Faculty of Computer Science, Electrical Engineering and Mathematics

Department of Computer Science

Research Group Design of Distributed Embedded Systems

(Prof. Dr. Franz Josef Rammig)

Bachelor thesis

Concurrent shared memory access for Android applications and real-time processes

Author:

Marvin Damschen

First reviewer:

Dr. Simon Oberthür

Second reviewer:

Prof. Dr. Uwe Kastens

In cooperation with:

Dr. Wolfgang Mauerer,

Siemens AG

September 12, 2012

Abstract

This thesis aims to discuss and enhance AndroitShmem, the proof-of-concept shared memory service for Androit. Androit itself is an effort to make the Android operating system real-time capable and fulfil industrial needs in terms of usability and determinism.
The first two chapters give an overview of Android and its potential in the industry when extending it with real-time capabilities. Android Framework parts and Android's Linux kernel extensions involved in this thesis' improvements to AndroitShmem are explained. Androit and AndroitShmem are comprehensively introduced.
The third chapter compares ashmem with a memory-mapped file in tmpfs with regard to allocating shared memory in the real-time Android context. Shared memory allocation using a memory mapped file in tmpfs is implemented in AndroitShmem. Furthermore, its performance is shown to be equivalent to using ashmem and considerably better than the previously used primary storage.
The fourth chapter profoundly introduces possible synchronisation techniques for AndroitShmem and illustrates the problems of locks. A concept for synchronising client access to the shared data using transactional memory is explained and implemented. The implementation is then analysed and formally verified using the SPIN model checker.
In conclusion, this thesis takes AndroitShmem from its proof-of-concept state to being practically usable and thus furthers the efforts to make Android real-time capable.

Declaration of Authorship

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text and that this work has not been submitted for any other degree or professional qualification except as specified.

Paderborn, September 12, 2012

Marvin Damschen

Contents

List of Figures

List of Tables

1. Preface

1.1. Motivation

Android is the number one mobile platform for smartphones and tablets today. Its rapid rise from its first stable release in September 2008 to becoming the top-selling operating system for smartphones took just about two years. [21] The consistent look and feel along with a sophisticated software development kit seem to please end-users as well as app developers like numerous available apps and steep download figures suggest. [22]

Android is more than just another operating system for smartphones and tablets, it is a software platform on top of Linux, giving you extensive possibilities to built your own connected device. Android is open sourced, but its licensing model is also suitable for commercial organisations which want to keep their Android developments closed (this does not apply to the Linux kernel). [14]

The conjunction of the Linux kernel with an easy to use and program application platform makes Android an interesting platform for embedded devices of all kinds. Linux provides a portable and mature system core, as the Android platform on top of it allows efficient development cycles of easy to use applications. This thesis will range in the exploration of using Android in an industrial embedded domain.

When using Linux on an embedded system in an industrial domain, real-time capabilities are crucial. Android was not created with mechatronic systems or real-time applications in mind, however it is possible to patch certain versions of the Linux kernel supplied with the Android source to make it real-time capable. [13] The userspace (non-kernel part) needs additional features to let Android applications interact with real-time Linux processes. One main feature needed is data exchange.

1.2. Current status

Mauerer et al. [13] developed an appliance enabling basic data exchange between real-time processes and Android applications called AndroitShmem. The Linux kernel running at Android's core was previously extended by the 'preempt_rt' patchset to feature real-time processes. AndroitShmem currently consists of a shareable data structure, a server process providing access to the shared data structure, an example client written in C and a library to be used in Android applications.

The shared data structure consists of variables of simple types only at this point. More advanced data structures like linked lists are also possible, but require sophisticated synchronisation mechanisms between multiple processes accessing the data structure.

The AndroitShmem Server uses a file which is then mapped into memory to store the shared data structure. Currently, the file to be mapped into memory is stored in secondary storage. The communication between the server and the clients, as well as the mapping of the pointer to the data structure into processes accessing it, is done through Binder. Binder is the Inter Process Communication and Remote Method Invocation mechanism provided by Android. Basically every interaction across multiple processes is done with it. [7]

The library consists of a C file implementing functions which are bound to Java methods using the Java Native Interface and Android Native Development Kit. This way Android Java applications which use AndroitShmem are still easily creatable and can use the shared data structure in an efficient way.

AndroitShmem is basically a proof of concept at this point. It provides the basic implementation needed for data exchange between real-time and non-real-time clients, but lacks a sophisticated synchronisation (although some synchronisation parts are implemented). Furthermore, storing the shared data in a file in main memory instead of secondary storage could lead to considerable performance improvements.

1.3. Goals of this thesis and structure of this document

The first goal of this thesis is to improve AndroitShmem's performance by replacing secondary storage used for the shared data by primary storage (Chapter 3). The possible approaches for this improvement will be explored. A concept will be developed and implemented. Its effects will be measured and compared to the current implementation.

Secondly, this thesis aims to introduce a sophisticated synchronisation mechanism for processes

sharing memory through AndroitShmem (Chapter 4). It will take a look at basic mechanisms which enable synchronisation between processes sharing data e.g. atomic operations, spinlocks and semaphores. The knowledge of these mechanisms will be used to discuss non-blocking algorithms. The synchronisation mechanism proposed in [13] will be used to develop a concept of a synchronisation mechanism for AndroitShmem. This concept will be implemented and its results thoroughly analysed. A formal verification of an equivalent model of the synchronisation mechanism will prove its correctness.

As an introduction, Chapter 2 will explain fundamental notions and concepts of Android, real-time systems and previous works by Mauerer et al. [13].

Chapter 5 will summarise the work done in this thesis and point out open topics to improve AndroitShmem and real-time Android as a whole.

2. Introduction

This chapter will introduce concepts and terminology used throughout this thesis. Additionally, it will describe the current state of the technologies this thesis aims to improve.

2.1. Android

Android [2] is a software platform built by the Open Handset Alliance[1]. It relies on the Linux kernel as an abstraction layer between hardware and software, as well as core system services like process management and drivers. It includes an operating system, middleware and key applications. It is optimised for mobile devices.

Android makes it easy for developers to built feature rich applications, as they can use the same APIs[2] as core applications[3] do. Applications can publish their functionality and make it usable to other applications. Furthermore, functionality provided by core applications can be replaced by third-party applications, making Android highly customisable.

Android treats core applications and third-party applications equal to give developers as much liberty to extend the system as possible. This also means that if you want an application to be treated special e.g. let it stay active even in low memory situations, you have to do it explicitly and take special care.

2.1.1. Dalvik Virtual Machine and Zygote

Android applications are mainly written in the Java programming language. They are compiled to Java Virtual Machine compatible bytecode before being converted to *Dalvik Executables* [16]. The Dalvik Executable format is specifically designed for memory and processor constrained systems. Dalvik Executables are run in the *Dalvik Virtual Machine* [16], Android's own virtual machine to run the vast majority of Android applications.

[1] Founded by Google, `www.openhandsetalliance.com`

[2] Application Programming Interface

[3] e.g. web browser, calendar, text messaging, ...

The Dalvik Virtual Machine is optimised for low-memory requirements. It has a just-in-time compiler and register-based architecture. In contrast, the Java Virtual Machine is stack-based.

The Dalvik Virtual Machine uses its own class library and does not claim to be compatible to standard Java Virtual Machines, but even calling and being called by native applications is implemented compatible to the Java Native Interface. [17]

To isolate applications for security, Android applications are not run in the same instance of a Dalvik Virtual Machine. Instead, for every application a new instance of a virtual machine is executed. *Zygote* [17] is a system service which was created to do this efficiently.

Zygote starts a Dalvik Virtual Machine at system startup, initialises it and preloads commonly used libraries. After this, Zygote listens for requests to start a new Android application. If an application start is requested, the preloaded virtual machine will be cloned. It is then ready to be used by the new application.

2.1.2. Linux kernel extensions

Android extends the Linux kernel by several features to adapt it to mobile devices. The Android specific IPC[4] mechanism *Binder*, a new shared memory allocator called *ashmem* and the so called *lowmemorykiller* are extensions dealt with in the following.

