way communication pipe from the kernel to the user process, referred to as the system heap. [3] [4]

For user processes, shared memory is the most direct, therefore the most efficient, form of interprocess communication available in Linux, but not in Windows CE. Semaphores have to be used to synchronize access to the shared memory in Linux as the

kernel does not ensure exclusivity. Memory mapped files are the most efficient way to transfer large amounts of data between user threads in Windows CE, also requiring the use of semaphores for synchronization. Processes in Linux can also communicate with each other through memory mapped files, similar to Windows CE. For medium sized data transfers, both Linux and Windows CE support TCP/IP sockets that are available in both operating systems' networking library. The Windows CE registry is a space that can be used for medium sized data transfers. In Linux, a pipe can be used to funnel data between processes; however, it only provides one way communication. Pipes have a built-in flow control mechanism so that the writer does not overflow the reader and the reader blocks if the pipe is empty. Similarly, Windows CE provides point-to-point message queues for one way communication. Both the pipe and the message queue are to be used for small amounts of data transfers due to overhead associated with the tools. [7] [8]

DISCUSSION

Synchronization and Interprocess Communication Property	Linux 2.6	Windows Embedded CE 6.0
Kernel mode synchronization tools	Spin lock, semaphore, completion variable	Critical Section, Semaphore, Event
Sleeping allowed when holding lock or critical section	No	Yes
Preemption allowed when holding lock or critical section	No	Yes
User mode synchronization tools	Semaphore	Semaphore, Event
User mode interprocess communication tools	Shared memory, memory mapped files, socket, pipe	Memory mapped files, socket, registry, point-to-point message queue, system heap

Table 4: Synchronization and Interprocess Communication Comparison

Table 4 summarizes the synchronization and interprocess communication tools and characteristics in Linux and Windows CE. In kernel space, an examination of the policy surrounding critical sections reveals some important differences. Linux does not permit preemption or sleeping in processes that are holding a spin lock. Whereas, threads holding a Critical Section in Windows CE can be preempted for higher priority threads or for interrupt handling. Sleeping is also allowed in the thread which holds the Critical Section. This behavior is consistent with Windows CE's hard real time credentials. Since a real time interrupt, and its related ISR and IST, may need to be processed without delay, the current thread, whether it is holding a Critical Section or not, can be preempted. Subsequently, the synchronization in Windows CE is better for an embedded system with hard real time requirements.

Under Linux, the process holding a spin lock has to complete without sleeping or being preempted. Although such a policy can result in delayed reaction to critical interrupts, it has the advantage that processes vying for a heavily contended lock will eventually progress in their execution since the process holding the lock will eventually yield. Heavily contended locks often protect scarce resources in the system without which processes may starve. A communication radio on an embedded device is a good example since, typically, there is only one of those in a system but many processes may be interested in transmitting or receiving on it. Without the strict policy employed by Linux, there is a possibility that a thread that sleeps or gets preempted while holding a heavily contended critical section will cause severe degradation in performance for all processes or threads. In Windows CE, priority inversion is employed to guard against long waits by higher priority threads for acquiring a Critical Section, but the thread holding the Critical Section with its bumped up priority can still sleep. Therefore, Linux synchronization mechanism seems to be better suited for systems with heavily contended resources.

Both operating systems offer similar sets of tools for interprocess communication between user processes. Some tools, however, are unique to each OS. Linux shared memory, for example, serves as an efficient transfer mechanism for large amounts of data. Shared memory is not available in Windows CE; therefore, the developer of user applications that require large data transfers between processes may be better served with Linux. The Windows CE registry, on the other hand, provides a communication medium between processes that does not exist in Linux. The system heap is another tool not available in Linux. It permits the Windows CE kernel to communicate directly with user processes. For user processes, this can be beneficial because information from the kernel can be received by the user process without making any system calls.

Chapter 8: *Power Management*

An embedded system is typically powered by a limited life battery instead of the inexhaustible AC power; therefore, the system has to proactively look for ways to reduce power consumption whenever possible. There are many opportunities to conserve power in various situations ranging from idle time to system suspend. The system hardware has a prime role to play in saving power. Modern embedded processors support varying degrees of power savings, while many peripherals also support clock and power gating. The operating system, however, is responsible for managing and coordinating the power state transitions in the system. The device drivers and board-specific code implement the power saving features over the hardware domain.

Originally targeted for the desktop environment, there was very little power management support in the early Linux kernels. Windows CE, on the other hand, was created for embedded systems so it was much further ahead in the area of power management. As it has evolved for use in laptops and most recently for embedded systems, Linux has undergone a substantial revolution in the power management field. In Windows Embedded CE 6.0, basic suspend and resume are supported in addition to more fine grained power levels for individual device drivers. Similarly, basic support for suspend and resume can be implemented in Linux's machine layer by registering board specific implementations of suspend and resume related functions with the kernel's *suspend_ops* structure. Device drivers can also register specific implementations of suspend and resume functions with the kernel provided hooks in *dev_pm_ops*. Upon request to suspend or resume the system, the kernel calls the registered drivers and the machine layer functions to enter the requested power state. One of the easiest ways to conserve power in a system while it remains on is to turn off the clock to a peripheral

when it is not in use. Device drivers are expected to proactively gate and ungate clocks, assuming hardware support, using clock APIs that must be implemented in the machine layer.

