PORTING ANDROID TO MICROKERNEL: AN EXPERIMENTAL APPROACH

by

JUNZHE ZHANG

A THESIS

submitted in partial fulfillment of the requirements for the degree

BACHELOR OF SCIENCE

Department of Computer Science
College of Literature, Science and the Arts

UNIVERSITY OF MICHIGAN
Ann Arbor, Michigan

2014

Approved by:

Major Professor
Peter M.Chen

Abstract

As smartphones become more popular, the requirement of dependability is also higher. This thesis tries to achieve this goal by providing an architectural solution. As the most widely used modern mobile system, Android is studied as a special case. This thesis proposes a design of porting Android to Minix3, a microkernel system. A preliminary porting work is also presented.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

Smartphones have gained big popularity these days. By combining computing power of desktop and connectivity of mobile devices, smartphones provide features and functionalities that benefit people. Therefore, smartphones, as wells as other smart mobile devices, have been primary tools of communication and computation for many people.

A lot of investment has been made to explore what can be done with their computing power and connectivity. Companies like Google, Microsoft and Apple have separate departments specialized in working on mobile devices. Many new applications have been introduced and realized. Mobile payment is not a new thing for most people. Some machine learning technique are utilized to personalize you phone. More personal information are collected and analyzed through mobile devices. These applications demand high dependability and security.

1.1 Demand on Dependability and Security

Most of smartphone operating system enhance dependability and security by setting highly restricted access control model. However, this solution relies on integrity of system kernel. But kernel integrity can no longer be guaranteed on rooted devices. Another solution is virtualization. For example, Android runs its application on Dalvik, a modified Java virtual machine. However, this solution has also been criticized for performance degradation. Also, Android introduces several new modules to Linux kernel to enable a Server-client model. This will be further discussed in Chapter 4.

These solutions are all based on an assumption that the system is based on a monolithic kernel design, which links many modules into kernel. Though there is a clear logical structure with a basic executive layer and extensions that provide added functionality, all code runs in a single protection domain. Also, all the code runs in a single address space, this means that an invalid pointer or buffer overflow in any module can easily trash the entire system. Therefore, mobile operating system doesn’t put too much trust on its kernel and introduce extra mechanism to enhance dependability.

If the operating system is, instead, based on a microkernel, which has already taken dependability into consideration with a multi-server design, we could potentially improve dependability. Also, with a modular design, restrictions imposed later by some mobile system might be removed, meaning that system dependability can be improved without compromising performance.

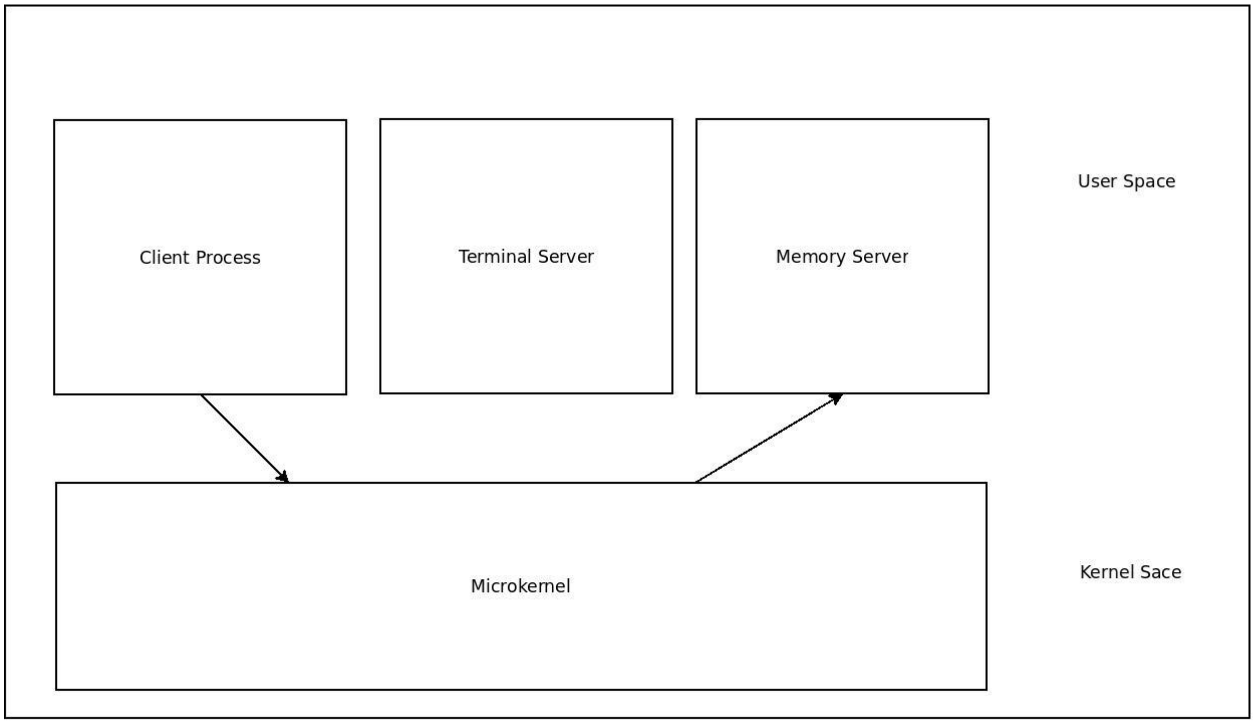

Fig 1. Structure of a multi-server microkernel system

1.1.1 A Modular OS Design

A multi-server design divides the OS functionality into several independent user-level processes. The ability of each single process is also tightly controlled. Kernel only maintains minimum set of basic functionality that cannot be done in user space. This type of design is called microkernel. The structure of a microkernel-supported multi-server design is illustrated in Fig 1.

1.1.2 Pros and Cons

While moving most of OS out of kernel doesn't necessarily reduce the number of bugs, it make bugs less devastating by making a kernel-level bug to user-level. In monolithic design, a kernel bug is highly likely to spread and damage the entire system. By contrast, a bug in an isolated sever in multi-server design can be reduced into a local problem. By tightly controlling a server's accessibility and scope, kernel can easily turn down the faulty components and try to fix it on-fly. For example, driver failures are one of the most common failures in today's OS like Windows XP and Ubuntu. System usually trust driver modules, assuming they are dependable.

However, this is not always the case. Driver code are usually written by less-experienced developers, they tend to violate principle that the system impose. Also, due to the update rate of current hardware devices, driver code are produced in a fast pace, therefore not fully tested. A classification of driver failures on Window XP shows that the use of user-level driver would structurally reduce 67% of 2528 OS crashes analyzed [Ganapathi et la, 2006].

There are two primary drawbacks of multi-server system design. First, after changing a kernel module to a user-level process, its ability of accessing other parts of OS are restricted. However, it needs to communicate with other components to perform task which it cannot by itself. Therefore, a small performance overhead due to context switching and message passing is expected. Second, a full backward compatibility is not possible for running a kernel driver as user-level process. Several changes need to be done to driver code. Furthermore, some driver modules are no longer suitable to be implemented as driver modules in multi-server system design. Other type of abstraction needs to be found for these modules. Nevertheless, the compatibility of multi-server systems can be achieved at a level high enough for practical use.

1.2 Special Case: Android

Now, I will focus on a special case of mobile operating system, Android. There are two reasons that I choose Android. First, Android, combined with its variations, account for 77% of all smartphones shipped worldwide in 2013 [ABI Research, 2014]. Second, Android is open. Strictly speaking, the Android is not a purely open-source project. Proprietary software like device driver are kept close to protect manufacturer’s benefits. But most of Android source code is released under the Apache License. The code available online is enough for study.

1.2.1 Popularity and Variations

According to recent report released by ABI Research, Android shares the most market of mobile operating system by supporting 77% of smartphones shipped in 2013. Another interesting need to be mentioned is that is market share includes 25% of modified Android versions. These versions, including Cyanogenmod, are forked from AOSP (Android Open Source Project). These system are compatible with Android but adding more features that Android doesn’t come with. Some of them eventually evolve into a different operating system. For example, Firefox OS is a Linux kernel-based open source operating system for mobile devices developed by Mozilla. It is built upon Android modified kernel and a subset of Android middleware code.

1.2.2 Threats

Though many different versions are released or under developing, they are all based on the same kernel, a Google modified Linux-kernel. It is a monolithic design, featuring high performance. However, the design of Android system indicates that Android system doesn't fully trust it kernel. To enhance it dependability, Android runs all its application on Dalvik virtual machine and enforce isolations between applications and native system services. A special process, Zygote, is created during initialization to monitor access control of each system components. These solutions rely on the integrity of Android kernel. However, kernel integrity cannot be ensure when a device is rooted. Rooting can be done voluntarily by the user or by malware. A recent study found 88 vulnerabilities in Android kernel [Coverity Inc, 2010], which indicates that rooting is a serious threat. These vulnerabilities is inherent to the monolithic architecture of the Android kernel. Also, because Android and its forked versions are based on a same set of kernel, this makes it vulnerable to a same type of attack.

