Comparison of Android OS and Minix internal architecture

Michael B Motlhabi
University of the Western Cape
Computer Science Department
Modderdam Road
Bellville 7535
diablonuva@gmail.com

ABSTRACT

This document explains the two main kernel architectures of operating systems: the monolithic kernel which is used by Android and the microkernel which is used by Minix. In this paper we look at the two approaches of building and designing the operationg system (OS) and compares which is the best and what are the related benefits and disadvantages. We look at how the Android OS has been modified from the Linux kernel to work on a mobile station like a cell phone with limited resources. This report further aims to discuss the different ways in which a monolithic kernel performs Inter-Process Communication (IPL) compared to a microkernel. We will end by discussing where and why one of these designs can better be used to perfom at thier peak.

Keywords

Kernel, Monolithic, Microkernel, Android, Minix, Device Drivers

1. INTRODUCTION

Kernel is the most important part of an operating system, as Figure 1 shows, and it consists of two parts, kernel space (privileged mode) and user space (unprivileged mode). The early concept of kernel, monolithic kernel, dedicates all the basic system services like memory management, file system, interrupt handling and I/O communication in the privileged mode. Constructed in layered fashion, they build up from fundamental process management up to the interfaces to the rest of the operating system [6]. Running all the basic services in kernel space has several drawbacks that are considered to be serious, for example, large kernel size reults in lack of extensibility and bad maintainability. A large kernel size with large Lines of Code (LoC) make the system to be unreliable. Recompilation of the whole kernel is needed during error fixing or addition of new features and fuctionalities in the system which is time and resources, resources which might not be available in a small device like a cellphone.

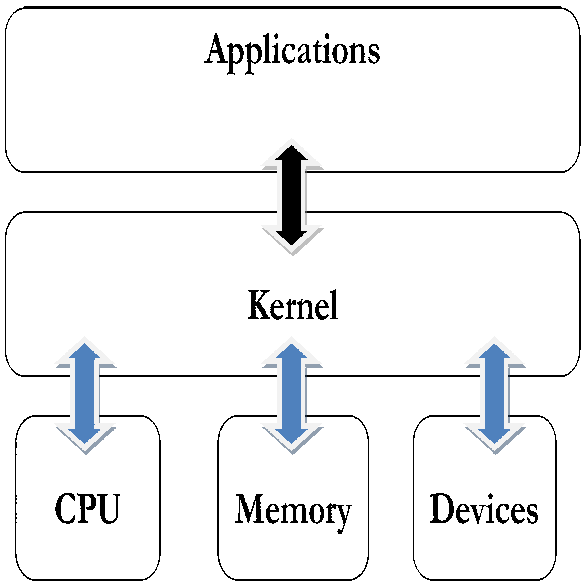

Figure 1: The generic architecture of operating systems.

So deciding which OS to use and how its stuctured is very important.

2. ANDROID OS ARCHITECTURE

The architecture The Android operating system (OS) run on a mobile phone and this makes it face principal challenges when it comes to things like memory management, security, process management, power management, network stack and the driver model. To better face these challenges Android uses a modified version of the well tested and well known Linux kernel version 2.6 for the above mentioned core system services.

Android OS is is owned and developed by Google, and Google usually refers to the android OS as a software stack. In this stack there are layers which group several programs together that support the specific operating systems functions. The modified Linux kernel acts as an abstraction layer and sits between the software stack and the rest of the hardware. Figure 1 explains this concept and shows the relationship between the kernel and the rest of the system.

2.0.1 The origin of Android architecture

Since Google used the already existing Linux version 2.6

operating system to build Androids kernel it inherited some of its functionality, especially when it come to hardware drivers. Drivers live in the kernel, they are important programs that control the hardware of the device [1]. The presences of device drives also allow programmers to write high-level applications independently for whatever specific hardware device. Take for example the Sony Ericsson X-10 has a camera [6].

The Android kernel includes a camera driver, this driver is very important for allowing the user to send commands to the camera hardware. Without drivers the user can not be able to use the hardware of his/her device. Typically a device driver would work as follows a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating system specific. They usually provide the interupt handling required for any necessary asynchronous time dependent hardware interface.

2.0.2 The four Android layers

In the simplest case as shown on Figure 2, the android architecture has four layers. From the bottom up its the Linux kernel, libraries with Android runtime, the application framework layer and finally the applications layer. On the same level as the libraries is the Android runtime layer includes a set of core Java libraries which has the Dalvik virtual machine. These java libraries are what application programmers use to build and run their applications. We can think of libraries as a set of instruction that tell the device how to handle different kinds of data.