Binder

Almost whenever two applications on the Android platform interact, it happens through the Binder IPC mechanism. Binder is a customised rewrite of OpenBinder[5] for Android. Originally, OpenBinder was developed by Be Inc. under the lead of Dianne Hackborn, who is now involved in the development of Android. [7] It is object oriented and provides file exchange as well as remote procedure calls between processes. Additionally, Linux file descriptors and object references can be exchanged, allowing an efficient way of data exchange through shared memory.

Methods on a remote object can be called as if they were local methods by calling remote methods synchronously. This means the sender will be blocked until it receives the result from the called method on the remote object. For this, the Binder framework is able to start and stop threads. [17] However, the receiver will not be blocked by a remote call in contrast to the sender. This is achieved through a thread pool where threads are kept ready to handle requests.

To accomplish all these tasks, Binder is implemented as a kernel driver. This also avoids copy

[4]Interprocess Communication

[5]http://www.angryredplanet.com/ hackbod/openbinder/docs/html/

operations between the kernel and userspace.

To facilitate applications to publish and find functionality of other applications, there is a special service which all applications can contact. This service is called *Service Manager* [4] (also called Context Manager) and is itself contacted through its (publicly known) Binder interface. If an application wants to publish functionality, it has to register itself at the Service Manager with an unique identifier. An application requesting functionality of another application uses this unique identifier to query the Service Manager.

Types of arguments to remotely callable methods always have to implement a special interface called `Parcelable`. This enables actual arguments to be marshalled at the sender for transmission and unmarshalled at the receiver.

Binder uses a client-server model for communication. An example interaction between two applications 'App A' and 'App B' could proceed like in the following description (see also Figure 2.1).

1. App A wants to call `method(object)` of App B. For this App B has published this functionality to the Service Manager. App A contacts the Service Manager to request functionality of App B.
2. The Service Manager responds by sending the proxy (also known as client stub in other RPC[6] mechanisms) for making calls on App B to App A.
3. App A calls `method(object)` of App B on the previously acquired proxy. The actual parameter `object` of the call is marshalled using the `Parcelable` interface implemented by `object` and forwarded with the call to the Binder kernel driver. App A is blocked.
4. The Binder kernel driver relays the call to a thread in App B's thread pool.
5. App B unmarshals the actual parameter and processes the call. The result is then marshalled and send to the Binder kernel driver.
6. The Binder kernel driver forwards the result to App A. App A is unblocked and unmarshals the result.

Sadly, documentation on Binder is scarce, so most information on Binder has to be derived from the barely documented source code or third-party documents.

[6]Remote Procedure Call

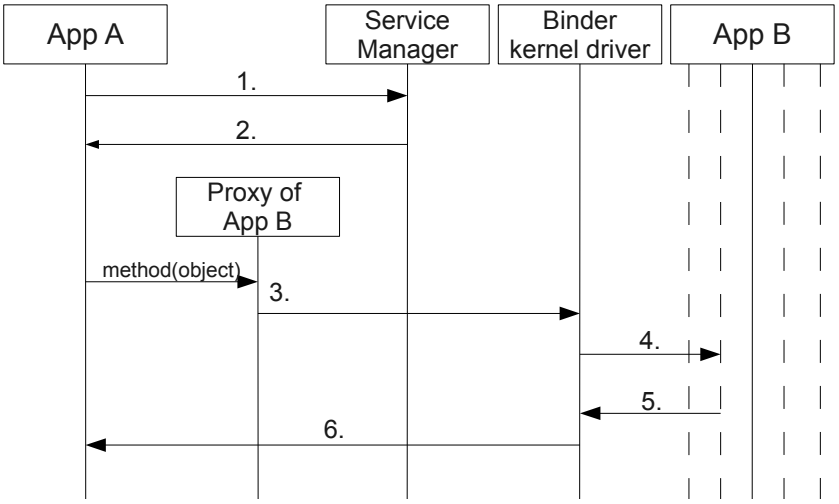

Figure 2.1.: App A calls a remote method of App B

ashmem

ashmem [19] stands for 'anonymous shared memory'. Android does not support POSIX[7] interfaces to allocate shared memory intentionally, but uses ashmem instead. ashmem is a file-based shared memory interface to userspace applications. It is optimised for memory constrained environments.

ashmem gives the Linux kernel the ability to evict memory under memory pressure. Memory pages are 'pinned' by default, which means that the kernel must not evict them unless they are 'unpinned' explicitly.

Where documentation on Binder is scarce, documentation on ashmem is virtually non-existent.

lowmemorykiller

Current versions of the Linux kernel have a feature called *OOM*[8] *killer*. The kernels of major Linux distributions are configured such that processes can request more memory than the system has currently available. This makes memory use very efficient, because processes do normally not use their allocated memory directly nor do they use all memory allocated throughout their lifetime. The downside is that this can lead to conditions where memory is so low that core kernel tasks cannot be executed any more.

The OOM killer actively avoids those situations. It gets active when the system runs out of memory and selects a process to kill using a set of heuristics. [15]

[7] Portable Operating System Interface - Standardised API defining the interfaces between operating system and applications

[8] Out of memory

To solely rely on heuristics can lead to processes being killed which are important to the user or even the system. For this purpose the behaviour of the OOM killer can be adjusted by changing the value in the file `/proc/<pid>/oom_adj`, where `<pid>` is replaced by the id of the process for which the OOM killer's behaviour should be adjusted. The higher the value in this file, the higher is the probability of the corresponding process being killed. A process can be immunised against the OOM killer by setting the value in `oom_adj` to the constant `OOM_DISABLE` (defined in `linux/oom.h`, currently set to -17).

The documentation of version 3.5 of the Linux kernel (released in July 2012) states that in the near future `/proc/<pid>/oom_adj` will be replaced by `/proc/<pid>/oom_score_adj`, allowing much finer grained control of the OOM killer's behaviour (see: `Documentation/feature-removal-schedule.txt` in the Linux kernel's sources).

During the development of Android the lowmemorykiller [15] kernel driver was created. It extends the OOM killer by the possibility to set six OOM levels instead of just one (memory critically low). This way, the lowmemorykiller can get active and notify processes earlier and group them into six classes corresponding to the OOM levels. The six OOM levels and assigned process classes are defined and described in the `ProcessList` class of the Android source code.

When the free memory falls below the first level, unimportant background processes are notified. They do not exit directly, but save their state. When free memory decreases further under the next level, they had enough time to clean up their cache and can get exited cleanly. Simultaneously, more important background processes are notified to save their state. Once the free memory falls under the next four levels one after the other, it goes on as before; a more important class of processes gets notified and the class of processes notified at the previous level is exited. The last and most important class of processes includes the currently visible application.

Android applications running in Zygote have a so called *lifecycle*. This means an application can be in different states, depending on the application's visibility. The Android framework's `ActivityManagerService` assigns Zygote processes to process classes dynamically according to the corresponding application's lifecycle. When a process enters a new process class, its `oom_adj` value gets adjusted correspondingly. The user's interactions with the application is thus tied to the framework's decision which process to kill.

2.1.3. Android Init Language

"In Unix-based computer operating systems, init (short for initialisation) is the first process started during booting, and is typically assigned PID number 1. It is started by the kernel using a hard-coded filename [...]. Init continues running as a daemon until the system is shut down,

and is the direct or indirect ancestor of all other processes." [23] [6]

Android uses its own init program. At startup it parses a file called `init.rc` written in the *Android Init Language*. The Android Init Language consists of four classes of statements: *Actions*, *Commands*, *Services*, and *Options*.

Actions are a named sequence of Commands with a trigger. The trigger defines under which Event the Action should be executed. Events can e.g. be the very early initialisation of init itself or the mounting of the file system. A set of Commands allows to alter the partition layout, create and write into files, change the network status and more. Services are any programs started by init directly. Options are attributes to Services changing how and when a Service is run.

Developers can extend `init.rc` by importing their own *.rc files into it. This allows customised partition setups or custom Services being executed when a defined Event is triggered e.g. system startup or a plugged in charger.

For more information on Android's init process and its configuration see `system/core/init/readme.txt` in the sources of the Android Open Source Project in version 4.0.4.

2.2. Real-time

"Real-time computing may be defined as that type of computing in which the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced." [18]

A *real-time system* is a computer system where at least some of the tasks are *real-time tasks*. Real-time tasks are subject to *real-time constraints*, meaning they must guarantee response within a limited amount of time. The point in time until which the response has to be given is called *deadline*.

