Advanced Android Power Management and Implementation of Wakelocks

Michael B Motlhabi
University of the Western Cape
Computer Science Department
Modderdam Road
Bellville 7535
2706912@uwc.ac.za

ABSTRACT

This paper looks to investigate how Android operating system manages power on a relatively small platform such as a mobile phone or a tablet. The aim of this document is to further find the differences that exist in the design of Android power management modules as compared to the parent kernel from Linux. We will investigate to check if platforms with smaller batteries like mobile phones require a lot of power management or the same as larger and power hungry devices likes desktop computers. This paper will look deeper into why Google decided to use Linux's kernel as opposed to developing one from scratch.

Keywords

Wakelock, Android, Power Management, Advanced Power Management, Advanced Configuration and Power Interface

1. INTRODUCTION

For normal desktop computer, power management (PM) is used to reduce power consumption and reduce cooling requirements. Lower power consumption means lower heat dissipation, which increases system stability, and less energy use, which saves money and reduces the impact on the environment. For mobile device and embedded system device, it's much more important because the battery power is very limited.

Saving battery for mobile devices has been a goal for the industry for many years and as such reduction of battery consumption is even more important since mobile phones consume more battery than the generation of mobile phones before them. Today operating system developers are faced with a fundamental problem of saving system resources as much as possible. One of the most important resources is power. The introduction of smart phones has made it possible for users to work like accessing the internet, recording video, setting appointments and a lot more on their mobile devices to complete the same tasks they use to on their personal computers. Not so long ago power never used to be of vital importance because the desktop computer was permanently connected to an electrical outlet. On desktop computers power managed and the operating system deals with power but in an almost casual and simple way. Power Management has never been more important to give user more battery life.

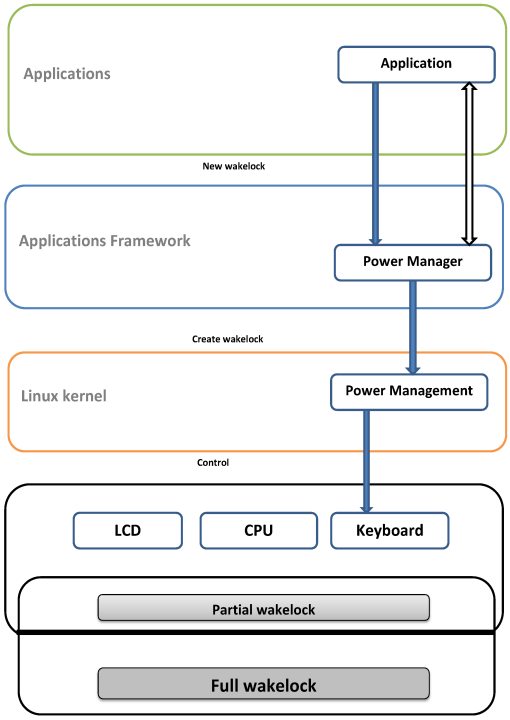

Figure 1: Android Power Mangement Architecture with wakelocks.

Android operating system is a grandchild of Linux which is a desktop operating system. So Android developers had to come with a more power sensitive solution. Figure 1 shows the internal structure of Android and how the power management has been modified to work on a device with limited power. The principal difference is that mobile devices

.

have limited resources like power and desktop computers have "unlimited" power supply. Android uses wakelocks to dynamically power down while the system is still running. This feature is useful for devices that are not being used, and can offer significant power savings on a running system [8]. In order to maximize power saving Android devices support a range of runtime power states which use names such as "off", "sleep", "idle", "active", and so on. In essence a mobile device that can save power and run the longest without the user having to recharge it is ideal.

The rest of the paper is dedicated to discussing the differences between Android and Linux. Section 2 explains the strategies used in Linux to deal with power, here two power management methods are introduced and thier advantages and disadvantages are discussed. Also different power states that exist within the Linux system are examined.

Section 3 discusses how the Android operating system uses the Linux kernel and it's power reduction model to achive a more agressive power management solution. Here the role of different wakelocks are explained and the the relationship between the Linux power manager and the Android power management is broken down. This section investigates how Android is incorporated into the Linux kernel to provide appropriate power management to mobile devices.

Section 4 offers a conclusion to the discusion and exposes how Android is able to achive the goal of saving power given all the challenges related to mobile devices. Here an overall picture of power management in both Android and Linux is conceptualized.