2.0.3 Android Libraries

As I mentioned above the libraries layer also has a virtual machine called Dalvik. The Dalvik virtual machine is just like any other software application but it behaves it is an independent device with its own operating system. The importance of the Dalvik virtual machine in Android is that it lets the Android OS run each application as its own process [8].

This method is critical for a number of reasons. This means that no application is dependent on another. Should an application crash, it will not affect any other applications running on the device. For example, if the camera cashes, it will not affect the USB port on the phone or the screen, because these are essentially running on different processes. One more important reason is that this approach is good for memory management. This is because Android applications are converted into Dalvik Executable format which is can be executed very quickly.

2.0.4 Androids monolithic kernel

Since Android runs a modified version of the Linux kernel it is a monolithic kernel. A monolithic kernel is an operating system architecture where the entire operating system is working in the kernel space. This means device drivers and kernel extensions run in the kernel space with full access to the hardware, but not all, some run in the user space. For instance file systems run in user space while display drivers run on kernel space. Operations running in the kernel space have authority over the ones that run in user space [7, 9]. This is called kernel mode preemption, which allows device drivers

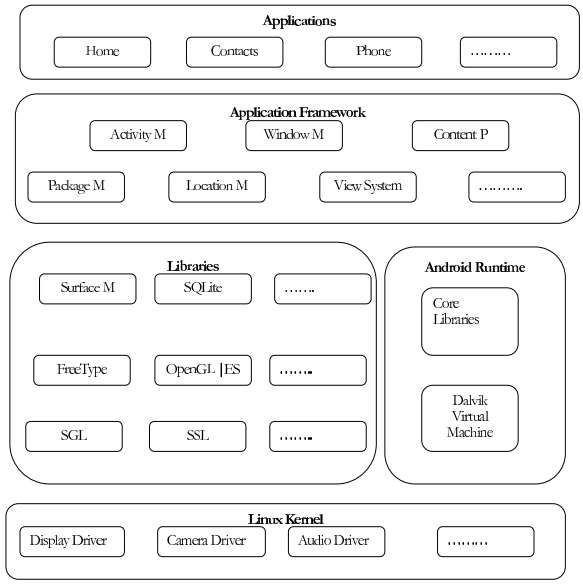

Figure 2: The architecture of Android OS.

to be preempted under certain conditions. This functionality was introduced to handle hardware interrupts correctly and efficiently and to also improve support for multiprocessing. The Android OS supports multiprocessing because of the above mentioned functionality and because of its Dalvik virtual machine which allows applications to have their own processes.

2.0.5 Message passing and process communication

Monolithic kernels pass messages differently from other architechures. For Androids kernel there are several methods for inter-process comunication (IPC) methods between thread used in Linux 2.6 version, such as signals, pipes, FIFO, system V IPC, sockets and system V IPC which includes message queues, semaphores and shared memory [3]

3. MINIX OS ARCHITECTURE

Minix is the forefather of Linux, but their internal architecture differs fundamentally in that Minix has a microkernel and Linux a monolithic kernel. The creators of Minix and Linux engaged in heated debates about which architecture is the best. Tanenbaum said “writing a monolithic kernel is a giant step back into the 1970s“.

3.0.6 Introduction of Minix

When Minix came into the scene it had been well established that most operating system crashes are due to bugs in device drivers. Because drivers are normally linked into the kernel address space, a faulty device driver can wipe out kernel tables and bring the system crashing to a halt. This had to do with the architecture of the operating systems of that time. The Minix operating system greatly mitigated this problem by reducing the kernel to an absolute minimum and running each driver as a separate, unprivileged process in user space [7].

3.0.7 Internal structure of Minix

Figure 3 shows the architecture of the Minix microkernel. If we look at the user space its clear that each driver is

a user process protected by the memory management unit (MMU) the same way ordinary user processes are protected. They are special only in the sense that they are allowed to make a small number of kernel calls to obtain kernel services. Typical kernel calls are writing a set of values to hardware I/O ports or requesting that data be copied to or from a user process.