Real-time tasks may be further classified in *hard real-time* and *soft real-time* tasks. Hard real-time tasks must meet their deadline, otherwise it is a system failure. E.g. it is unacceptable for an aircraft computer to miss the deadline to lower the landing gear. Soft real-time tasks are less critical. It makes sense to schedule them even if they miss their deadline, but the quality of the task will suffer. An example would be the decoding of a video stream.

The industrial embedded domain couples computer systems and the real world tightly and puts real-time demands on computer systems as such. Thus, real-time capabilities are the exclusive

concern of Linux in this domain. To provide the userfriendliness of Android to the industrial embedded domain, it has to become real-time capable.

2.2.1. Androit

*Androi**t*** [13] is the real-time Android implementation proposed by Mauerer et al.. They followed the approach to separate Android Java applications running on the Dalvik Virtual Machine and real-time applications running natively (see Figure 2.2). This way, the Dalvik Virtual Machine need not to be replaced or modified to become real-time capable, which would be a fundamental change to the Android framework. The Linux kernel is extended by the 'preempt_rt' patchset to make it support real-time tasks. This patchset was chosen above the numerous other available patchsets, because it is predicted to become integrated into the mainline Linux kernel in the near future. [13] To make the Android Java and real-time domain work together, data exchange is essential. *AndroitShmem* is an appliance which was created to make data exchange between these two domains work adequately. Its ideas are described in more detail in the following section.

With this approach Android applications are still easily programmable and usable. They function as a user interface to real-time applications running natively as real-time Linux tasks.

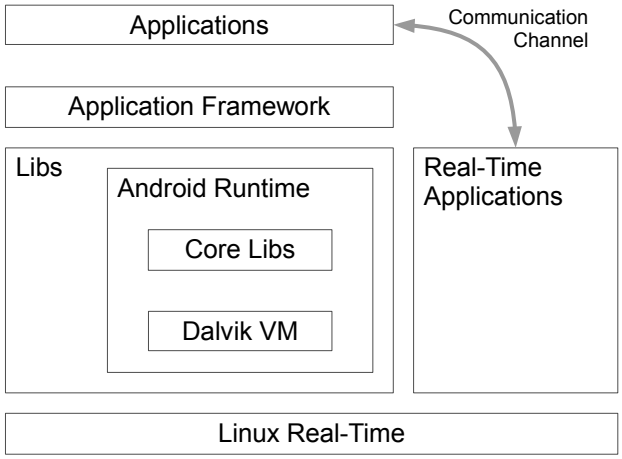

Figure 2.2.: Architecture diagram of Androit, taken from [13]

2.3. AndroitShmem

AndroitShmem is an appliance proposed by Mauerer et al. [13] to facilitate data exchange between Android applications and real-time tasks on an Androit system. AndroitShmem consists of a server, which allocates memory for the shared data structure and runs as a service to make it

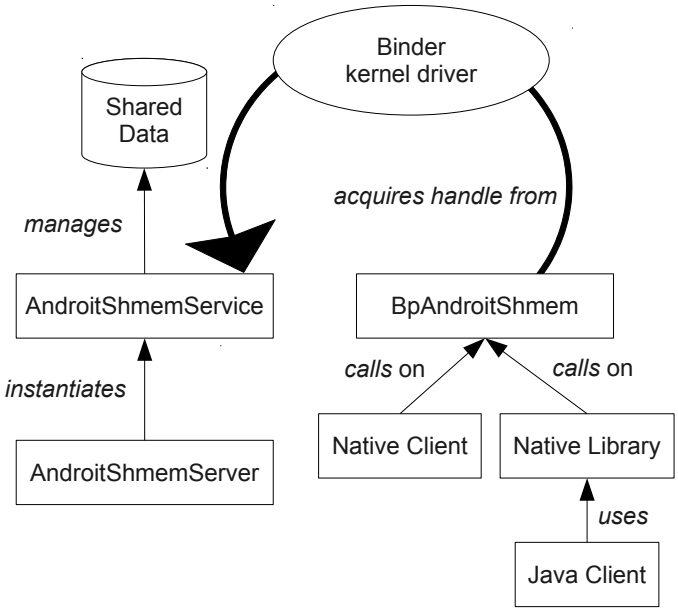

Figure 2.3.: Shematic overview of AndroitShmem

accessible through Binder. Every process (real-time or non-real-time) which acquires a handle to the shared data from the server is an AndroitShmem client and uses the server's Binder interface. Native clients get a pointer to the shared data and access it directly, whereas Java clients use wrapper types provided by Java Native Interface. A shared library provided by AndroitShmem makes the shared data easily accessible to Android Java applications. Figure 2.3 schematically shows how clients acquire a handle to the shared data from the AndroitShmem server.

Interfaces

For the server and clients to be able to communicate through Binder, an interface was created. This interface is called `IAndroitShmem`. It is implemented as a C++ class. The class extends the `IInterface` interface, which is the base class for all Binder interfaces. `IAndroitShmem` declares the virtual function `getShmem()`. It will later be used for clients, as well as for the server internally to obtain a handle to the shared data structure.

`IAndroitShmem` is extended by two classes. The first one is the server local Binder interface called `BnAndroitShmem`. `BnAndroitShmem` defines the function `onTransact()`. `onTransact()` will be called by the Binder kernel driver whenever a call on the server's Binder interface has to be processed. It will then identify the type of the call and respond to it. In this case the only possible call type is `GET_SHMEM` and the appropriate response is the result of `getShmem()`.

The second class extending `IAndroitShmem` is `BpAndroitShmem`, the remote Binder interface or 'proxy' to the server. It defines `getShmem()`, declared in `IAndroitShmem`, for clients. This

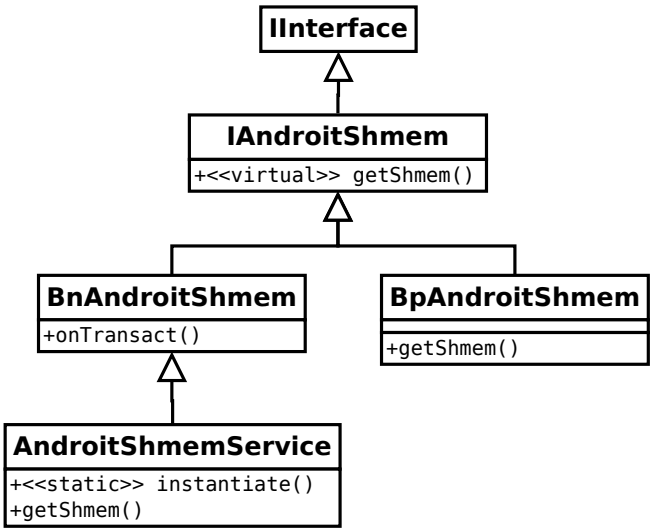

Figure 2.4.: Simplified class diagramm of AndroitShmem interfaces

implementation of `getShmem()` queries the server for a handle to the shared data through Binder (call type `GET_SHMEM`) and returns it.

Figure 2.4 shows the hierarchy of Binder interfaces. Details which are unimportant for the purposes of this thesis were omitted. The class `AndroitShmemService` will be explained in the following.

Server

The server is implemented in a class called `AndroitShmemServer`. Its purpose is to instantiate the shared memory service implemented in the class `AndroitShmemService`, start the thread pool for processing Binder calls and log the status of the shared memory periodically.

`AndroidShmemService` extends `BnAndroitShmem`, the local Binder interface. On its instantiation it opens a file in secondary storage and maps it into memory with the help of a class called `MemoryHeapBase` provided by Android. `MemoryHeapBase` allows easy exchange of pointers to the shared data between processes through Binder. The service's `getShmem()` implementation will just return a handle to the previously allocated memory.

`AndroitShmemService` registers itself as a service at Android's Service Manager to publish its functionality i.e. clients will then be able to obtain a proxy to the AndroitShmem service and call `getShmem()` on it.

Clients

Clients can be implemented natively or as Android Java applications. Native clients can query Android's Service Manager directly to acquire the proxy to `AndroitShmemService` as implemented in `BpAndroitShmem`. The proxy then povides `getShmem()` for the client to obtain a handle to the shared data. The shared data structure can then be accessed directly.