In Windows CE, multiple device driver power states, which can range from D0, the fully on state, to D4, the fully off state, can be used to perform the clock and power gating. Unlike Linux, a clock gating framework is not provided. Drivers and applications alike are notified of a power state change if they have registered to receive the notifications. While suspend and resume calls require a power-aware driver to implement the *XXX_PowerUp* and *XXX_PowerDown* interface, the D0-D4 power notifications are provided via an IOcontrol call. [4] [9] [10]

In Windows CE, the Power Manager driver manages and coordinates the overall system power management scheme. An advantage for Windows CE over the Linux is that, since the Power Manager driver is written in a layered form, power management can be easily customized for a particular platform. On the contrary, an advantage for Linux over Windows CE is that drivers could take advantage of the voltage regulator framework to control their power sources when suspend or resume is called. Windows CE does not provide a framework for power regulator control, which must be provided by the device manufacturer. [4] [10]

Another situation where power savings can be maximized is when the processor is idle because there are no processes or threads to execute. In the case of Linux, the device manufacturer provides an implementation of *cpu_idle* function specific to the processor. *cpu_idle* usually runs a special instruction that puts the processor in a low power state from which it can rapidly return to active state. If the hardware permits, it may also disable the periodic timer that would otherwise regularly wake up the processor just to service the timer interrupt. Likewise, in Windows CE, the device manufacturer has to

provide an implementation of the *OEMIdle* function that can save power when the processor is idle, with a provision to disable the periodic timer interrupt. Newer processors may support Dynamic Voltage and Frequency Scaling (DVFS), which permits reducing the processor's clock frequency and voltage at runtime. Linux 2.6 contains the *cpufreq* framework that can be customized to the processor for implementing DVFS in accordance with a certain policy such as on-demand, which uses the CPU utilization metrics to adjust frequency and voltage. Unlike Linux, the DVFS implementation in Windows CE is left entirely up to the device manufacturer. [4] [10]

DISCUSSION

Power Management Property	**Linux 2.6**	**Windows Embedded CE 6.0**
Suspend/resume support	Yes	Yes
Idle time power savings	Yes	Yes
Granular device power states	No	D0-D4
DVFS, clock and power gating frameworks	Yes	No
Easily customizable power management scheme	No	Yes

Table 5: Power Management Comparison

Table 5 summarizes the key power management characteristics examined in this chapter for both Linux and Windows CE. Although Linux matches the suspend-resume and idle time power saving features offered by Windows CE, the Power Manager driver and the granular device power states of Windows CE make it easier for the developer to implement a consistent power management scheme throughout the system, including

both kernel and user space. For example, the Power Manager can transition a system to the D2 power state when the system has been idle for a certain period of time. It sends out a D2 power state to all drivers and applications registered for the D2 power notification; subsequently, the driver can gate the clock to the peripheral and the application can change its behavior based on the system power state. Flexibility is not lost, however, since each driver can choose to support only the power states that make sense for the peripheral. Linux expects device driver developers to be proactive in implementing power savings, which makes the system more decentralized. Different developers may end up implementing varying degree of power savings in each component of the system, making it difficult to attain a uniform power scheme across the system. Additionally, user processes cannot receive power notifications like they can in Windows CE. Moreover, the ability to customize the Power Manager in Windows CE makes it better suited for developers wishing to fine tune the power policy to a platform.

Linux, on the other hand, provides more tools to the developers to save power. The clock and power gating frameworks shortens the development time to support these features in the drivers. The DVFS framework, *cpufreq*, can be adapted to the processor to save more power by reducing frequency and voltage at run time. Windows CE requires device manufacturers to develop their own frameworks for these items from scratch. While this provides more flexibility to the developer, it can increase development time. Therefore, implementation of power management is faster for embedded systems using Linux rather than Windows CE.

Chapter 9: *Conclusions*

In this paper, we examined two 32-bit embedded operating systems: Linux 2.6 and Windows Embedded CE 6.0. As the layer of software that controls the hardware, the operating system is responsible for managing the hardware to suit the needs of applications that execute on the system. The key aspects of both OSs – process management and scheduler, interrupt handling, memory management, synchronization and interprocess communication, and power management – were discussed to reveal their strengths and weaknesses.

In the area of process management, Linux is better for quick process creation and execution, while Windows CE offers finer grained execution units: threads and fibers. An examination of the two schedulers revealed that Windows CE may be the better choice for less interactive systems since the developer has more control over the system performance. Although both operating systems exercise preemptive multitasking, Linux has the potential for better overall system performance on systems that need to support a high degree of multitasking due to the fairness policy implemented by its scheduler. Windows CE, however, is hard real time capable, whereas Linux can only meet soft real time deadlines, in part owing to the fact that its scheduler does some extra accounting work.