Therefore, it is worthy of providing a microkernel support for the Android. A microkernel system might not remove current security leaks or bugs existing in Android. It could serve as architecture for a high demand on dependability.

1.3 The Focus of This Thesis

Having discussed background of current mobile operating system and its drawback, it's time to define exact focus of this thesis and it contributions.

The focus of this thesis is mainly about porting Android system to Minix, a microkernel system. Minix is operating system which is originally introduced as teaching materials. It puts modular system design to an extreme. In recent years, Minix community has put a large amount of effort to support more software and platforms. The details of Minix will be further discussed in Chapter 3. By porting Android system onto a microkernel, we can discover how design of Android can be split and fit into microkernel design. Due to time limit, this thesis will focus on implementation details of three proprietary drivers: logger, binder, and ashmem, which support the entire Android system. Different ways of implementation method will be discussed. In addition, prototype of porting Zygote process, which hold the entire Android runtime, will also discussed.

Now that the focus of this thesis has been positioned, we offer a view on main contributions of thesis. First, this work provide a detail description the design of Android in detail. Though Android has been the most popular mobile operating system, few material is available discussing the implementation of Android middleware and driver code. Another contribution of this thesis is providing a prototype of how Android can live on a microkernel system. Preliminary work, including poring primary drivers of Android are done and analyzed.

Chapter 2 - Previous Work

Many previous work has been done in supporting other operating system with microkernel. In this section, we present those which have been influential for later operating system design. Section 2.1 talks about L4 microkernel family. Section 2.2 discusses Mach and its efforts of supporting BSD and Linux. Sections 2.3 is about Exokernel.

2.1 L4

L4 is a family of second-generation microkernels. It was created by Jochen Liedtke [Liedtke, J, 1995]. There have been various re-implementations of the original binary L4 kernel interface. Now, L4 applies to the whole microkernel family including the L4 kernel interface and its different versions.

L4 microkernel family can be regarded as the fastest microkernel. The reason for its efficiency is that besides microkernel design, it puts high performance in high priority throughout its design. Compared with other microkernel system, it has small cache footprint. This makes difference in performance between L4 and other microkernels.

2.1.1 L4Linux

L4Linux is a port of Linux kernel to the L4 kernel API. It ported Linux by a hypervisor approach. L4Linux runs in user-mode on top of the L4 kernel, side-by-side with other L4 kernel application [Härtig, H., Hohmuth, M., & Wolter.J, 1998]. It is binary compatible with the normal Linux/X86 kernel and can be used with any Linux distribution.

2.1.2 L4Android

L4Android encapsulates the original Android operating system in a virtual machine. It is based on L4 kernel. It allows for highly secure applications to run side-by-side with the virtual machine. In addition, they provide a drivers for hardware abstraction layers to let Android operating system running on the virtual machine [Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., & Peter, M., 2011].

2.1.3 P4/Linux

P4/Linux is a small real-time systems developed by SYSGO AG company. It is a V2 compatible L4 microkernel called P4. Then it runs on x86, ARM and PowerPC architectures. On top of this kernel, a port of Linux based on User Mode Linux was down. It mainly follows traditional L4Linux design.

2.2 Mach

Mach is a multiprocessor operating system kernel and environment developed by Carnegie Mellon University. It is often mentioned as one of the earliest versions of a microkernel. It defined some concepts that now are common in microkernel design. However, not all versions of Mach are microkernel. The project was ran by Carnegie Mellon University from 1985 to 1994. Its derivatives are still in use in many commercial system such as Mac OS X.

2.2.1 Mach and BSD

Mach was originally implemented as a drop-in replacement for BSD version of UNIX. The motivation for replacing UNIX kernel is that UNIX's concept of file might not be suitable for system with high demand on efficiency.

Mach started with by replacing UNIX's concept of file into a task, which can be viewed as a process. A task can consist of a number of thread [Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., & Young, M. 1986]. Though this might be common for most people today, Mach was the first system to define concept of task and thread in modern way.

Mach also implemented an inter-process communication (IPC) system. Processes pass message to get service or resource. This leads to one of the biggest difference between UNIX and Mach. In UNIX, user-level program access kernel procedure by trapping. However, in Mach, user-level program sends a message to a predefined port of kernel. This concept was later widely deployed by other modern system.

2.3 Exokernel

Exokernel is an operating system kernel developed by MIT. Now it also refers a group of operating system deploying similar designs.

Exokernel gives each user process a clone of the actual computer with a subset of the resources. Exokernel running in kernel mode at the bottom of all other processes. Its job is to allocate resources to virtual machines and then doing sanity checks. Therefore, exokernel can support multiple operating system by virtualization. Each virtual machine is running independently with each other except that it is restricted to only using resources that exokernel has allocated for it.

Chapter 3 - Minix3 Design

This chapter provides an insight of Minix3 Design. Emphasis is placed on those features that are different to monolithic design that people may be familiar with. This chapter focuses on how microkernel design affects Minix3.

This chapter starts with an overview about Minix system. Then Section 3.2 – 3.6 respectively offers details of processes, IO, memory management, file system and reincarnation server of Minix3.

3.1 Minix3 Overview

Minix was originally written as a teaching material by Tanenbaum as an alternative to UNIX system in 1987. As Minix develops, more and more UNIX-like features were added into the system. In 1997, Minix2 was released. The development continued slowly till 2004. The system was redesigned and released as Minix3 [Andrew, S. T., & Albert, S. W. 2006].

The Minix follows microkernel design and puts it to an extreme. The kernel size are restrictively controlled to reduce kernel bugs. Device driver are not fully trusted, and moved to user-space. The system follows multi-server structure, which runs most of system services as user-mode processes. Client processes request services by sending messages to specific server process. As shown in Fig 2, the system can be separated into three layers: kernel, driver, servers and applications. The kernel in the bottom layer schedules processes and manages the transitions between ready, running, and blocked states of processes. Message handling is also done in kernel. I/O ports and interrupts access are also supported by kernel.

Drivers and servers are both running as independent user-level processes in layer 2 and 3. However, driver processes have the most privileges. Server processes provide useful services to the user processes. Process manager, file system, virtual memory and reincarnation server are among those most essential processes in Minix3. Application processes are running on layer 4. They have least privileges.