Java clients acquire a handle to the shared data with the help of a shared library implementing JNI functions for the Java class `SharedMem`. `SharedMem` can then access the shared data through Java wrapper classes provided by the Java New I/O library. Java programmers do not have to get involved with Android's Service Manager or Binder, they can obtain a handle to the shared data very easily with the call of one function while communication with the server happens in the background.

Clients access the shared data concurrently. It is crucial to synchronise access to the shared data in a way that no real-time task has to wait for its access until a non-real-time task has finished its work. Furthermore, the shared data has to be allocated in a way it will not be withdrawn from the system under any circumstances, as this would be fatal in a real-time environment. These will be the main topics of the next chapters.

2.4. Conclusion

This chapter introduced fundamental notions in the Android and real-time context. Androit, an effort to combine both worlds, was presented. Additionally, the current state of AndroitShmem was described.

The following chapter will cover the first improvement made to AndroitShmem in the course of this thesis. It will improve its performance by using main memory instead of secondary storage for the shared data.

3. Using main memory as shared storage

This chapter will base on the notions and concepts presented in the previous chapter. It will comprehensively introduce an improvement made to AndroitShmem. The goal of this chapter is to use the main memory as shared storage for the AndroitShmem server instead of secondary storage, as it was previously the case. It will discuss two possible approaches. After this, the implementation and performance measurements made in the course of this thesis will be presented.

3.1. Fundamental notions

This section will introduce the concept of *memory-mapping*, as well as a file system which temporarily stores data in main memory. The conjunction of both will be one approach to allocate shared memory for AndroitShmem. It will be compared to the approach of using ashmem in Section 3.2.

3.1.1. Memory-mapped file

One solution to share data between processes is using a file. Processes then have to open the same file to read from and write to it to share information. There is one inconvenience: whenever a process wants to update the shared data, it has to seek in the file. It would be much more convenient to be able to use pointer arithmetic to update the shared data. [8]

A *memory-mapped file* [11] in Unix systems is a segment of virtual memory which has been assigned to a file that is physically available in the file system. In other words the file is mapped into the different address spaces of the data sharing processes. Thus, the file can be edited with pointer arithmetic as if it was primary memory. This also leads to an increase in performance, as no system call needs to be executed to alter parts of the file. A simple switch to the process'

local memory is sufficient instead of an expensive context switch.

3.1.2. tmpfs

tmpfs stands for 'temporary file system' (see: `Documentation/filesystems/tmpfs.txt` in the Linux kernel sources in version 3.5). It is a file system which keeps all files in virtual memory. All files stored in a tmpfs mount are temporary, no files on hard disk will be created. This means that on unmount (e.g. at reboot) everything stored in it will be lost. tmpfs stores everything in the kernel's internal caches and has the ability to swap out unneeded pages.

As long as files stored in a tmpfs mount are held in the kernel's caches and not swapped out, access to them should be considerably faster than access to files in persistent file system mounts, as tmpfs stores them in primary instead of secondary storage.

3.2. Concept

This section discusses two different approaches of enabling the AndroitShmem server to use primary storage for sharing data and presents a concept using the approach which exposes to be superior for the purposes of this thesis.

3.2.1. tmpfs versus ashmem

As described above, Android does not support POSIX interfaces for allocating shared memory, but uses ashmem (see: Section 2.1.2). This results in two feasible ways to implement shared memory using main memory for this thesis' purposes:

1. Set up a tmpfs mount for AndroitShmem, create a file for memory-mapping and let the AndroitShmem server map this file to provide shared memory. Make sure the shared memory does not get swapped out of main memory.

2. Allocate memory through ashmem, make sure allocated memory is not revoked under any circumstances and let the AndroitShmem server manage this allocated memory as shared memory. Additionally make sure the shared memory stays in main memory.

The first approach requires more effort to allocate the shared memory, because the tmpfs has to be established and the file created. This has to be done before the AndroitShmem server is started. At best, AndroitShmem's requirements are directly met after system start up. This means that the init process at boot up has to be altered. Since it is favourable to start the

AndroitShmem server at system start up and set its `oom_adj` value to `OOM_DISABLE`, the init process will be altered anyhow. Furthermore, Android allows inclusion of custom init scripts into the init process in an easy way.

The second approach does not require any additional efforts to allocate the shared memory. Source code analysis showed that memory allocated through ashmem is pinned by default. This means that the kernel may not revoke this memory. As no documentation on this behaviour is available, a deeper source code analysis would be mandatory to make sure this is the case. Furthermore, the lack of an openly available documentation implies that there are no statements about the stability of ashmem. It is not clear, if it is completely stable or will undergo changes in the future which might affect its suitability for AndroitShmem.

As it is critical to provide a stable environment for real-time applications, the first approach is the superior one. It relies on well known, documented and stable techniques. Additionally, ashmem itself uses a tmpfs instance in the background, therefore also performance-wise the first approach should be in no way inferior.

3.2.2. Using tmpfs

The approach for allocating the shared memory in main memory will be to use tmpfs and a memory-mapped file. To establish a tmpfs mount, the Android init process will be altered. This will be done by including a custom script into Android's standard init script in the file `init.rc`. The custom script will create a directory and mount a tmpfs instance in it. It will then create the file to be memory-mapped later and set its permissions in a way that no normal user can access it. Additionally, a shell script will be run which starts the AndroitShmem server and makes sure it will not be killed by the kernel under any circumstances. The AndroitShmem server will map the previously created file into memory and use this memory for sharing.

3.3. Implementation

As tmpfs exposed to be the superior approach for allocating shared memory for the purposes of this thesis, this section will follow this approach. It will present the implementation written in the course of this thesis, which is based on the concept developed in the previous section.

3.3.1. Init process adjustments

The Android framework makes it easy to adjust the init process to your needs by custom init scripts. Listing 3.1 shows how external scripts can be included into Android's `init.rc`.

```
1 import /init.${ro.hardware}.rc
2 import /init.androitshmem.rc
3
4 ...
```

Listing 3.1: Abstract from init.rc

Line 1 includes hardware-specific init scripts, line 2 was added to include the custom init script-file `init.androitshmem.rc`.

Next the contents of `init.androitshmem.rc` have to be defined. It is shown in Listing 3.2. Lines 1 to 6 define an Action with the trigger `init`. `init` is a trigger that occurs directly after init set up the most important properties for the init process itself. It is used to set system-global variables and set up important filesystem mount-points. Additionally to the trigger the Action consists of four Commands. The first Command in line 3 creates the directory `/mnt/shm`. It is readable and writeable for and belongs to the user and group `root`. Other users or groups do not have access. Line 4 mounts a tmpfs instance to the previously created directory with the same access rights. Note that this implies that nothing can be executed from this mount. Line 5 writes a zero to the file `/mnt/shm/map` and thus creates this file. This file will be used as the storage for the shared data by the AndroitShmem server. Line 6 sets access rights such that only the user root can read from and write to the recently created file.

Lines 8 to 10 define a second Action with the trigger `boot`. This trigger occurs after the filesystem has been set up and is used to do basic network initialisation, set permissions to system core files and start system Services. Beside the trigger the Action consists of the Command '`start androitshm`', which causes init to start the Service `androitshm`.