2. POWER MANAGEMENT IN LINUX

Power management in operating systems is important due to the ever increasing power demand of mobile devices especially cell phones. In order to reduce wasted power, multiple hardware power saving features are employed by Linux such as clock gating, voltage scaling, activating sleep modes and disabling memory cache. Each of these features reduces the system's power consumption at the expense of latency and/or performance [7].

These trade offs on a Linux system are managed by either Advanced Power Management (API) or Advanced Configuration and Power Interface (ACPI). APM is an older, simpler, BIOS based power management subsystem, which is still used on older systems [5]. Newer systems use ACPI based management instead. ACPI is more operating-system centric [4] than APM and also offers more features such as a tree structure for powering down devices so that subsystem components are not turned off before the subsystem itself.

2.1 Advanced Power Management (APM)

APM consists of one or more layers of software that support power management in computers with power manageable hardware. APM defines the hardware independent software interface between hardware-specific power management software and an operating system power management policy driver. It hides the details of the hardware, allowing higher-level software to use APM without any knowledge of the hardware interface. The APM software interface specification defines a layered cooperative environment in which applications, operating systems, device drivers and the APM BIOS work together to reduce power consumption. In brief, APM can extend the life of system batteries and thereby increases productivity and system availability [14].

2.2 Advanced Configuration and Power Interface (ACPI)

ACPI allows control of power management from within the operating system unlike APM were power management is done at BIOS level. With ACPI the user can specify at what time a device, such as a display monitor, is to turn off or on. If the user chooses they can specify a lower level of power consumption when the battery starts running low so that essential applications can still be used while other less important applications are allowed to become inactive. The operating system can reduce motherboard and peripheral device power needs by not activating devices until they are needed. With ACPI, the computer can power itself down to a deep sleep state but still be capable of responding to an incoming phone call or a timed backup procedure [14]. In short ACPI gives both the user and the operating system the power to customize the system to suit their personal power needs.

2.3 System Power Management

Power management in any operating system, is considered a necessity due to the ever increasing power demand of todays computer systems [12]. As mentioned in the previous section most Linux based systems manage power consumption via the ACPI.

Linux also has a functionality called System Power Management (SPM): the process of placing the entire system into a low power state. In a low power state, the system is consuming a small, but minimal amount of power yet maintaining a relatively low response latency to the user. The exact amount of power and response latency depends on the state the system is in.

2.4 Power States

The states a system can enter on Linux are dependent on the underlying platform, and differ across architectures and even generations of the same architecture. There tend to be three states that are found on most architectures that support a form of SPM, though. The kernel explicitly supports these states -Standby, Suspend, and Hibernate, and provides a mechanism for a platform driver (an architectural port of the kernel) to define new states.

2.4.1 Standby

Standby is a low-latency power state that is sometimes referred to as "power-on suspend". In this state, the system conserves power by placing the CPU in a halt state and the devices in the state. The power savings are not significant, but the response latency is minimal typically less than 1 second.

2.4.2 Suspend

Suspend is also commonly known as "suspendto-RAM". In this state, all devices are placed in sleep state except main memory, is expected to maintain power. Memory is placed in self-refresh mode, so its contents are not lost. Response latency is higher than Standby, yet still very low between 3-5 seconds.

2.4.3 Hibernate

Hibernate conserves the most power by turning off the entire system, after saving state to a persistent medium, usually a disk. All devices are powered off unconditionally.

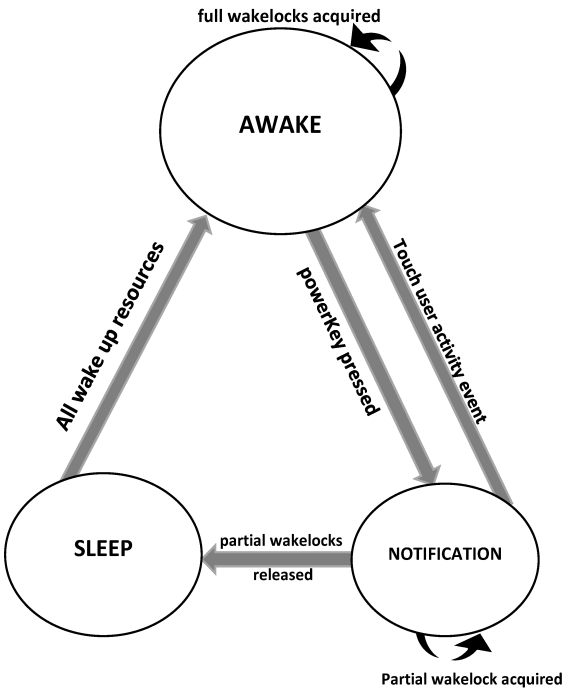

Figure 2: System cycle for flow of events durring process life.