3.0.8 Message passing and process communication

The architecture of Minix is such that most of the work is done in user space. This fact requires a way in which processes can communicate and because Minix it allows all processes to cooperate, interprocess communication (IPC) is of crucial importance in a multiserver operating system. However, since all servers and drivers in Minix run as physically isolated processes, they cannot directly call each other's functions or share data structures. Instead, Minix does an IPC by passing fixed-length messages using the rendezvous principle.

When both the sender and the receiver are ready, the system copies the message directly from the sender to the receiver. In addition, an asynchronous event notification mechanism is available. Events that cannot be delivered are marked pending a bitmap in the process table. Minix then elegantly integrates interrupts with the message passing system [9]. Interrupt handlers use the notification mechanism to signal I/O completion.

This mechanism allows a handler to set a bit in the driver's 'pending interrupts' bitmap and then continue without blocking. When the driver is ready to receive the interrupt, the kernel turns it into a normal message. [2, 5, 4]. And this is how Minix is able to let processes in user space communicate with its microkernel.

3.0.9 The Minix microkernel

Minix differs from Linux and effectively from Android in a sense that Android and Linux are monolithic kernels and Minix is a microkernel as shown in Figure 1. A small microkernel runs in kernel mode with the rest of the operating system running as a collection of fully isolated user-mode server and driver processes. Since Minix uses a microkernel this means that kernel stacks such as device drivers, protocol stacks and file systems are removed from the kernel to run in user space [2].

The advantage of this is that should there be a misbehaving or malfunctioning driver it will not cause the whole system to fail. If a device driver is running from the user space then it can be fired up and killed just like any other application. Sometimes device drivers need direct memory access (DMA) so they can write data arbitrary locations of physical memory and such device drivers need to be trusted these drivers can live in the kernel. Most device driver are not DMA capable and as such in a microkernel they live in user space[9]. Above the microkernel is the device driver layer. Each I/O device has its own driver that runs as a separate process in its own private address space, protected by the memory management unit (MMU) hardware [1, 3].

Because operating system consists of a microkernel process and a set of user processes, all typically running in a common virtual address space. The microkernel controls access to hardware; allocates and deallocates memory; creates, destroys, and schedules threads; handles thread synchronization with mutexes; handles interprocess synchronization

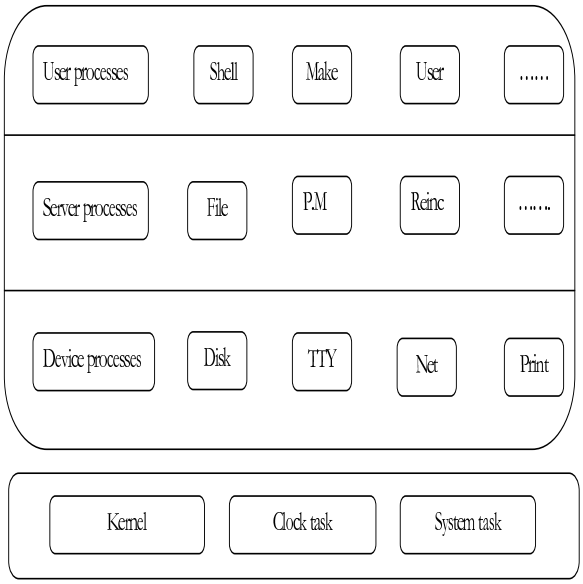

Figure 3: The generic architecture operating systems.

with channels; and supervises I/O. Each device driver runs as a separate process not interfering with another.

The best advantage of designing an opreating system this way is that all kernel data structures are static [2]. All of these features greatly simplify the code and eliminate kernel bugs associated with buffer overruns, memory leaks, untimely interrupts, untrusted kernel code, and more. Of course, moving most of the operating system to user mode does not eliminate the inevitable bugs in drivers and servers, but it renders them far less powerful [5, 4]. A kernel bug can trash critical data structures, write garbage to the disk, and so on; a bug in most drivers and servers cannot do as much damage since these processes are strongly compartmentalized, and they are very restricted in what they can do [3].

4. MICRO VS MONOLITHIC KERNEL

In this section we will compare the two operating system that we have been discussing above, we will look at both graphical and tabular representations of both the monolithic and microkernel. From this we will be able to do a direct comparison of operating system kernels and highlight the design choices, and provide insight into different niches and the evolving technology of kernels.

Since operating systems are designed to be functional for their intended purpose; in this respect they are difficult to compare. Benchmark numbers are rarely enough when comparing two or more operating systems. For example, while message passing often slows down microkernel designs there are many real-time microkernels. On a similar note, monolithic designs are often toted as being faster, yet many monolithic systems (such as Linux) are not capable of real-time operation.