The Service `androitshm` itself is defined in lines 13 to 16. Line 13 declares the Service name `androitshm` for the shell script located in `/system/bin/init.androitshmem.sh`. The Options in lines 14 to 15 tell init to start this Service as user and group `root`. The Option `oneshot` in line 16 tells init not to restart the Service when it exits. Lines starting with a hash are comments and will be ignored by the parser.

```
1  on init
2      # create AndroitShmem mount point
3      mkdir /mnt/shm 0600 root root
4      mount tmpfs tmpfs /mnt/shm mode=0600,gid=0
5      write /mnt/shm/map 0
6      chmod 0600 /mnt/shm/map
7
8  on boot
9     # start the service 'androitshm' defined below (AndroitShmemServer)
10    start androitshm
11
12 # declare AndroitShmemServer
13 service androitshm /system/bin/init.androitshmem.sh
14    user root
15    group root
16    oneshot
```

Listing 3.2: init.androitshmem.rc

The shell script `init.androitshmem.sh` is defined in Listing 3.3. Line 1 defines this file as a shell script, using Android's default shell located at `/system/bin/sh`. Line 3 starts the AndroitShmem server located in `/system/bin`. The '&' starts the server in the background such that further commands can be issued concurrently. Line 4 writes the constant number '-17' to the file `/proc/$!/oom_adj`. During execution '$!' will be substituted to the the process id of last process launched in background, in this case this is the AndroitShmem server. The constant number '-17' stands for the constant `OOM_DISABLE` and tells the lowmemorykiller not to kill the AndroitShmem server under any circumstances (see section 2.1.2).

```
1 #!/system/bin/sh
2
3 /system/bin/AndroitShmemServer &
4 echo -17 > /proc/$!/oom_adj
```

Listing 3.3: init.androitshmem.sh

After these adjustments the tmpfs mount with the file to be memory-mapped and the AndroitShmem server get initialised at system startup. Furthermore, the lowmemorykiller is told not to kill the AndroitShmem server. But so far the file in the tmpfs mount is not used by AndroitShmem.

3.3.2. AndroitShmem adjustments

Since the AndroitShmem server is the central manager of the shared memory, it is the only component of AndroitShmem that has to be adjusted to work with the tmpfs mount. Furthermore, the server already used the Android framework's class `MemoryHeapBase` to map a file into memory and make it shareable through Binder. Thus, only minor adjustments have to be made.

```
66 ...
67     shmemHeap = new MemoryHeapBase("/mnt/shm/map",
           sizeof(struct androit::shared));
68 ...
```

Listing 3.4: Excerpt from AndroitShmemServer.cc

Listing 3.4 shows the only line that has to be edited. On instantiation of the `MemoryHeapBase` object, the constructor gets passed the new location of the file to be mapped into memory i.e. `/mnt/shm/map` as previously defined in `init.androitshmem.rc`.

One line of code is additionally important to mention, it is shown in Listing 3.5. Line 73 locks the shared memory into the AndroitShmem server's main memory using the function `mlock()`. This means that the corresponding memory pages will not get swapped out to secondary storage. Otherwise it could occur that real-time processes would have to wait for the memory page to get loaded into main memory again, which would be an unacceptably costly operation.

```
74 ...
75         res = mlock(shmemHeap->getBase(), shmemHeap->getSize());
76 ...
```

Listing 3.5: Excerpt from AndroitShmemServer.cc

This section finishes the implementation of a mechanism which enables the AndroitShmem server to use main instead of secondary memory for shared storage. The next section evaluate these efforts by measuring performance and comparing the results.

3.4. Performance measurements

Performance was measured in an emulator running Android 4.0.4 on a conventional desktop computer. The measure was nanoseconds taken to alter the shared data. As shown in Listing 3.6, the shared data consisted of a structure composed of an integer, a floating point and an array

```
1 struct data_struct {
2     int     integer;
3     float   fp;
4     long    arbitrary[1024];
5 };
```

Listing 3.6: Shared data used for performance measurements

of long integer with 1024 entries. To prevent the the system from caching the variables' values, every component of the struct was declared volatile. The struct was used to perform some sample shared data accesses by an AndroitShmem client, which had to acquire a pointer to the shared data from the AndroitShmem server to access it. In every access the client altered the values of the integer, the float and every entry of the array once. Measurements were taken for the initial implementation with a memory-mapped file in secondary storage, a memory-mapped file in primary storage using tmpfs as described above and shared memory acquired through ashmem. Furthermore, three different types of measurements were taken: the first access of a client to the shared data after Android was just started up completely, an access of a client after the shared data was accessed previously and ten consecutive accesses by a client after the shared data was also already accessed before, acquiring the pointer to the shared data just once. Every measurement was taken ten times.

To get a value in nanoseconds representing how long a run took, two times were taken by the client and then subtracted. The first time was taken after the pointer to the shared data structure had been acquired, directly before the shared data was accessed. The second time was taken directly afterwards. The time was taken from a high resolution timer provided by the CPU supporting a resolution of one nanosecond. The results were revised such that the time it took to get the time value itself is not included.

3.4.1. First access

Table 3.1 shows the measurements' results for the first access to the shared data structure after Android booted up completely. Using primary storage took just under 60% of the time taken when using secondary storage, which is quite an improvement. The results also show that there is no considerable performance difference between tmpfs and ashmem.

	secondary storage	primary storage (tmpfs)	primary storage (ashmem)
average	313335 ns	182915 ns	187017 ns
minimum	274979 ns	176633 ns	169209 ns
maximum	341325 ns	191852 ns	202066 ns

Table 3.1.: Time taken by a client for the first access to manipulate the shared data

3.4.2. Subsequent access

The measurements' results for a subsequent access can be found in Table 3.2. Again with primary storage taking just over 40% of the time taken when using secondary storage, using primary storage is a considerable improvement. The time taken for subsequent (i.e. not first) accesses is notably shorter than the time taken for first accesses, because the resources needed for the manipulation are already cached. The shared data's variables are marked volatile however, as mentioned before. The differences between the time taken when using tmpfs in comparison to ashmem are again only some microseconds and not considerable.

	secondary storage	primary storage (tmpfs)	primary storage (ashmem)
average	101417 ns	44400 ns	41651 ns
minimum	86745 ns	40747 ns	39493 ns
maximum	112733 ns	47948 ns	43852 ns

Table 3.2.: Time taken by a client to manipulate the shared data once, after the shared data was accessed before

3.4.3. Consecutive accesses (subsequent)

Table 3.3 contains the measurements' results when manipulating the shared data structure ten times in a loop after the shared data has been accessed before. With taking just over 70% of the time of secondary storage, using primary storage again obtains an improvement. The improvement is not as big as one might expect from previous results, because despite declaring the variables as volatile, the Linux kernel caches the memory-mapped file internally. This behaviour is not deactivatable without far reaching interventions. Again, differences between using tmpfs over ashmem are negligible.

	secondary storage	primary storage (tmpfs)	primary storage (ashmem)
average	178259 ns	132790 ns	131795 ns
minimum	164436 ns	120510 ns	127076 ns
maximum	198441 ns	138757 ns	138705 ns

Table 3.3.: Time taken by a client to manipulate the shared data ten times, after the shared data was accessed before

3.5. Conclusion

This chapter discussed how to provide shared data stored in primary storage. A concept was developed and implemented. Furthermore, it was shown how the init process of Android can be adapted to include custom scripts. The measurements' results were presented to evaluate the implementation.

Overall it was shown that the concept of using tmpfs in combination with memory-mapping leads to a stable and performant shared memory solution. Access times to the shared data could be improved considerably. Additionally, it was shown that ashmem cannot provide a better performance than tmpfs for the purposes of this thesis. In sum, the approach of using the stable and well-tested tmpfs proved to be superior to the approach of using the virtually undocumented and therefore obscure ashmem.

The following chapter will deal with synchronisation. The most typical synchronisation techniques will be discussed. A concept of how to synchronise realtime and non-realtime clients accessing AndroitShmem will be presented, implemented and analysed.

4. Synchronisation

AndroitShmem clients read from and write to the shared data concurrently, thus synchronisation is essential to maintain data integrity. After moving the shared data from secondary to primary storage in the last chapter, objects used for synchronisation can efficiently be stored in and accessed from it. This chapter will discuss basic synchronisation mechanisms and use this knowledge to take a deeper look at the synchronisation mechanism proposed in [13] for AndroitShmem. The implementation developed in the course of this thesis is based on the proposed synchronisation mechanism and will be explained in detail. Additionally, the implementation will be analysed and verified by using a model which was created for this thesis.

4.1. Fundamental notions

This section will discuss basic synchronisation mechanisms as an introduction to the more complex synchronisation mechanism designed for AndroitShmem, which will be presented in Section 4.2.

4.1.1. Synchronisation mechanisms

In the following, commonly used synchronisation mechanisms will be explained in general. *Atomic operations* are a required basis for the more powerful *locks*. Locks enforce limits on access by multiple processes to a resource. Processes cooperate with each other by sharing a lock. The resource may only be accessed if the lock could be acquired. *Spinlocks*, *semaphores*, *mutexes* and *reader/writer locks* are types of locks which will be explained below.