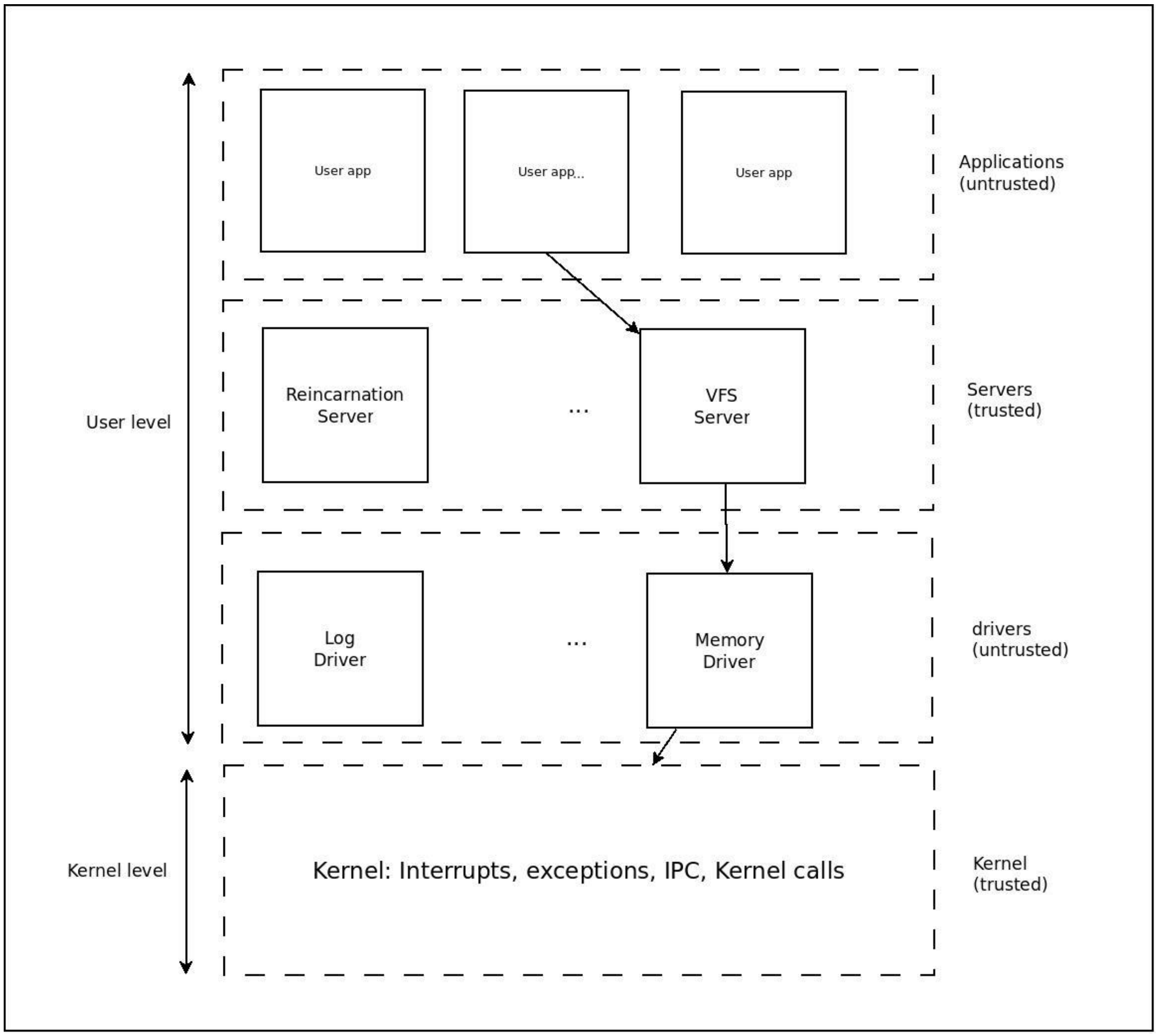

Fig 2. Minix3 system structure

3.2 Processes

Minix3 processes have general process model as Linux kernel. Each processes have three different states: Running, Ready and Blocked. It has hierarchies and can be handled as a group. Remainder of this chapter focuses on those are not common among other kernel.

As for initialization of processes, init is the first user processes. However, a few process gets to run before init. Clock is a kernel process handling system clock. It is invisible outside of kernel. The process manager is the first process to run in user-space. The reincarnation server starts next, it is made the parent of all the other processes started. Though init is not the first user process to run, it is still given PID 1 in Minix3. Init is made a child of reincarnation server.

Message passing is the main inter-process communication mechanism provided in Minix3. There are three primitives provided. They are called by C library procedures [Andrew, S. T., & Albert, S. W. 2006]:

send(dest, &message)

to send a message to dest.

receive(source, &message)

to receive from source, and

sendrec(src_dst, &message)

to send a message and wait from the *src_dst*. When a process sends a message to a process that is not waiting for a message. The sending process is blocked until *dest* does *receive*. If process A has send or *sendrec* message to process B. Then process B is not allowed to do a send or *sendrec* to Process A. Otherwise we have a deadlock. The fourth primitive of message passing procedure is

notify(dest)

This procedure is never blocked. It is used to make another process aware that something crucial has happened [Andrew, S. T., & Albert, S. W. 2006].

The Minix3 scheduler utilizes a multilevel queuing system. 16 queues are defined. An IDLE process is defined and runs in lowest level. The lowest level queue is only used when there is nothing to do. Driver processes usually come in higher priority than server processes. However, system can demotes processes to lower level to prevent faulty processes getting all resources.

Normal process management procedures like *fork(), exec(), exit(),* and *wait()* are supported by Minix system. But the implementation is more complex that other systems. Several servers and kernel maintain several different process tables. These table are private to its own scope and have different information. However, the total number of processes are synchronized between servers and kernel. Therefore, whenever a processes are created or destroyed, its information has to be updated in all process tables, which could be a big overhead.

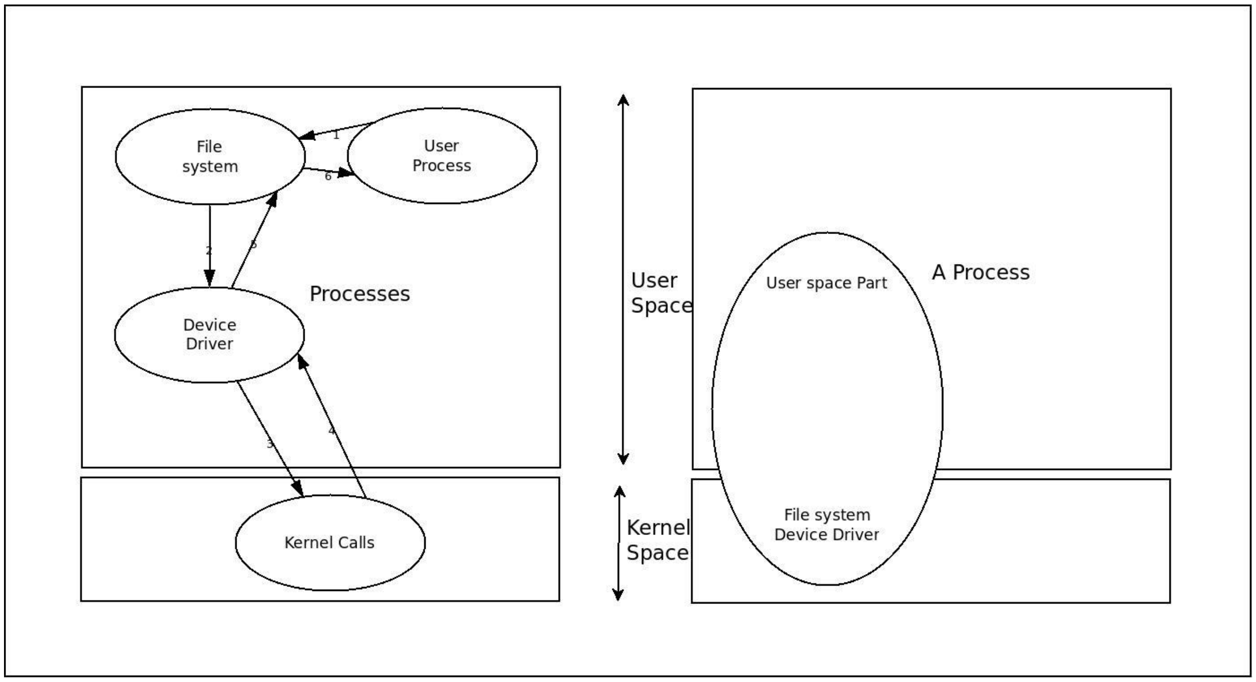

Fig 3. I/O process comparison between microkernel and monolithic kernel.

3.3 Input / Output

There are two types of I/O device driver on Minix3: character driver and block driver. For each type of I/O device, a separate I/O device driver is present. These drivers are independent user-processes. File operations can be performed on these drivers as in Linux kernel. However, people need to be aware of overhead caused by highly modular design. Fig 3 shows difference between procedure path on Minix3 and a monolithic kernel. When a file operation like read, write and open is called, the user process first send a message to file system. File system looks up it filp table to find corresponding filp related to device driver. Then device driver communicate with kernel to perform data copy. 1-6 extra message passing and context switching occur. By contrast, Monolithic system doesn't require any message passing. Kernel procedure are activated by trapping.

To simplify driver development on Minix3, two library, libchardriver and libbdriver are provided. The *transfer* function are used to handle read and write operation. However, file backed mmap has not been supported yet on Minix3.

3.4 Memory Management

Minix3 didn't support paging several years ago. Instead, segmentation was used. The reason for not using paging is that the designer hope it can be easily to be ported to other hardware. But not all hardware devices support paging.

As a new server vm introduced, virtual memory and paging mechanism are supported. Memory management is mainly handled by vm server. There three type of structures defined in vm server: *struct vir_region, struct phys_region* and *struct phys_chicks* [Andrew, S. T., & Albert, S. W. 2006]. The struct vir_region is a contiguous range of virtual address space. A vir_region doen't need to have real memory instantiated in it. The struct phys_region are implemented mainly for supporting memory mapping. This technique is similar to Linux kernel. Due to historical reasons, physical blocks are actually abstracted as struct phy_chicks. This name was used to indicate difference between segmentation and paging mechanism. However, this naming is inherited to current Minix3 release. A chick is of 1024 bytes. Each phys_region has a reference to a list of phys_chicks.

One thing needs to mention is that vm server is not the only place maintaining information about mapping between virtual address and physical memory. Both kernel and vm has separated information of address mapping. Whenever a table is changed, the other table need to be updated.

One of the most unique feature of Minix3 memory management is memory grants. Memory grant enables Minix kernel to monitor data exchange between processes in byte level. To perform data exchange, process has to call kernel to set a grant for the memory region to be exchanged. The kernel maintains a grant table to store the memory grants. Memory copy without valid grant will be rejected by kernel. There are two types of grant in Minix3: direct and indirect. Direct grant is the grant set by owner of the memory region. Also, other processes can forward the grant by indirect grant. Direct and indirect grant combined gives kernel full ability of monitoring memory exchange.