Hibernate is the only low-power state that can be used in the absence of any platform support for power management. Instead of entering a low-power state, the confugured PowerManager driver may simply turn the system off. This mechanism provides perfect power savings (by not consuming any), and can be used to work around broken power management firmware or hardware. The response latency is the highest about 30 seconds but still quicker than performing a full boot sequence.

2.5 Disabling Power Management

There are circumstances in which a driver must refuse a power management request. This is usually because the driver does not know the proper reinitialization sequence, or because the user is performing an uninterruptible operation like burning a CD. It is valid for a driver to return an error from a suspend() method call. For example, a driver may know that it cannot handle the request. This works to the system's benefit, since the PM core can check if any devices have disabled power management before starting a suspend transition [12].

3. POWER MANAGEMENT IN ANDROID

Most of the code in Linux is device drivers, so most of the Linux power management code is also driver-specific. Most drivers will do very little while others especially for platforms with small batteries like cell phones will do a lot. This causes the kernel to be modified for Android to include alarm driver, ashmem (Android shared memory driver), binder driver (Inter-Process Communication Interface) and most importantly power management [14].

Android power management which is built on the top of standard Linux Power Management which takes a more aggressive policy to manage and save power. Android based systems provide their own power management infrastructure labeled PowerManager that was designed based on the premise that a processor should not consume any power if no applications or services actually require power [1]. Android demands that applications and services request CPU resources via wakelocks through the Android application framework and native Linux libraries. If there are no active wakelocks, Android will shutdown the processor.

Figure 2 shows that the full state machine. There are three states: "SLEEP", "NOTIFICATION", and "AWAKE". The scenario is: While a user application acquires full wake lock or touch screen/keyboard activity event, the machine will enter or keep in the "AWAKE". If timeout happen or power key pressing, the machine will enter "NOTIFICATION". While partial wake locks acquiring, it will keep in "NOTIFICATION". While all partial locks released, the machine will go into "SLEEP". In "SLEEP" mode, it will transit if all resource awake. This state machine makes power saving of Android more feasible for mobile devices [5].

The Android kernel design is based on the Linux but does not use a standard Linux kernel. This means the Google team had to find other power saving solutions that are different from the standard Linux power management. As mentioned above there are many different power management techniques that can be deployed by operating system development teams to develop the best power management solution for their environment, such as APM or ACPI. Android platform is running on top of Linux operating system it implements a lightweight ACPI driver which is optimized for embedded systems [16].

A mobile phone will typically spend a large amount of time in a state where it is not actively used. This means that the application processor is idle, while the communications processor performs a low level of activity, as it must remain connected to the network be able to receive calls, SMS messages and so on. As this state tends to dominate the time during which the phone is switched on, the power consumed in this state is critical to battery lifetime. The Android operating system running on the application processor aggressively suspends to RAM during idle periods. Here by all necessary states are written to RAM and the devices are put into low-power sleep mode [6].

Currently Android only supports screen, keyboard, buttons backlight, and the brightness of screen. Because of full usage of CPU capability, it does not support suspend and standby mode. Figure 1 shows how Android PM works. Through the framework, user space applications can use PowerManger class to control the power state of the device. Normally runtime power management is handled by the drivers without specific userspace or kernel involvement. By device-aware use of techniques like: a) Using information provided by other system layers. b) Using fewer CPU cycles. c) Reducing other resource costs like switching off unused power supplies. d) Using device-specific low power states like using lower voltages.

3.1 Android system sleep model

This is a system wide low power state which is the same as hibernate on a desktop. This is something that device, bus, and class drivers collaborate on by implementing var-

FLAG VALUE	CPU	SCREEN	KEYBOARD
PARTIAL WAKE LOCK	on	off	off
SCREEN DIM WAKE LOCK	on	dim	off
SCREEN BRIGHT WAKE LOCK	on	bright	off
FULL WAKE LOCK	on	bright	bright

Table 1: Hypothetical wakelock states

ious role-specific suspend and resume methods to cleanly power down hardware and software subsystems, then reactivate them without loss of data.