Atomic operations

The simplest form of synchronisation are atomic operations. Atomic operations are guaranteed to be executed without any interruption, as if they consisted of a single machine instruction. Hardware support is explicitly needed to disable interrupts and prevent other processors in

```
1 function lock(int *lock) {
2     while(test-and-set(lock, 1) == 1)
3         ;
4 }
```

Listing 4.1: Example spinlock locking using `test-and-set()`

a multicore processor from execution. All processors supported by the Linux kernel support atomic operations. [12]

An example atomic operation is `test-and-set()`, which writes a new value to a variable and returns the old one. Operations like this are needed to implement more high-level synchronisation mechanisms like spinlocks.

Spinlocks

Spinlocks [12] are locks which cause a process to wait in a loop and check repeatedly until the lock becomes available. They should only be used for short-term waits, because the process stays active and does not do anything useful. This is also known as *busy waiting*. They do not produce much overhead, as no context switch to another process is executed.

A simple spinlock locking procedure is outlined in Listing 4.1, it uses an atomic `test-and-set()` function where the first parameter is the memory position to write to and the second is the value to be written.

Semaphores

A semaphore [18] [12] is a specially protected integer variable with a process queue on which only three operations may be performed: `wait()`, `signal()` and `initialise()`. They are used to control entry to and exit from critical sections. More specifically, they track how many units of a common resource are available for access by multiple processes. On initialisation, the semaphore's value is set to a non-negative integer value. When a process wants to enter the critical region protected by the semaphore, it executes `wait()` on the semaphore. `wait()` decrements the value of the semaphore by one. If the value is negative afterwards, the process executing `wait()` is blocked and enqueued into the semaphore's process queue, which means it "waits" for the semaphore to become available. After a process leaves the critical region, it executes `signal()` on the semaphore, incrementing the semaphores value by one. If the value was negative before i.e. at least one process is waiting in the semaphore's process queue, a

process is dequeued and then unblocked.

It is important to note that `wait()` and `signal()` must not be interrupted during execution and thus require atomic operations to be implemented.

Mutexes

Mutexes [18] [20] are a special case of semaphores. The semaphore is initialised with a value of one, additionally only the process which decreased the value can increase it to one again. Thus, maximal one process can be in the protected region at any time.

Reader/Writer locks

It is often not a problem to let multiple processes read from a data structure at a time, as writing to the data structure can only be done by a single process while no other process accesses it. Mutexes are too restrictive in this case.

This is where reader/writer locks [12] are used. They differentiate between read and write access to data structures and allow any number of processes to perform read operations concurrently. Write access is restricted to one process only. Furthermore, read access is not permitted while write access is in progress.

4.1.2. Negative aspects of locks

Locks may be powerful tools to control concurrency, but they introduce several negative aspects:

- They cause *blocking*, meaning processes trying to access a currently taken lock have to wait until the lock is released.
- *Deadlocks* can be introduced. At least two processes are involved in this. Both wait for the lock the other one has to finish, while not releasing the lock held before getting the other.
- *Livelocks* can occur. They are similar to deadlocks, but their states constantly change in respect to another, none progressing.
- Lock handling adds overhead to every access to the protected resource, even when chances to collisions are very rare.
- Contention for a lock limits scalability and adds complexity.
- *Convoying* can be introduced, meaning if a process holding a lock is descheduled, all

processes competing for this lock have to wait.

- They can introduce *priority inversion*, meaning a low priority process can detain a higher priority process when the lower priority process holds the common lock. Its implications are further discussed below.

[20] [18]

Priority inversion

Priority inversion [18] is a phenomenon that can occur in any system using priority-based preemptive scheduling. In real-time systems it becomes a critical issue to deal with.

Any system using priority-based preemptive scheduling should always run the process with the highest priority. Priority inversion denotes a situation in which a higher priority process has to wait for a lower priority process. This is the case when a lower priority process has locked a resource the higher priority process needs for execution. On attempting to acquire the common lock, the higher priority process will be blocked until the lower priority process releases the lock. In the worst case there may be some medium priority processes preempting the lower priority process. In this case the higher priority process would even have to wait for the preempting medium priority processes to finish, leading to a even more serious condition referred to as *unbounded priority inversion* [18].

Several approaches exist that avoid unbounded priority inversion, but the problem of priority inversion itself remains when using a common lock between higher and lower priority processes.

These aspects show that locks have to be used cautiously. Furthermore, the problem of priority inversion shows that shared locks between real-time and non-real-time processes must be avoided whenever possible.

4.1.3. Non-blocking algorithms

To avoid the negative aspects of locks, *non-blocking algorithms* [10] were introduced. Non-blocking algorithms ensure that mutual exclusion does not cause processes competing for a shared resource have to wait indefinitely for their execution. Non-blocking algorithms may guarantee two qualities of progress: *lock-freedom* and *wait-freedom*.

Lock-freedom

Lock-freedom is the weaker of both guarantees. It allows individual processes to be perpetually denied necessary resources and thus not finishing their tasks, but guarantees system-wide progress. An algorithm is lock-free if it satisfies that when the program processes are run sufficiently long at least one of the processes makes progress (for some sensible definition of progress). [10]

Wait-freedom

Additionally to guaranteeing system-wide progress, wait-freedom guarantees no process to be perpetually denied necessary resources. An algorithm is wait-free if for every operation the number of steps the algorithm will take before the operation completes is bounded. For realtime processes this quality is critical. Naturally, all wait-free algorithms are lock-free. [10]

A non-blocking algorithm for controlling access by multiple processes to shared data is *transactional memory*.

Transactional memory

The main idea of transactional memory [10] is to control concurrent access to the shared data analogously to database transactions.

In transactional memory a transaction is a sequence of steps executed by a single process. Transactions must be *serialisable* [10], meaning that they appear to execute sequentially, in a one-at-a-time order. This defines transactions to be atomic, which is a coarse grained property because transactions may include calls to several objects. Transactions may not introduce deadlocks or livelocks.

Transactions do not block by being executed speculatively i.e. they make tentative changes to the involved objects. If no synchronisation conflicts emerged while doing so, the tentative changes are made permanent. In other words the changes are committed. If else synchronisation conflicts emerged, the transaction is aborted and the changes discarded. On abort the transaction can be retried.

An important class of atomic operations for implementing transactional memory in software and non-blocking algorithms in general is *read-modify-write* [10]. Read-modify-write operations simultaneously read a memory location and write a new value into it. The new value can either be a completely new value or a function of the old one. A very frequently used representative of this class is `compare-and-swap()`.

```
1 int compare_and_swap(int* reg, int old_value, int new_value) {
2   int old_reg_value = *reg;
3   if (old_reg_value == old_value)
4     *reg = new_value;
5   return old_reg_value;
6 }
```

Listing 4.2: CAS for int values implemented in C (not atomic)

Compare-and-swap

`compare-and-swap()` or short CAS is an atomic function which reads the contents of a memory location and compares it to a given value. On equality it writes a given new value into it. Additionally, it returns the contents of the memory location read at the beginning. Every modern processor architecture supports CAS or an equivalent instruction. In C its code could look like shown in Listing 4.2, but note that the C code does not provide the crucial atomicity. [10]

4.2. Concept

The synchronisation mechanism for AndroitSchmem has to be designed cautiously such that priority inversion is not introduced and the real-time part is wait-free under reasonable assumptions. To achieve this, real-time writers will use a global sequence counter to maintain integrity and signal that a real-time write is in progress to non-real-time readers. Non-real-time writers will use the concept of transactional memory to update the shared data, which was introduced in the previous section. [13] A mutex for the real-time and one for the non-real-time writers ensure that only one real-time or non-real-time writer each can write to the shared data. Table 4.1 gives an overview of the concept and shows how clients are synchronised with each other. The next sections will explain this concept in detail.