3.5 File System

File system on Minix3 is mainly implemented by vfs server. The vfs server maintain four tables for managing file system. They are block table, inode table, superblock table and filp table [Andrew, S. T., & Albert, S. W. 2006].

Block table is defined in *cache.c* under vfs server directory. It functionality is defined in Table 1.

Procedure	Function
get_block	Fetch a block for reading or writing
put_block	Return a block previously requested with get_block
alloc_zone	Allocate a new zone (to make a file longer)
free_zone	Release a zone (when a file is removed)
rw_block	Transfer a block between disk and cache
invalidate	Purge all the cache blocks for some device
flushall	Flush all dirty blocks for one device
rw_scattered	Read or write scattered data from or to a device
rm_lru	Remove a block from its LRU chain

Table 1. Procedures for block management

Inode table is also needed to support vfs procedures. Many of the procedures are similar in function to the block management procedures. They are listed in Table 2.

Procedure	Function
get_inode	Fetch an i-node into memory
put_inode	Return an i-node that is no longer needed
alloc_inode	Allocate a new i-node (for a new file)
Wipe_inode	Clear some fields in an i-node
free_inode	Release an i-node (when a file is removed)
update_times	Update time fields in an i-node
rw_inode	Transfer an i-node between memory and disk
old_icopy	Convert i-node contents to write to V1 disk i-node
new_icopy	Convert data read from V1 file system disk i-node
dup_inode	Indicate that someone else is using an i-node

Table 2. Procedures used for inode management

The file super.c contains procedures to manage superblock and bitmap. The procedures are listed in Table 3.

Procedure	Function

alloc_bit	Allocate a bit from the zone or i-node map
free_bit	Free a bit in the zone or i-node map
get_super	Search the superblock table for a device
get_block_size	Find block size to use
mounted	Report whether given i-node is on a mounted (or root) file system
read_super	Read a superblock

Table 3. Procedures used for superblock management

File descriptor and filp table are maintained in *filedes.c*. A file descriptor and filp are needed before a file is created. The procedure *get_fd* is used to handle this. The *get_filp* procedure is used to check whether a file descriptor is in range, if so, a filp pointer is returned. The procedure *find_filp* is used to check whether a process is writing on a broken pipe.

3.6 Reincarnation Server

Reincarnation Server (RS) is a special server in Minix3. One of the difference between Minix3 and other modern system is that init process doesn't fork other processes during initialization. All user-level processes are created by RS server. This design enables RS server to keep track of the state of all driver and server process. Once RS server finds any abnormalities. It can either replace the faulty server with a fresh instance or stop the faulty server permanently.

Chapter 4 - Android Design

This chapter provides a description of Android design. I briefly introduce the general architecture of Android system, then discuss each part in detail. In the end, a high-level overview of compatibility of Android system Minix3 is provided.

The remainder of this chapter goes as follow. To start with, Section 4.1 provides overview of Android framework. Section 4.2 – 4.4 describe the Android framework in parts. In the end, Section 4.5 compares Android and Minix3 and explains why porting Android to Minix3 is possible.

4.1 Android Framework Overview

Android system can be viewed in two ways. In a vertical view, Android system can be divided into four parts: Android kernel and driver, middleware, application managers and applications running on the Android.

4.1.2 Software Stacks

Android kernel is a modified version of Linux kernel. The current Android versions includes a kernel based on Linux kernel 3.4. It mainly has the same functionalities as Linux kernel but adding several kernel modules. The most important of them are proprietary drivers: logger, binder and ashmem. Logger is the basis of system log of Android. Ashmem stands for Anonymous Shared Memory. It is introduced as the major shared memory mechanism for Android. Binder is the major inter-process communication for Android. It creates a binder instance in the binder driver. A binder instance can be referenced either a weak reference or a strong reference. A binder instance stands for a pipe to a system process. Once a process gets a binder reference, it can send information through binder driver to communicate with the connected process. All three parts will be discussed further in this chapter.

The second layer is the middleware. It consists of various native components. We will mainly look at three of them: Service Manager, Zygote process and Dalvik virtual machine. Service Manager can be viewed as a naming server of Android. It is running as a user process above kernel, keeping a mapping of service names and their binder references. Other system process gets connection to a specific system service by querying service through the service manager. Zygote is the process holding the actual android runtime. It is initialized by init process and acts as parent process for all other android processes. Dalvik is an instance created by Zygote. It has similar functionalities as Java virtual machine. Every android application has its corresponding Dalvik instance.

The third layer is the application framework. It consists of several service managers. However, they should not be confused with Service Manager we mentioned before. The Server Manager is an independent process initialized with Zygote process. However, service managers in application framework are child processes forked by Zygote process. They are running above Dalvik virtual machines to provide abstractions of system service to upper applications.

The final layer is the application layer. Android applications consists of activity and service. In fact, we can imagine them as foreground and background processes running on Linux. They both are implemented in Java and running on Dalvik.

4.1.2 Program Work Flows

Most of materials only discuss software stacks of Android system. However, what confuse people most are how these layers interact with each other. Here, we generalizes three system paths which can conclude most of Android system activities.

First, HAL stands for hardware abstraction layer. It serves to protect hardware manufacturers' benefits. Because Android is based on Linux kernel, it is under GNU license. If Android includes driver logic into kernel space, hardware manufacturers usually don't want to release it driver source code. But with HAL, manufacturers can define an interface only defining basic input/output to the device and hide actual logic into binary images.

The second path is Activity Components. It consists of Activity, Services and IPC abstraction in Android. However, all IPC abstractions are powered by binder driver which we mentioned earlier. We will discuss this later in this chapter.

The third path is UI. This path is mainly about how surface clients communicate with window manager and finally generate framebuffer. Since this thesis is focusing on preliminary work of porting Android to Minix3 platform. We won't look into this part in detail.

4.2 Proprietary Drivers

Let's first move to proprietary drivers. These drivers are kernel modules provided by Android to support the entire system. Understanding these drivers can help us study upper level framework of Android system.

4.2.1 Logger

Android system provides a light weighted logging system. This system is implemented as driver module in kernel space. To fully support this system, Android provides Java and C/C++ interfaces to work with this driver. Also, Android provide a tools, Logcat, to access the log driver.

To start with, let's look at the logger driver. It is implemented as a Linux driver module. In Android source code tree, its files reside in *kernel/common/drivers/staging/android/logger.h* and *kernel/common/drivers/staging/android/logger.c*. We won't touch the implementation details here, but summarize its general working process. Logger driver is implemented as a character driver, which means we can treat it as a special file and operate it with file operations. It has four minor devices: */dev/log/main*, */dev/log/radio*, */dev/log/events* and */dev/log/system*. When opening a certain minor device, the driver will allocate buffer on kernel memory and return a file descriptor to user. User can perform read/write operation to the log buffer with the file descriptor.

The Java and C/C++ interfaces defines functions we normally use to perform logging task on Android. Java interface performs actual functionalities through C/C++ functions. These two interfaces are connected through JNI (Java Native Interface). In Java interface, we can see different tags corresponding to different logging priorities are defined: info, debug, error and warning. C/C++ performs write operating to driver though *__write_to_log_kernel* function. *__write_to_log_kernel* writes logs to driver in vector format with *writev* function, which should be familiar to Linux developers.

Logcat is actually implemented in a fairly simple logic. It is a process which open logger driver, keep perform *read* operations and print the read information to the terminal. Functionalities for processing of log information are also provided. For example, user can look up logging information through the tag or source name.

4.2.2 Ashmem

Android provides ashmem subsystem to provide anonymous shared memory. It is implemented as a driver module. It maintains a list of unused buffer, which can be released by kernel. Also, it works with binder to realize shared memory functionalities in Android. We will briefly discuss ashmem in following three parts: its abstraction in Android application layer, its driver implementation and how it works with binder to realize shared memory.

First, ashmem's corresponding abstraction in Android application is *MemoryFile* class, which is defined in *frameworks/base/core/java/android/os/MemoryFile.java*. In fact, as we mentioned in previous section, when opening a Linux driver, user gets a file descriptor corresponding to the device he opens. Therefore, the *MamoryFile* actually opens the ashmem driver and stores the file descriptor in its scope.