Table 1 highlights some of the common benefits and pitfalls associated with the two operating system designs discussed in this paper. It should be noted that this table only suggests the common attributes of each design; as we have seen in Minix and Linux, both monolithic and microkernel-based designs are able to grow into very flexible, complete

Table 1: Monolithic kernel vs Microkernel

	Microkernel	Monolothic kernel
Advantage	Fault tolerant,	Direct implementation of kernel
	Easy to develop extra functionality	Fast; less overhead
	Less errors in kernel	System calls are used
	Clean programming API	Higher performance
	Portable	
	Easily adaptable	
	High reliability	
	Easier to design and follow	
	Limitless microkernel features	
Disadvantages	Slower; increased overhead	Hard to develop extra functionality
	No multiprocessing	Complicated programming API.
	Low performance	Not necessarily as secure
	More software of interfacing	Larger kernel size
	Message bug are harder to fix	Lack of extensibility
	Complicated process management	Unexplained usage

operating systems. The advantages and disadvantages of each design paradigm have to be regarded as flexible, each design suggests, but does not dictate, how a finished operating system will behave.

5. CONCLUSIONS

There are a number of ways to stucture an operating system. The primary reason for deciding to use one architecture instead of another lies in the requirements of your sysytem and the hardware it is running. Android uses a modied version of the Linux monolithic kernel which is one single program that contains all of the code necessary to perform every kernel related task. Most UNIX kernels are monolithic by default but ecently more UNIX systems have been adding the modular capability which is popular in the Linux kernel. The Linux kernel started off monolithic, however, it is now going towards a modular/hybrid design for several reasons. Microkernels have a different structure to the one we mension above.

Device drivers are outside the kernel space inside the user space and so communicate most microkernels use a message passing system of some sort to handle requests from one server to another. The message passing system generally operates on a port basis with the microkernel. As an example, if a request for more memory is sent, a port is opened with the microkernel and the request sent through. Once within the microkernel, the steps are similar to system calls.

Using this method creates a larger running memory footprint, more complicated interfacing software is required. The Linux team has found a way to make thier architecture more effecient and practical by introducing an achitecture that maximizes the advantages of the monolithic and the microkernel. As such Android runs a Modular or Hybrid kernel which adds fast development time for drivers, on demand capabitity and faster intergration of third party technology.

6. REFERENCES

[1] H. B. Andrew S. Tanenbaum, Jorrit N. Herder. Can we make operating systems reiable and secure? *Computer.*, 39(5):44–51, 2006.

[2] A. C. et al. An empirical study of operating system errors. *Proc. 18th ACM Symp. Operating System Principles, ACM Press.*, pages 73–88, 2001.

[3] H. H. et al. The performance of microkernel-based systems. *Proc. 16th ACM Symp. Operating System Principles, ACM Press*, pages 66–77, 1997.

[4] J. L. et al. Unmodified device driver reuse and improved system dependability via virtual machines. *Proc. 6th Symp. Operating System Design and Implementation,*, pages 17–30, 2004.

[5] M. S. et al. Recovering device drivers. *Proc. 6th Symp. Operating System Design and Implementation, ACM Press.*, pages 1–16, June 2003.

[6] M. H. Herman Hartig and J. Wolter. Taming linux. *the proceedings of PART 98*, pages 68–77, 2005.

[7] J. Liedtke. Testing the performance of u-kernel-based systems. *13th ACM Symposiumon Operating Systems principles*, pages 102–107, 1994.

[8] N. M. Qingguo Zhou, Wang Baojun. Case study of performance of real-time linux on the x86 architecture, the sixth real-time linux workshop. *Linux architecture*, pages 18–25, 2004.

[9] J. Soltys. Linux kernel 2.6 documentation. *Masters thesis, Comenius University, Bratislava.*, November 2006.

APPENDIX

A. HEADINGS IN APPENDICES

A.1 Introduction

A.2 Android OS architecture

The four Android layers.

Android Libraries.

The origin of Android architecture.

Androids monolithic kernel.

Message passing and process communication.

A.3 Minix OS architecture

Introduction of Minix.

Internal structure of Minix.

Message passing and process communication.

The Minix microkernel.

A.4 Micro vs Monolithic kernel

A.5 Conclusions