Linux's interrupt handling scheme revealed another reason why Linux is not a hard real time OS: all interrupts, even critical ones, are disabled while an interrupt handler is executing. Whereas, in Windows CE, all higher priority interrupts are enabled while the ISR is running, and all other interrupts are enabled after the ISR completes. Therefore, for an embedded system that has to support applications with strict timing requirement, Windows CE is better. Linux, however, offers more fairness in interrupt

handling since the servicing of the interrupt currently being handled is allowed to complete without preemption in favor of another interrupt. This scheme can be better suited for systems with interrupt priorities that are close together since power-consuming context switches are avoided. Windows CE, however, places more burden on the developer to ensure that thread and interrupt priorities are tuned carefully to ensure that the overall system performance does not degrade.

Both operating systems possess a similar memory management scheme that ensures each process is provided its own unique virtual memory space in which to grow. However, Linux comes out as a clear winner in terms of more efficient memory usage due to a fixed small kernel stack size and better managed process heap that tries to avoid fragmentation. Windows CE, in keeping with the theme discovered in the study of its process management, scheduler and interrupt handling mechanism, allows the developer more flexibility to choose an optimal stack size, but sufficient care must be taken to avoid using needlessly large stack sizes that can deplete the already scarce memory on embedded systems.

In the area of synchronization, Windows CE is better suited for hard real time applications due to the policy that allows thread preemption while holding any Critical Section. Linux, on the other hand, can perform poorly for hard real time applications since a process holding a spin lock cannot be preempted in order to service a higher priority event in the system. Linux is better suited to manage heavily contended resources in an embedded system. Windows CE is not able to guarantee that a thread will not sleep while holding the heavily contended resource. From an application developer perspective, interprocess communication tools in both OSs are comparable to a large degree. However, a couple of tools unique to each OS may provide an advantage. In Linux, shared memory provides an efficient mechanism for transferring large amounts of data.

Whereas, the Windows CE registry is an effective means to transfer data between processes and the system heap is useful for transferring data from the kernel to the user process.

While the Power Manager component makes power management a strong suite for Windows CE, recently added support in the Linux 2.6 kernel has managed to significantly close the gap between the two operating systems. Windows CE is the better choice for implementing a centralized, uniform, and custom power management scheme across the system. Although Linux power management falls short in those areas of power management, the frameworks provided for clock and power gating, as well as for DVFS, enables developers to more quickly implement power savings for their particular platform.

In general, the Linux OS emphasizes efficiency and fairness, while the Windows CE OS emphasizes quick reaction time and flexibility for the developer to optimize the system manually.

Linux® is the registered trademark of Linux Torvalds in the U.S. and other countries.

Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.

References

1. Linux Devices. 2007. Snapshot of the Embedded Linux Market. http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Snapshot-of-the-embedded-Linux-market-April-2007/ (accessed July 11, 2010).

2. Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. 2002. *Operating Systems Concepts*. 6th ed. New York: John Wiley & Sons, Inc.

3. Love, Robert. 2004. *Linux Kernel Development*. Indianapolis: Sams Publishing.

4. Pavlov, Stanislav, and Pavel Belevsky. 2008. *Windows Embedded CE 6.0 Fundamentals*. Microsoft Press.

5. Threads Window. http://msdn.microsoft.com/en-us/library/ms934827.aspx (accessed July 11, 2010).

6. Loh, Sue. Real Time and Threads. 2005. http://blogs.msdn.com/b/ce_base/archive/2005/08/31/458474.aspx (accessed July 11, 2010).

7. Mitchell, Mark, Jeffrey Oldham, and Alex Samuel. 2001. *Advanced Linux Programming*. Indianapolis: New Riders Publishing.

8. Forsberg, Christian. 2006. Interprocess Communication with the .NET Compact Framework 1.0. http://msdn.microsoft.com/en-us/library/aa446520.aspx (accessed July 11, 2005).

9. Power Documentation in the Linux 2.6 Kernel. 2010. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=tree;f=Documentation/power;h=4b415b09c30ddab75e07090693027b39246ff49e;hb=HEAD (accessed July 11, 2010).

10. Opdenacker, Michael, and Thomas Petazzoni. 2010. Power Management. http://free-electrons.com/doc/power-management.pdf (accessed July 11, 2010).

Vita

Anish Chandrakant Trivedi was born in Bhavnagar, India. At the age of 12, he immigrated to the U.S.A with his family. Later, he obtained a Bachelor of Science degree in Electrical Engineering with a minor in Computer Science from Columbia University in New York City. Anish has spent the majority of his professional career at Freescale Semiconductor, Inc. in Austin, Texas in various roles in the Multimedia Applications Division including Systems Engineer and Software Applications Engineer. Since 2007, as a Senior Embedded Software Engineer, he has been involved in writing device drivers and boot loader code for Windows CE and Linux operating systems. While continuing to work full-time at Freescale, in August 2008, he entered the Cockrell School of Engineering at The University of Texas at Austin to pursue a Master of Science in Engineering degree in Electrical and Computer Engineering with a concentration in Software Engineering.

Email: anishtrivedi@hotmail.com

This report was typed by Anish Chandrakant Trivedi.