The ashmem module is implemented by files in *kernel/common/drivers/staging/ashmem.h* and *kernel/common/drivers/staging/ashmem.c*. It can perform open, mmap, read, write, pin and unpin. People may think ashmem is implemented in a complicated way. However, it is fairly simple. When user opens the */dev/ashmem* driver, ashmem create a *struct ashmem_area* structure and stores the structure into the private buffer attached to the opened file. When the user tries to get the shared memory, ashmem calls *shmem_setup_file*, a function provided by Linux kernel to realize the shared memory. Therefore, most of complicated work has been taken care of by Linux kernel. This makes ashmem driver fairly light-weighted. People might be curious about pin/unpin operations provided by ashmem. The ashmem driver divides allocated memory into several *struct ashsmem_region*, these ashmem regions linked as a linked list. The *struct ashmem_area* consists of a pointer to the head of the linked list. This list provides information that which region of the allocated memory are being used by user. User pin a specific region of shared memory by explicitly calling pin function. Kernel can utilize these information to use unpinned region of ashmem when memory are limited.

Finally, we need to look at how shared memory are used in Android. As we can see from previous description, the shared memory is specified by the file descriptor returned when opening the ashmem driver. As for Linux developer, sharing the file table might be easy. When *fork()* is called, the file table is shared between parent and child process. However, in Android, all processes only have to be forked by Zygote process. Then how can processes share the memory? This is where binder comes to play. We will discuss what binder really is in later section. Now, we can just think binder is an IPC mechanism which can pass messages between two different processes. The general process are as follows: process 1 get the binder reference through Service Manager to build connection with process 2. Then process 1 copy the file descriptor to binder and send it to process 2. When the binder processing the file descriptor copy, the binder will know it is handling a file descriptor. Therefore, binder will copy the file structure specified by the file descriptor to process 2. This is similar to copy one entry of file table to another process. Therefore, process 2 gets access to the shared memory created by process 1.

4.2.3 Binder

All Android IPC mechanisms are actually realized by binder driver in the end. People may have used bundle or intent in Android development, these class are just several different abstraction of binder driver in application layer. Therefore, binder is one the most important system component in Android, also the most complicated one. It is glue which sticks different Android component together. When people studying Android binder, they tend to be confused by its hierarchical abstractions and complex call processes. This section will mainly focus on how binder differs from other IPC mechanism and how this is realized.

IPC	Number of Copy
shared memory	0
binder	1
socket / pipe / message queue	2

Table 4. IPC mechanism and number of copy

Linux kernel has already provided many IPC mechanism, including shared memory, socket, pipe, and message queue. In Table 4 we generalize the major different between binder and these IPC mechanisms. The biggest difference is the times of data copy in a single transaction. Shared memory doesn’t need data copy. However, it is hard to control and trace. As for socket, 2 times of data copy, from user-space to kernel and from kernel to user-space, are too much cost for a mobile device. But socket mechanism imposes a client-server model in IPC, which secures the transaction and easier to control the system. To make the advantage of client-server communication model while reducing times of data-copy, Android introduces binder mechanism. Binder enforce the client-server communication model between processes, while only requiring one time data-copy for each transaction.

How this is realized? To figure this out, we need to look into binder driver in detail. The basic operation binder driver provides are read/write. The data copy only happens when write is called. Fig 4 illustrates how binder driver works. As indicated on Fig 4, when a certain process opens binder driver, it create a binder instance in binder driver. Once the binder instance is created, the binder driver does following operations: first, it creates a buffer for this binder instance, and map this buffer to the address space of the calling process. Then binder driver stores the binder instance into its internal map and return a strong binder reference to the calling process. When other process wants to send information to this process, it calls write operation along with the reference to the binder instance created by the target process. Then the binder driver uses *copy_from_user* to copy data from sending process to the buffer it creates for the target process. Because this buffer is shared between binder driver and target process, not further copy operation is required. Finally, binder driver notify the target process to accept the sent data.

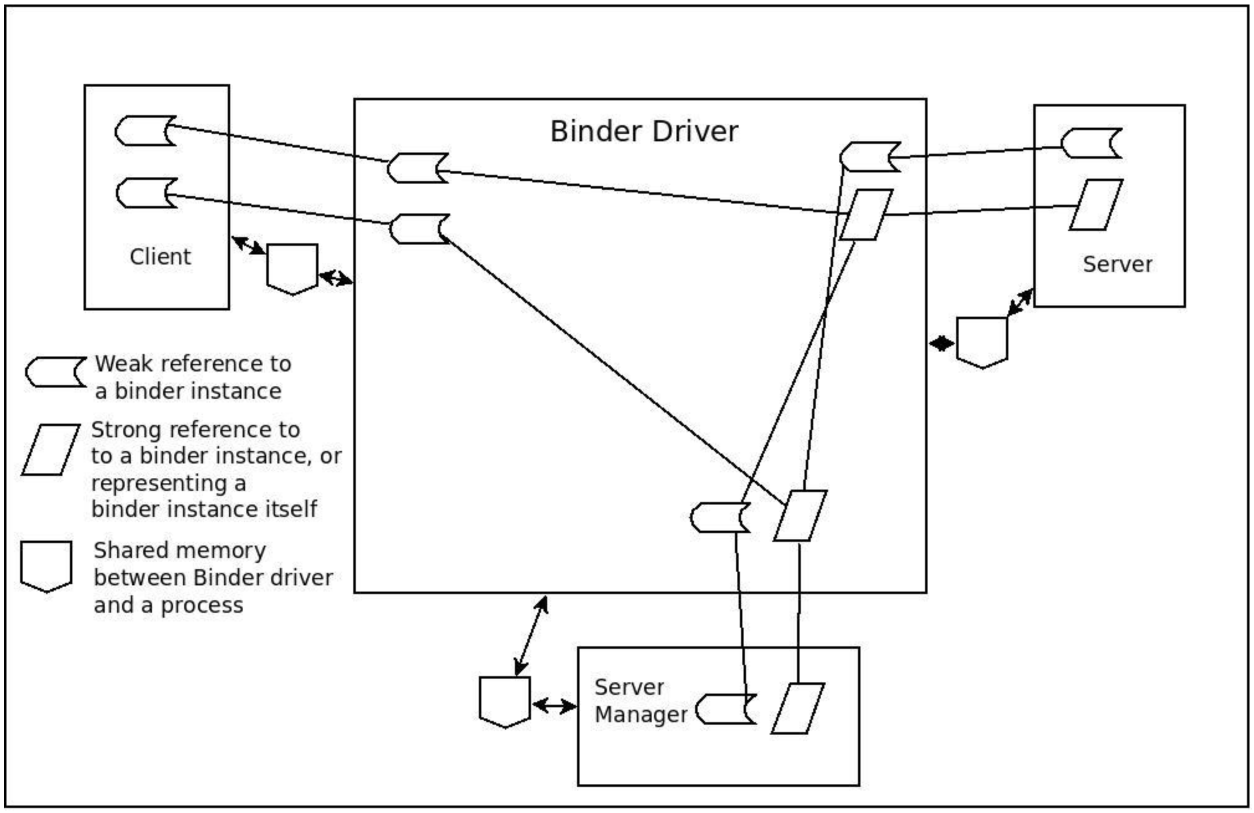
Binder Driver
Client
Server
Server Manager
Weak reference to a binder instance
Strong reference to to a binder instance, or representing a binder instance itself
Shared memory between Binder driver and a process

Fig 4. Illustration of Binder driver structure and working process

The next question is how process get the binder reference to another process. This done by Server Manager we mentioned earlier in this chapter. Server Manager is working as a name server in Android system. Whenever an Android service is created, it has to register itself to the Server Manager. The Server Manager store the mapping between service name and it binder reference in its scope. Other process can communicate with Server Manager through binder driver to get binder reference of server. The binder reference of Server Manager is defined as a unique global value 0 once the Android system starts.

There are last two things needs to discuss about binder. First, difference between weak binder reference and strong binder reference. As shown on Fig 4.2.3.1, only the creator of binder instance owns the strong reference to the binder. The binder reference which other processes get is only weak reference. This difference enables binder driver to impose different access control policy to different owners of binder reference. The second thing is the anonymous binder. Though processes can get binder reference through Server Manager. Not all binder has to get register to the Server Manager. Once a binder connection has been built between two processes. They can create private binder and send the reference through the existing binder connection. Then, this binder connection is private to this pair of processes. This mechanism enhances security of IPC communication in Android system.

With the introduction of binder mechanism, Android can finally utilize client-server model. This is a crucial feature, because Android can finally builds up a multi-server-like system above Linux kernel. The zygote can monitor all Android processes and kills faulty system components when needed. This feature improves the dependability of the entire Android system.

4.3 Application Components

Application components are core of the Android system. These components can be divided into four types: Activity, Service, Broadcast Receiver and Content Provider. To facilitate these components, there three modules including application thread, looper and installation. The remainder of this section briefly summarize features of these components.