4.2.1. Real-time part

Writer

Before a real-time writer makes changes to the shared data, it acquires the real-time writers' mutex and increases the sequence counter by one (initially zero). This way it is made sure that no real-time write is in progress and the sequence counter is increased to an odd value. Processes

Synchronised by	real-time writer	real-time reader	non-real-time writer	non-real-time reader
real-time writer	(first) lock	sequence counter	transactional memory	sequence counter
real-time reader	sequence counter	not necessary	sequence counter	not necessary
non-real-time writer	transactional memory	sequence counter	(second) lock	sequence counter
non-real-time reader	sequence counter	not necessary	sequence counter	not necessary

Table 4.1.: Synchronisation concept overview

wanting to read from the shared data can infer from the odd value of the sequence counter that a (real-time) write is in progress. Then the real-time writer can safely make changes to the shared data.

After making changes, the real-time writer increases the sequence counter by one again and releases the real-time writers' mutex. This finishes the real-time write. The increase of the sequence counter set it to an even value, meaning that no real-time write is in progress. Furthermore, an increase of the sequence number denotes an update to the shared data.

Reader

Real-time readers will wait until the sequence counter has an even value and save this value temporarily. Then they read information from the shared data. After this, they compare the previously saved value from the sequence counter with the current one. If the values match, the data was consistently read and the readers can proceed. Otherwise, the shared data was updated while reading and the information as well as the sequence counter have to be reread. This is repeated until the information was consistently read.

4.2.2. Non-real-time part

Writer

As mentioned before, non-real-time writers will use transactional memory to make changes to the shared data. Therefore non-real-time writers will not make changes to the shared data

directly, but to a copy.

At first, a non-real-time writer acquires the non-real-time writers' mutex and waits for the sequence counter to have an even value. It then saves this value temporarily and begins working on a copy of the shared data. Afterwards, it compares the previously saved and current value of the sequence counter to check for consistency. If the copy is still consistent with the current data after the changes were made, the copy is exchanged with the current data and the sequence counter increased by two to signal a data update. This has to happen atomically. If the copy is not consistent any more, the copy has to be updated to the current values, the sequence counter reread and the changes redone. This is repeated until the shared data could consistently be updated.

To be able to exchange the copy of the shared data and the current one atomically, the copy has to reside in the shared memory at the AndroitShmem server. This way not the whole data structure, but only a pointer to the currently active one has to be updated atomically. This is possible with atomic functions like CAS (explained in Section 4.1.3).

Reader

A non-real-time reader behaves exactly like a real-time reader. Furthermore, real-time and non-real-time readers can naturally access the shared data concurrently without interference.

4.3. Implementation

This section will discuss the implementation of the concept presented in the previous section. The implementation was developed in the course of this thesis.

4.3.1. Preliminary considerations

This section will explain the choices made on how to implement the synchronisation mechanism for AndroitShmem.

Non-real-time writer

An important point is that on success the non-real-time writer has to compare the sequence counter, update it and exchange the pointer to the active data copy all at once in a single atomic operation. Otherwise, consistency cannot be guaranteed under every possible circumstance.

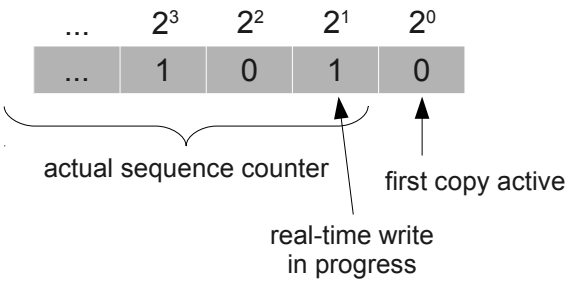

Figure 4.1.: Illustration of an exemplary sequence counter value

CAS is not sufficient to perform this operation, as it only supports one variable to compare and write to. A fitting operation for this would be DCAS (double compare-and-swap), which behaves like CAS extended to two variables. Unfortunately DCAS is not available on any widespread computer architecture nowadays. It can be implemented using CAS as presented in [9], but this approach would be costly for the purposes of this thesis. To circumvent this issue, the pointer to the active data structure as well as the sequence counter will be encoded into one integer variable. This way CAS can be used to perform the update atomically.

Previously, the first bit of the sequence counter was used to denote if a real-time write is in progress (value either odd or even). This signal will be shifted into the second bit to be able to use the first bit to denote which of the two copies of the data structure is the active one. Figure 4.1 illustrates how bits of a sequence counter value are reserved to store special information.

Mutexes

In Linux systems mutexes are generally supplied by the pthread[1] library, which is implemented in Android's own implementation of the standard C library namely *Bionic*. As stated in `bionic/libc/docs/OVERVIEW.TXT` in the Android 4.0.4 source code, Bionic's implementation of the pthread library does not support process-shared mutexes as needed by AndroitShmem to synchronise real-time and non-real-time writers respectively when accessing the shared data. When looking deeper into the actual implementation of pthread, this exposes not to be entirely true. In `bionic/libc/bionic/pthread.c` (l. 839 ff.) of the Android 4.0.4 source code the documentation states "our current implementation of pthread actually supports shared mutexes but won't cleanup if a process dies with the mutex held.". Furthermore, shared mutexes are used in core parts of the Android framework like the Surfaceflinger and Audioflinger, the window and audio compositor respectively. Therefore, the shared mutexes implemented in Bionic's pthread

[1] POSIX Thread

library can safely be used, but have to be rechecked in future versions of Android.

Atomic functions

The Android toolchain, used to build applications for Android as well as Android itself, includes the popular GCC[2] [5]. GCC supports built-in functions that can be used inside program code and are implemented by the compiler specifically for the target architecture. Among these are atomic functions as needed to implement the previously presented concept, making them easy to use.

4.3.2. Realisation

IAndroitShmem.h

To make the synchronisation mechanism easily usable to all clients accessing AndroitShmem and implement it in one place, it will be implemented in the header file of `IAndroitShmem`. This file is included in every AndroitShmem component.

In a first step, the shared C-structure has to be extended as shown in Listing 4.3. `data_struct` declares how the shared data looks like, the actual payload. In this case it just contains some examplary variables. `shared` is the shared structure that contains objects used for synchronisation, as well as the payload. The objects for synchronisation are encapsulated in an inner struct. Among these are the two mutexes for real-time and non-real-time writers respectively and the sequence counter. Furthermore, `shared` contains the shared data as declared by `data_struct` twice to support transactional memory. All of these parts were explained in detail in Section 4.2.

On start of the AndroitShmem server, the shared data structure has to be initialised to make synchronisation work correctly. The function `init_shared()` as shown in Listing 4.4 fulfils this purpose and will be called during the initialisation of the AndroitShmem server. Line 3 defines a mutex attribute, which is initialised and gets the process-shared state attribute set. Afterwards, it can be used to initialise the shared mutexes declared in the structure `shared` (lines 7-11). Lines 13 to 18 set sample values on the shared data. `init_shared()` returns the collected return values from the initialisation of the mutexes in form of an integer value. If everything went fine, the value returned is zero.

Real-time and non-real-time readers have to wait for the sequence counter to have its second bit unset (no real-time write in progress) and store it. They also have to compare a stored sequence counter value to the current one. These operations are also implemented in `IAndroitShmem.h`.

[2]GNU Compiler Collection

```
1  struct data_struct {
2          int    integer;
3          float  fp;
4          long   arbitrary[1024];
5  };
6
7  struct shared {
8      struct {
9          pthread_mutex_t rt_wlock;
10         pthread_mutex_t nonrt_wlock;
11         unsigned int sequence;
12     } protect;
13     struct data_struct data[2];
14 };
```

Listing 4.3: Shared C-structure as implemented in IAndroitShmem.h

```
1  static int init_shared(struct shared *shared) {
2      int result;
3      pthread_mutexattr_t attr;
4
5      shared->protect.sequence = 0;
6
7      pthread_mutexattr_init(&attr);
8      pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
9      result = pthread_mutex_init(&shared->protect.rt_wlock, &attr);
10     result += pthread_mutex_init(&shared->protect.nonrt_wlock, &attr);
11     pthread_mutexattr_destroy(&attr);
12
13     for (int i = 0; i < 2; i++) {
14         shared->data[i].integer = 42;
15         shared->data[i].fp = 23.42;
16         for (int j = 0; j < 1024; j++)
17             shared->data[i].arbitrary[j] = 1024 - j;
18     }
19
20     return result;
21 }
```

Listing 4.4: Function to initialise the shared data structure (IAndroitShmem.h)