The most important concept to understand is that "turning off the screen" is not equal to "sleep mode". Since you may listen to some music or download some games with large size while the screen is off, but the CPU is still running to do these jobs, therefore it's not in "sleep mode." As a result, sleep mode is defined as a mode that the CPU is idle. While the system is in sleep mode, the time spent in this mode is not counted in the device uptime (the device uptime can be obtained by callingSystemClock.uptimeMillis()) [15].

The Android system goes into sleep when its screen and key guard is locked. Now in such a case when an alarm activity gets called, in its OnCreate method a systems wakelock is acquired by the Calling Service, which in this case is Alarm Manager. Now when OnCreate returns it also releases the wakelock. So suppose if you started an audio ringtone in the activity, it won't play until user unlocks the screen. In order to force the system out of this sleep, one has to aquire the wakelock, keep it and release it when no longer needed.

In between acquire and release, the system stays out of sleep and responsive for the user. Now one can also force the system to get out of sleep by calling userActivity(long when, boolean noChangeLights). The system would jump out of sleep until the specified time. It would be good to take a detailed look at the actual documentation for PowerManger [14].

In principle the wakelocks framework is really concerned with whether or not the processing of all wakeup events is complete and, consequently, whether or not it is possible to suspend the system at a given instant of time. That's why it allows user space processes that may take part in the processing of wakeup events to use wakelocks in the first place. However, this means that within the wakelocks framework the races between wakeup events and the suspend process can only be avoided if the relevant user space processes are modified to use wakelocks. While that may be good for Android, whose user space already is designed with using wakelocks in mind, at least to some extent, it is not very practical for other Linux-based systems with user space which is not aware of the wakelocks interface. Still, the possible races between the suspend process and wakeup events affect those systems as well and there are in a way to avoid them without redesigning user space completely.

3.2 Runtime power management model

Drivers may also enter low power states while the system is running, independently of other power management activities. Upstream drivers will normally not know or care if the device is in some low power state when issuing requests; the driver will auto resume anything that's needed when it gets a request.

Android however does not use APM or ACPI directly for power management as shown in Figure 1. Android instead has its own Linux power extension. PowerManager instead the core power driver shown in Figure 1 as "Power" was added to the Linux kernel in order to facilitate this functionality. This module provides low level drivers in order to control the peripherals supported by the Power Manager [2]. These peripherals currently include the screen display and backlight, keyboard backlight and button backlight. Each peripheral's power is controlled through the use of wakelocks. These locks are requested through the API whenever an application requires one of the managed peripherals to remain powered on in each lock setting shown in Table 1.

If no wake lock exists which "locks" the device, then it is powered off to save battery life. In the case of multiple power settings the transition is managed through the use of delays based on system activity. A sample of this behavior is shown in Table 2 for the screen backlight [15]. In addition to wakelocks the PowerManager also monitors the battery life and status of the device. This service coordinates with the power circuitry charging in the battery and also powers down the system when the battery reaches a critical threshold [10].

One of the most important things to notice is that there are two classes of wakelocks: The kernel and user space. All of the user space partial wakelocks go through one partial wake lock, called PowerManagerService (see Figure 2). The full wakelocks and user activity countdown timer are all implemented in Java and call on to sys Android power request state, which turns the screen on and off and implicitly, also keeps the CPU on. The kernel still has sys Android power acquire full wakelock, but it is not used. By the time the KeyEvent wake lock is released, someone has either called userActivity on the power manager, which resets the countdown timer, or they have not. This means that the device should not wake up [11].

Table 1 shows some of the available locks that are implemented in Andoid during system up time. If you hold a partial wakelock, the CPU will continue to run, irrespective of any timers and even after the user presses the power button. In all other wakelocks, the CPU will run, but the user can still put the device to sleep using the power button [3]. Table 1 shows which wakelocks are available and being used by Android to reduce power consumption on a mobile device.

All power management calls follow the same basic sequence: a) Acquire handle to the PowerManager service. b) Create a wake lock and specify the power management flags for screen, timeout, etc. c) Acquire wake lock. d) Perform operation (play MP3, open HTML page, etc.). e) Release wake lock. The Android framework requires the application and services to request a CPU resource with a wakelock, through the application framework and native libraries. Android framework exposes seven different types of wakelocks for applications and services as shown in Table 1. For instance keep the CPU running, while the screen is on.