4.3.1 Activity

Activity and Service is the basic component for Android application. They combined to build a complete application. Android system provide a complete mechanism to start these Activity and Service. Also, these components can communicate through binder mechanism we introduced earlier in this chapter.

In Android system, there are two ways to start Activity, one is clicking icon of application. The other is to start *subActivity* inside the Activity. Since this thesis focus on preliminary work of porting Android, we will focus how Zygote process generate these Activity.

As the Activity is triggered. Launcher or other Activity communicates with *ActivityManagerService* to start an Activity. Here, *ActivityManagerService* is an Android service generated by Zygote as Zygote starts.

Then, *ActivityManagerService* pauses caller component through binder. As the caller component gets into paused state. The *ActivityManagerService* create a new process to start a *ActivityThread* instance. This new process is created by Zygote process. The *ActivityManagerService* can notify *Zygote* to create a new process.

As *ActivityThread* starts, it creates a binder instance and pass the reference to *ActivityManagerService*. Then *ActivityManagerService* can communicate with *ActivityThread* through this binder reference.

Finally, *ActivityManagerService* notify *ActivityThread* to start Activity.

4.3.2 Service

Starting Service component is basically the same as Activity. The main difference of Service is that Service usually running in an independent Android process. Here, we briefly go over the process of starting Service and binding Service.

As for starting Service, the caller process will communicate with *ActivityManagerService* through binder to create a new process. Then the new process creates a binder and pass the binder reference to *ActivityManagerService*. The *ActivityManagerService* then start the Service process through its binder reference. The general process is similar to starting Activity.

When an Activity needs to bind a Service. It first communicate *ActivityManagerService* through binder to request a specific Service. The *ActivityManagerService* request the Service with *onBind* function to get binder reference. After *ActivityManagerService* gets the binder reference, it pass this reference the caller Activity. Finally, the Activity build connection with requested Service the binder reference.

4.3.3 Broadcast

Broadcast is a message passing mechanism provided by Dalvik. It has few relation with binder driver. On the other hand, it is closer to Message-Driven Bean provided by J2EE. Since this thesis won’t focus on actual implementation of Dalvik virtual machine. We won’t discuss actual work process of Broadcast.

However, we can briefly summarize difference between binder and broadcast. As mentioned in previous sections, to communicate through binder, a process needs to get a reference to a binder instance of the target process. However, in Broadcast, the receiver doesn’t even have to know the sender of the message. It only needs to listen a certain type of message on a specific port. Broadcast brings more flexibility of message passing for processes in Dalvik virtual machine.

4.3.4 Content Provider

Content Provider is another type of abstraction of binder IPC provided in Android system. The reason of designing Content Provider is due to various data input for a mobile devices. As for a mobile device, data source can be internet, database or local files. As data size increases, the complexity of handling different type of data source will also increase. To provide a better interface for data handling, Content Provider is introduced to hide the complexity data I/O from Android application

The actual data transfer between application and Content Provider is still through binder interface. However, there is another problem. Data from Content Provider is usually large. In this case, binder mechanism is not efficient anymore. This is where ashmem driver can help. As described before, processes can share file descriptor to a share memory through binder driver. This is what exactly happen in Content Provider. An application creates a shared memory, and pass the descriptor of this memory to Content Provider. Then application and transfer large chunk of memory through shared memory.

4.3.5 Zygote and Process

Zygote process, as the meaning of its name, forks all application process and system server process in Android system, except middleware native service. This sections will explains how Zygote process are generated and how it forks other process.

Zygote process is created by init process, same as most of processes in Linux. The starting process is defined in *init.rc* file. After Zygote process is created, it starts to initialize Android system and forks a process called *SystemServer*. *SystemServer* continues starting other system services including *PackageManagerService* and *ActivityManagerService*. However, *SystemServer* doesn’t fork these processes by itself. It communicates with Zygote process to fork these system servers.

After Zygote process finishes initializing the Android system. It waits for messages from *ActivityManagerService* to fork process for Activity or Service.

4.3.6 Messages and Looper

Though Android IPC mechanism is primarily based on binder and ashmem driver. The message handling inside Android is different. When a server finishes a request, it sends a reply back to client. However, in Android system, this reply message is not directly passed to client. Instead, it push the message into a message queue of the client process. Each application or system component in Android has a message queue. Each application can have a thread waiting under an infinite loop to handle its message queue. And the Looper is the abstraction of the infinite loop and message queue.

4.4 Compatibility with Minix

After analyzing Android and Minix3 design, we can easily realize that both systems share some similarities. Also, differences also exist. This section mainly analyzes compatibility of Android with Minix system.

4.4.1 Similarities

The biggest similarity that Minix3 and Android share is the multi-server design. As discussed above, Android can be viewed as multiple services and activities running above the Linux kernel. Android achieve this design by introducing a new IPC mechanism, binder. With binder, Android imposes a client-server model in system IPC. Each process has to have a thread waiting on an infinite loop for binder request. As for Minix3, it is inherently a multi-server system. This shows that Minix3 and Android designers come to an agreement that client-server model is a safe choice for inter-process communication.

Second, both Android and Minix3 has separate process running as a naming server. In Android, all activities and services have to register themselves on Server Manager when initialization. To get service from a specific server, the client must query the Server Manager to get the reference of a binder owned by the target server. In Minix3, Reincarnation Server offers nearly the same functionality as Server Manager. In Minix3, all programs are running as independent processes above kernel. To get build communication with a target server, a process must get the endpoint of the target server. This endpoint is generated and stored on Reincarnation

Server once a server gets started. In both design, the naming server provides a verification ability. Only verified service or activities can be accessed. This naming server can also easily create an access control list to restrict accessibility of certain service. This kind of possibility provides potential for high dependability and security demand.

4.4.2 Differences

One of the differences of Android and Minix3 design is the overhead of communication. Though adopts a multi-server design, Android system still runs above a Linux kernel. As analyzed in Chapter 3, accessing driver module in monolithic system is fairly cheap. User-level process can call kernel-procedure by trapping. No message passing is needed. However, in Minix3, a driver operation may take up to 6 messages. Without violating microkernel design, it is impossible to reach the same level of performance as Android has on driver accessing. Since Android system makes large use of binder driver, this overhead can make big differences in porting.

The second difference is the restriction on data copy between processes. On Android, a *copy_from_user()* call can be done with one step. However, on Minix3, the designer enforces tight restrictions on data copy between processes. To copy a data from another processes, the user must ask kernel to set up a grant from kernel, send the grant to the target process, and finally target process copy the data from kernel with the grant. This copy process costs 6 message passing and context switching, which is huge compared with Android.

Chapter 5 - Experiment

This Chapter describes the preliminary work I have done in port Android to Minix3 platform. Section 5.1-5.2 summarizes porting method and implementation details of porting propriety drivers and prototypes of Zygote process. Section 5.2 talks about modification I made in kernel. Section 5.3 presents performance measure and analysis of an IPC experiment.

5.1 Porting Proprietary Drivers

In this section, I presents porting method I used in porting these drivers. Also, I explains some trade-off in porting.

5.1.1 Logger

The logging system consists of three parts: logger driver, system library and Logcat service. To start with porting, we need to find their spots in Minix3 system. In fact, they are just exactly the same in Mini3 as they are in Android.

The logger driver's functionality can be easily ported in Minix3. Supported by libchardriver provided in Minix3, logger driver can be easily implemented as a character driver in user-space. The logger driver source file is located in log folder under drivers directory. It initializes four minor devices: */dev/alog/main, /dev/alog/radio, /dev/alog/events, and /dev/alog/system.* The alog is specified to indicate that they are ported Android log devices. The operation to logger driver in Minix3 is basically the same as Android from user point of view. However, it should always to keep in mind that file operation on Minix3 may consists of up to 6 messages and context switching as overhead.

The library handling logger driver in Android is implemented in Java with a few native functions. They are ported in Minix3 in C version. Functionalities are the same. The function calls are listed in Table 5.

Procedure	Function
d	Send a DEBUG log message
e	Send an ERROR log message
i	Send an INFO log message
v	Send a VERBOSE log message

w	Send a WARNING log message
log_println	Write one line to logger driver

Table 5. Interfaces of log library for writing log message.

The Logcat tool is implemented as a server on Minix3. Once started, it is running as a server process in user-space. This is where functionalities are compromised during porting. The Logcat tool consists of extensive functionalities, including formatting output and rotating output log entries. Since this thesis is more focused on providing prototypes for Android porting on Minix3. Logcat on Minix3 only implements basic feature of pulling log entries from logger driver. As project moves on, more functionalities can be extended with current design.

5.1.2 Ashmem

The ashmem on Android mainly contains two parts: ashmem driver, defined in *ashmem.c* and *ashmem.c*, and system library, defined in *ashmem-dev.c*.

Defining counterpart of ashmem driver in Minix3 of ashmem driver is not as easy as logger. Android implements ashmem as a driver module mainly for two reasons: first, to fully utilize kernel functionalities. Inside kernel space, ashmem can directly call several kernel procedure to realize kernel memory mapping. Second, to follow the mmap and file descriptor pattern. These two benefits on Android become limitations for Minix3 implementation. Normally, when requiring a shared memory, Android user makes following calls:

int fd = ashmem_create_region("name", size);

*void *addr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);*

Here, mmap function does a file-backed memory mapping to get the buffer of allocated file structure. However, since file structure are implemented with a different filp structure, which doesn't contains a buffer, file-backed mmap is still not supported on Minix3. This pattern cannot be followed. In addition, after moving driver modules, each system call inside inside driver module can cost at least one message passing. Also, Minix3 imposes several developing restrictions on driver to enhance its dependability. All these things combined make the ashmem no longer suitable as a driver module on Minix3.

Instead of driver process, porting ashmem as a server process can be easier. It has more flexibility in developing on Minix3. Also, by directing communicating with a stand-alone server process, about 4 message passing can be saved compared to driver implementation. However, with limited file-back mmap support on Minix3, a compromise has to be made. In current design, mmap is replaced by a new system function *ashmem_mmap* with same input parameters. With this type of design, we can preserve original interface to biggest extents. Once file-back mmap is fully supported, update can be easily done by a simple typdef inside *ashmem-dev.c*

To sum up, as for ashmem port, ashmem is now ported as a server process. A new *ashmem_mmap* is defined in *ashmem-dev.c* file. A list of procedures is provided in Table 6 and Table 7.

Procedure	Function
ASHMEM_SET_NAME	Set name of a memory area
ASHMEM_GET_NAME	Get the name of a memory area
ASHMEM_SET_SIZE	Set size of a memory area to be allocated
ASHMEM_GET_SIZE	Get size of a memory area
ASHMEM_SET_PROT_MASK	Set protection mask of a memory area
ASHMEM_GET_PROT_MASK	Get protection mask of a memory area
ASHMEM_PIN	Pin a region of a memory area
ASHMEM_UNPIN	Unpin a region of a memory area
ASHMEM_GET_PIN_STATUS	Get pin status of a region of a memory area

Table 6. System calls provided by ashmem server in Minix3

Procedure	Function
ashmem_create_region	Create a shared memory, return the id associated to it
ashmem_set_prot_region	Set the protection mask to the memory area
ashmem_pin_region	Pin a region of a memory area
ashmem_unpin_region	Unpin a region of a memory area
ashmem_get_size_region	Get size of a memory area

Table 7. Procedures provided to manage ashmem

5.1.3 Binder

The interface of binder driver is fairly simple. However, as we discussed in previous chapter, the underlying design of binder is profound. This reflects on its code size and complexity. As we analyze the source code, *binder.c* has around 3665 lines of code, which is the most of three drivers.

However, a large amount of functions defined in *binder.c* are working with a waitequeue defined in each binder instance. The reason for incorporating a waitqueue in Android binder is because that Linux kernel is not a multi-server system. To working with Android running above binder driver, the binder has to find implement some message handling mechanism. When porting binder to Minix3, because Minix3 is inherently a multi-server system. Once we port binder as a certain system component, message handling has already taken care by kernel. Therefore, we can drop all code doing message handling inside *binder.c*

Benefited from Minix3's design, we can focus rest of binder code and decide how we position the binder driver into Minix3 system design. The binder is implemented as a driver module. The reason for this is to let binder driver get full access to kernel data structure. In fact, the memory mapping used in Android binder is done by directly working on *struct vm_area_struc*t and *struct vm_struct*, people familiar with Linux kernel development may realize these are just basic virtual memory structure for user-space and kernel. As we discussed in previous section, driver on Minix3 is just a user process. In addition, Minix3 has enforced many restrictions for driver developers. If we are not working with real device, implementing system service in driver process doesn't give us any benefit. Therefore, binder should be implemented as a server process on Minix3.

Since memory mapping between different processes have been supported on Minix3, the rest of binder porting can be really easily. For every write operation, one data need to be performed. Remember, data copy in Minix3 is expensive. Here, we can introduce a new kernel procedure, without modifying too much of kernel code, to get performance optimization for the binder. This new kernel function is called *bindercopy*, which can directly perform data copy with virtual address. Performance difference will be measured in later chapter.

To sum up, binder can be ported as a server process. Due to Minix3's multi-server design, message handling code in *binder.c* can be saved for porting. The kernel acceleration can be provided for binder server.

System calls provided by binder server is listed in Table8.

Procedure	Function
BINDER_WRITE_READ	Perform write and read operation

BINDER_SET_CONTEXT_MANAGER	Set context manager, which is Server Manager.
BINDER_VERSION	Return the version number of binder

Table 8. System calls provided by binder server in Minix3.

5.2 Zygote and Process Initialization

Zygote in Android has been discussed in previous chapter. It is a process forked by init process. Then Zygote is responsible for generating all Android system processes.

The actual implementation of *Zygote* process on Minix3 might be a lot of work. The Zygote code is tightly combined with Dalvik virtual machine, which actually falls out of the focus of this thesis. However, we can still discuss a prototype for Zygote process before we port Dalvik virtual machine.

Different from other system, init process on Minix3 doesn’t fork system processes after initialization. This work is handled by Reincarnation Server. Therefore, our Zygote should works the same way. The current proposed Zygote prototype is an independent server process. To keep unity of system design, Zygote still listen on service fork request. However, it acts as a forwarding server, which forwards these forking request to Reincarnation Server. With this kind of mechanism, Zygote can still listen and control process creation of all Android processes. Also, Reincarnation Server maintains its ability of monitoring the entire Minix3 system.

5.3 Kernel Modification

A new kernel call is added to provide acceleration for binder server. The interface in kernel is like following:

*Int do_bindercopy(struct proc *caller_ptr, message *m_ptr);*

The corresponding system library interface is:

Int sys_bindercopy(endpoint_t dst_t, endpoint_t src_t, vir_bytes dst_addr, vir_bytes src_addr);

The kernel call provides ability of directly copying data between two processes with their virtual memory. The kernel grant system are avoided. The downside of this function is that it breaks the dependability in some level. To keep this leak small, *sys_bindercopy* is only allowed to be used by binder server. Otherwise, error will be returned, no memory copy takes place.

5.4 Performance Measure

For measurement, a PC containing Intel i7, 16 GB memory was used. Two user-level process running in different address spaces. One is a client process and the other serves as server. Client process repeatedly performs request to server through binder. Server always replies to clients and process its requests. 100000 requests are sent and time elapsed is recorded.

As for Minix3, the binder is running as a server process. On the Android, binder is located in kernel as a driver module. Three instances of operating system are tested. Two are Minix3 with proprietary drivers ported. One of them uses newly-added kernel function as acceleration. As for Android, Android-x86, a forked Android version is used. The reason of using Android-x86 is to let three instances run on the same device.

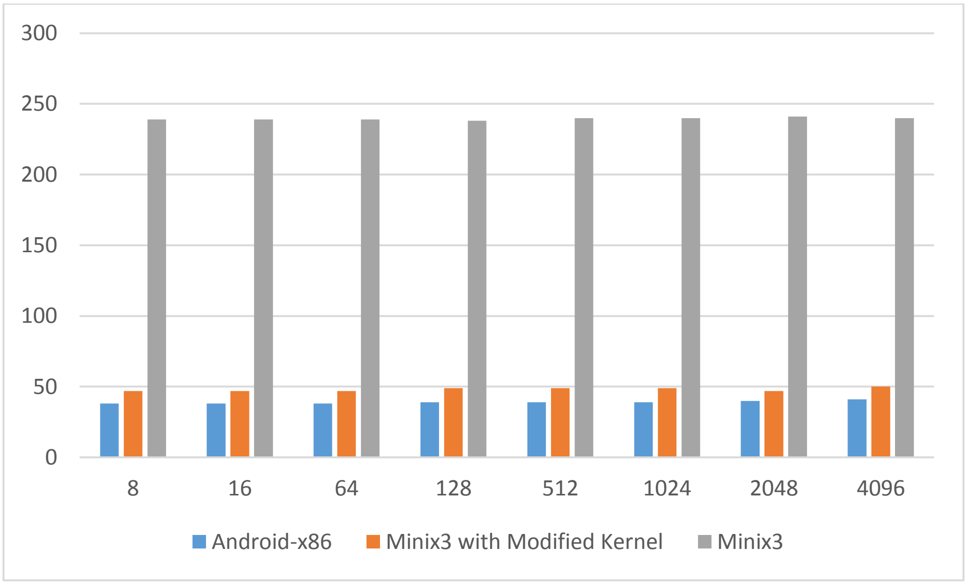

Fig 5. Performance measurement with read-write through binder

As we can see, Minix3 without any kernel acceleration performs worst. Kernel functions brings about 75% improvement in performance. The Minix3 with kernel acceleration achieves almost same performance as Android-x86. This proves that, the biggest hinder of microkernel system is not the message passing itself, but the communication complexity brought be modular design. In original Minix3 kernel, data copy has to go through memory grant system, which takes 8 more messages during on binder transaction.