Developers choose the most appropriate lock for their applications [9]. Incorrect usage of wakelocks, might result in great power waste. It is therefore very important to use the most efficient locks depending in the hardware being used.

If the operating system does not and any active wakelock, it shut down the CPU. Android uses wakelocks partial or full which expose the platform power management to the developer. An Application requests a wake lock, PowerManager makes sure that the device is powered on until the application releases the wakelock [13]. An alternative way of implementing wakeness is userActivity which takes a time period and keeps the application processing part of Android alive. This is one method to keep power consumption down.

4. CONCLUSION

This paper examined the main differences between Android for mobile devices and Linux power management techniques. We also have examined Android in a bottom-up fashion from its architecture to obtain a general overall understanding. We focused on the following topics power management and wakelocks. We got to understand why and how Android uses wakelocks to solve run-time and sleep state power issue. For instance Android's wakelocks address a number of problems. They allow the kernel to avoid races between the suspend process and wakeup events, wakelocks are activated whenever wakeup events occur, which prevents the kernel from suspending the system or causes it to abort suspend in progress [7]. Also they provide a direct standard for the kernel to decide when opportunistic suspend should be started, the kernel attempts to suspend the system whenever there are no active wakelocks.

Moreover, it helps to avoid the problem with "bulk" applications that may cause the system to use too much energy, "bulk" applications are not allowed to use wakelocks and therefore they cannot prevent the kernel from suspending the system [14]. In addition to that, it makes the kernel collect statistics related to wakeup events. Namely, each wakelock object contains fields in which statistical information is recorded, like the number of times the wakelock has been activated, the total time it has been active, or the maximum time it has been active continuously, whenever it is activated or deactivated.

The Android power management method has not had a clean run though, and the most controversial aspect of the Android's suspend infrastructure is that it starts system suspend from kernel space, basically taking over an interface that was intended for use by (privileged) user space processes and because of that it requires applications to interact with the kernel in a very unusual way and the interface provided by it for this purpose does not really match other interfaces between the kernel and user space.

Android power management techniques of using wakelocks on the Linux kernel (Linux uses wakeup events) were not efficient when they released the G1 (first Android phone), but since then they have become more and more reliable and energy efficient [5].

5. REFERENCES

[1] L. Benini and G. D. Micheli. Case study of performance of real-time linux on the x86 architecture. *Proceedings of the Sixth Real-Time Linux Workshop*, pages 18–25, 2004.

[2] L. Benini, G. D. Micheli, and L. K. Morallez. Case study of performance of real-time linux on the x86 architecture. *in FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science. Washington, DC, USA: IEEE Computer Society*, pages 372–378, 1995.

[3] A. Bogliolo, S. Cavallucci, and B. Ricco. Monitoring system activity for os-directed dynamic power management. *Proceedings of the 1998 international symposium on Low power electronics and design*, 5:28–33, Aug 1998.

[4] D. Bornstein. Dalvik vm internals. *Google I/O Developer Conference*, 23:17–30, November 2008.

[5] P. Brady. Android anatomy and physiology. *Google I/O Developer Conference.*, 22:1–16, June 2009.

[6] E. Harris and M. Sri-Jayantha. *Portable tools for Performance analysis*. Prentice Hall, 2006.

[7] H. Hartig, M. Hohmuth, and P. A. Barns. System integration for the android operating system. *Proceedings of CollaborationCom.*, 7:73–88, 2010.

[8] M. Hohmuth, F. Maker, and T. Brown. A survey on android vs. linux. *Proceedings of the 16th ACM Symposiumon Operating Systems principles*, 6, November 1997.

[9] C. Hwang and A. Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM Transactions on Design Automation*, 13:18–25, 2000.

[10] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 68–77, 2005.

[11] N. Kappiah and D. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 44:166–172, November 2005.

[12] S. Salas and E. Hille. *Introduction to Android Internals*. John Wiley and Sons, New York, 2007.

[13] M. G. Srivastava and R. W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Trans. Very Large Scale Integr. Syst.*, 4:42–55, 1996.

[14] M. Stemm and Y. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *13th ACM Symposium Operating Systems principles*, 16:102–107, 1997.

[15] A. S. Tanenbaum, N. Jorrit, and H. Herder. Can we make operating systems reiable and secure? *ACM Transactions on Design Automation*, 39:5–44, 2006.

[16] F. Yao and S. Shenker. A scheduling model for reduced cpu energy. *Proceedings of PART 98*, pages 68–77, 2005.