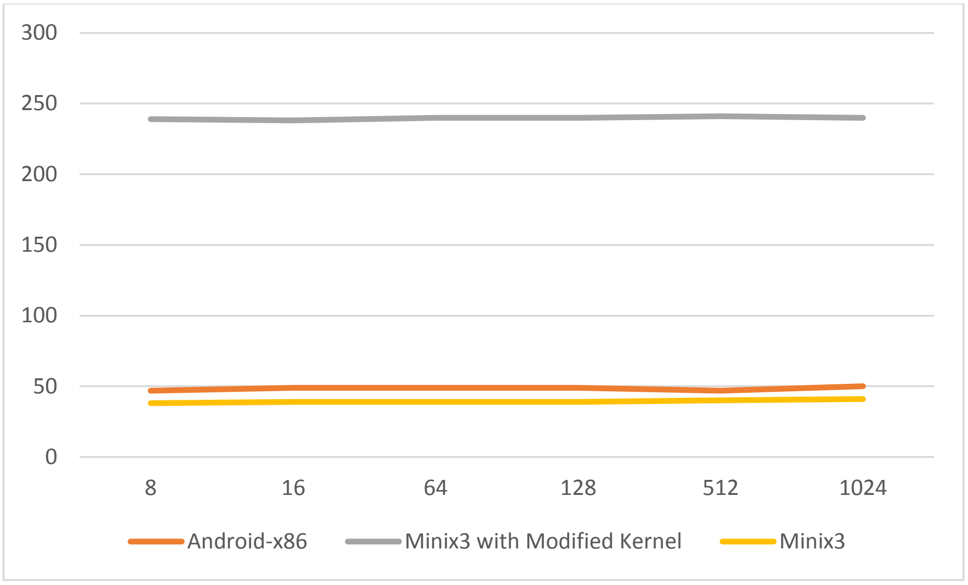

Fig 6. Performance measurement with read-write through binder

Another thing need to notice is that performance doesn’t fall with size of transaction, as shown on Fig 6. However, this reflects that on modern system, one-page data is almost the same as several bytes. As for larger size data, the Android will utilize ashmem instead to handle. The binder only needs to care about smaller data.

Chapter 6 - Future Work

This thesis provides prototypes of dividing Android system and porting them to Minix3 platform. Also, a preliminary work of porting proprietary drivers is presented. To make Android applications running on Minix3, there are still a large amount work to do. In this chapter, we will points out those components which can be important for future porting.

The remainder of this chapter start with Dalvik virtual machine. Section 6.2 explains hardware abstraction layers.

6.1 Dalvik Virtual Machine

Dalvik virtual machine is one of the most important component in Android. It connects Java interface and native system service, creating an efficient and powerful Android system. Dalvik must be studied for future porting.

Dalvik virtual machine is quite similar to Java virtual machine. The biggest difference between Dalvik and JVM is that Dalvik's instruction set is register-based. On the other hand, JVM uses stack-based instruction set. Besides that, Dalvik provides functionalities same as JVM, including memory management, garbage collection, just-in-time compiling, Java-Native-Interface and process management.

Zygote works closely with Dalvik virtual machine. As discussed previously, Zygote creates all processes Android system needs. Once a new process is created, a Dalvik instance is also initialized. After initialization work done by Zygote, the initialization process will be continued by Dalvik instance.

6.2 Hardware Abstraction Layer

Hardware Abstraction Layer (HAL) is used by Android system to seal the device driver implementation and provides interface to upper layers. There are two major benefits for using HAL. One is to provide a standard for those who want to port Android to their devices. The other is that HAL resides in user-space. Since Linux kernel is under GNU license, hardware manufacturers probably don't want to open all of their source code. In this case, HAL, which is under Apache license, is right place for manufacturers to implement functionalities for their products.

HAL has to be properly designed and ported in Minix3. Because it provides a uniform interface for software layer running above. Big modification of its interface will lead to a huge amount of rewriting work in later porting. A porting method to keep unity of interface between HAL and upper system has to be found.

Chapter 7 - Summary

This chapter puts an end to this thesis by summering the research and highlighting lessons learned. The central question of this thesis is that whether it is possible to run a modern mobile operating system on a microkernel system. As a special case, Android system is studied. This thesis analyzes Android system part by part, and provides preliminary work of porting Android lower level services. Though it still needs a lot of work to fully support Android system, they can be based on work and prototypes provided in this thesis.

Section 7.1 restate the problem and summarize methods used. Section 7.2 presents lessons learned during this project.

7.1 Statement of the Problem and Summary

In this thesis, we have researched whether a modern mobile operating system is possible to run on a microkernel. Android has been the most popular operating system running on smart devices. However, all of existing Android and its variations are based on a uniform Linux kernel. Therefore, Android system is selected as a special case to study. A monolithic kernel design is not suitable of demand on high dependability. To improve system's dependability, Android utilizes many mechanism in its design to control system components' ability on accessing system resources and services. Therefore, we try to port Android to a microkernel system, and see whether a microkernel can simplify design of Android system and open up those restrictions.

We start with porting proprietary drivers on Android system. These drivers are logger, ashmem and binder. After fully studying the design of Android system, we find that these drivers support most of functionalities of Android. Most of system components Android provides are different abstractions for these drivers. Therefore, properly porting can provide a solid foundation for future porting.

Both binder and ashmem are implemented as server processes on Minix3 to fully utilize Minix3's functionalities. The logger stays the same as driver. We also offers a prototype of Zygote process. It is designed as a server process forwarding requests to Reincarnation Server in Minix3. To accelerate binder's performance, a new kernel call is added in Minix kernel. This brings 46% improvement of overall performance.

7.2 Lessons Learned

In this section we present some important lessons we learnt in this project.

First, a multi-server system design simplifies implementation. Android system tries to realize multi-server system based on Linux kernel. It introduces binder driver to achieve this goal. The binder driver puts many work on implements its message handling mechanism. However, with inherently following multi-server system design, the Minix3 kernel provides well support for message handling. The code complexity in binder is reduced on Minix3.

Second, a highly modular system design brings huge overhead. As we see in performance measure, communication overhead is the biggest challenge hindering Minix's performance. Certain functionalities are achieved by several system components. Therefore, a highly modular design can bring high complexity in communication between system components. When considering improving performance of a microkernel system, reducing communication complexity between system components should be one of high priority.

References

Herder, J. N. (2010). Building a dependable operating system: fault tolerance in MINIX 3.

Andrew, S. T., & Albert, S. W. (2006). Operating Systems-Design and Implementation.
Liedtke, J. (1994, January). Improving IPC by kernel design. In *ACM SIGOPS Operating Systems Review* (Vol. 27, No. 5, pp. 175-188). ACM.

Liedtke, J. (1995). *On micro-kernel construction* (Vol. 29, No. 5, pp. 237-250). ACM.
Lackorzynski, A. (2004). L4Linux porting optimizations. *Master's thesis, Technische Universitat Dresden*.

Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., & Peter, M. (2011, October). L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (pp. 39-50). ACM.

"Develop | Android Developers." *Develop | Android Developers*. http://developer.android.com/index.html, Web. 01 Apr. 2014.

"Welcome to the Android Open Source Project!" *Android Developers*. https://source.android.com/, Web. 01 Apr. 2014.

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., & Young, M. (1986). Mach: A new kernel foundation for UNIX development.

Engler, D. R., & Kaashoek, M. F. (1995). *Exokernel: An operating system architecture for application-level resource management* (Vol. 29, No. 5, pp. 251-266). ACM.

Ganapathi, A., Ganapathi, V., & Patterson, D. A. (2006, December). Windows XP Kernel Crash Analysis. In *LISA* (Vol. 6, pp. 49-159).

"Mobile Device OS." *ABI Research*. https://www.abiresearch.com/market-research/product/1015462-mobile-device-os/, Web. 01 Apr. 2014.

Coverity Inc. Coverity Scan 2010 Open Source Integrity Report. http://www.coverity.com/html/press/coverity-scan-2010-report-reveals-high-risk-software-flaws-in-android.html, 2010.

Härtig, H., Hohmuth, M., & Wolter, J. (1998, September). Taming linux. In*Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART'98)*.