Listing 4.5 shows these functions. `seq_begin()` reads the sequence counter value from the shared structure and checks if the second bit is set (lines 4-6). If it is set, there is currently a real-time write in progress, therefore the reader has to wait and reread the sequence counter (lines 7-8). `cpu_relax()` as used in line 7 is a procedure that has to be implemented architecture specific. It should give the CPU a hint that it currently executes a wait loop to be able to optimise accordingly. On x86 CPUs this is implemented as '`asm volatile("rep; nop" ::: "memory");`', which is the same instruction commonly used to wait in spinlocks. As soon as the second bit in the sequence counter's value is unset, the value is returned. `seq_doretry()` compares a given, previously stored sequence counter value to the current one. If the values match, it returns false. This would mean the data could consistently be read and no retry has to be done. If the values do not match, true is returned meaning a retry has to be done. Both functions are declared inline, to avoid the overhead of the function calls.

```
1  static inline unsigned seq_begin(const struct shared *shared) {
2      unsigned int sequence;
3
4      sequence = shared->protect.sequence;
5
6      while (sequence & 2) {
7          cpu_relax();
8          sequence = shared->protect.sequence;
9      }
10
11     return sequence;
12 }
13
14 static inline bool seq_doretry(const struct shared *shared, const unsigned
       int start) {
15     unsigned int sequence;
16     bool inconsistent = false;
17
18     sequence = shared->protect.sequence;
19
20     if (sequence != start)
21         inconsistent = true;
22
23     return inconsistent;
24 }
```

Listing 4.5: Sequence counter handling functions (IAndroitShmem.h)

Additional helper functions for beginning and ending real-time writes, as well as beginning and ending non-real-time writes will be explained when they are needed further below.

real-time write

The sample native client should represent a real-time client, it therefore has to be extended accordingly. Listing 4.6 shows how a real-time writer accesses the shared data synchronised and writes to it. It presents a shortened version of the function implemented in `AndroitShmemClient.cc`, excluding error handling and debug message output to have a clearer view on the essential parts.

Before writing to the shared data, the real-time writer has to call `begin_rt_write()` on the previously acquired handle to it. `begin_rt_write()` is a small function defined in `IAndroitShmem.h`, which acquires the real-time writers' lock to make sure only one real-time writer is writing and then increases the sequence counter by two. Afterwards, the second bit is set in the sequence counter, denoting to other clients that a real-time write is in progress. The real-time writer can then safely determine which copy of the shared data is active by looking at the first bit of the sequence counter (line 7). After this it can write to it, increasing the value of an integer in the shared data by one in this case (line 8). When the real-time client has finished writing, it has to call `end_rt_write()` defined in `IAndroitShmem.h`. It will again increase the sequence counter by two and release the real-time writers' lock. The second bit is unset afterwards, denoting no real-time write is in progress anymore. The increase of the sequence counter additionally indicates that the shared data was updated.

```
1  void doWrite() {
2      struct shared *container = getSharedData();
3      int active_data;
4
5      begin_rt_write(container);
6
7      active_data = container->protect.sequence & 1;
8      container->data[active_data].integer++;
9
10     end_rt_write(container);
11 }
```

Listing 4.6: Examplary write access to the shared data by a real-time client

The helper functions `begin_rt_write()` and `end_rt_write()` use the GCC builtin

__sync_add_and_fetch() to increase the sequence counter. Therefore, the counter is increased atomically and without compiler optimisations which could reorder reads from and writes to it. This ensures consistent states during these increases.

Non-real-time write

AndroitShmem's shared library for clients makes the shared data easily accessible to Android Java applications via Java Native Interface calls. It should be used to implement non-real-time applications with an easy to use user interface and has to be extended to support writing to the shared data.

Listing 4.7 shows a shortened version of the C++ implementation of the corresponding Java Native Interface function `updateByteBuffer()`, which the Java client can use to update the values of the shared integer and float variables. It excludes error handling to make the example of how to write to the shared data with a non-real-time client easier to read. Before any write accesses, the non-real-time writer has to call `begin_nonrt_write()` on the shared data. This function is defined in `IAndroitShmem.h` and acquires the non-real-time writers' lock to make sure only one is writing at a time. The counterpart to end the write process is `end_nonrt_write()`, which is also defined in `IAndroitShmem.h` and releases the lock. After making sure there is only one non-real-time writer active, it can determine which copy of the shared data is inactive by looking at the first bit of the sequence counter and inverting it (line 11). This copy is then used to write to with the goal of committing it after a successful write. To be able to check if the write was successful in the end, the value of the sequence counter before touching the shared data is stored in line 13. Afterwards, the new values for the integer and float variables can be written to the inactive copy of the shared data (lines 15 to 16). It is important to note that although it is not intended to write to the other variables, they have to be updated to the values currently stored in the active copy of the shared data (lines 18 to 19). Otherwise, it might be the case that the shared data is 'updated' to outdated values eventually. After the actual writing, the client tries to commit the now updated inactive copy to make it active (line 21). This uses a built-in version of the atomic `compare-and-swap()` function. It compares the current value of the sequence counter to the previously stored one. If the values do not match, it returns false and causes the client to retry the write operation, the current sequence counter stays untouched. In the case of a match, the sequence counter is increased by four and additionally gets its first bit inverted. The increase by four signals that an update to the shared data was made, it is the smallest increase the non-real-time client can make, as the bits representing one and two are reserved. The invert of the one first bit makes the previously inactive copy of the shared data the client wrote to the active one. All in all, this shows how the non-real-time client writes to

the shared data using transactional memory.

```
1  extern "C"
2  void Java_com_androit_SharedMem_updateByteBuffer(JNIEnv *, jobject, jfloat
       updateFloat, jint updateInt) {
3      struct shared *container;
4      unsigned int start_seq;
5      int update_data;
6
7      container = getSharedData();
8
9      begin_nonrt_write(container);
10
11     update_data = 1 - (container->protect.sequence & 1);
12     do {
13         start_seq = seq_begin(container);
14
15         container->data[update_data].fp = (float) updateFloat;
16         container->data[update_data].integer = (int) updateInt;
17
18         for (int i = 0; i < 1024; i++)
19             container->data[update_data].arbitrary[i] = container->data[1-
                   update_data].arbitrary[i];
20
21     } while (!__sync_bool_compare_and_swap(&container->protect.sequence,
           start_seq, (start_seq+4)^1));
22
23     end_nonrt_write(container);
24 }
```

Listing 4.7: Examplary write access to the shared data by a non-real-time client

Reading the shared data

Reading the shared data in a synchronised manner is essentially the same procedure for real-time and non-real-time clients. As the version for the non-real-time client becomes a bit longish because it includes code to handle the transition from native to Java code, a shortened version of the real-time read as implemented in `AndroitShmemClient.cc` will be presented here.

Listing 4.8 shows how a real-time client reads information from the shared data. It first saves

```
1  void doRead(void *arg) {
2      struct shared *container = getSharedData();
3      unsigned int start_seq;
4      int active_data;
5
6      do {
7          start_seq = seq_begin(container);
8          active_data = container->protect.sequence & 1;
9
10         LOGD("AndroitShmem read content: integer=%d, float=%f", container->
               data[active_data].integer, container->data[active_data].fp);
11
12     } while (seq_doretry(container, start_seq));
13 }
```

Listing 4.8: Examplary read access to the shared data by a real-time client

the current value of the sequence counter and then determines the active copy of the shared data by looking at the first bit of it (lines 7 to 8). This suffices as preparation to read from the shared data. In this case the contents of the integer and float variables are printed out as a debug message (line 10). After having read the data, the client has to check if this could consistently be done by using the function `seq_doretry()` from `IAndroitShmem.h` (line 12). If this is the case, the condition of the do-while statement is false and the reading process finished. If it is not, the the loop will be reexecuted and the data reread. The loop is repeated until the data was consistently read.

4.4. Analysis

After the implementation of the synchronisation mechanism was presented in the previous section, this section will describe how it was tested and analysed. The correctness of the implementation will be proven by verifying an equivalent model which was created in the course of this thesis.

4.4.1. Testing

To test the synchronisation mechanism, a native client and an Android Java application as shown in Figure 4.2 were implemented. Both are capable to write to and read from the shared

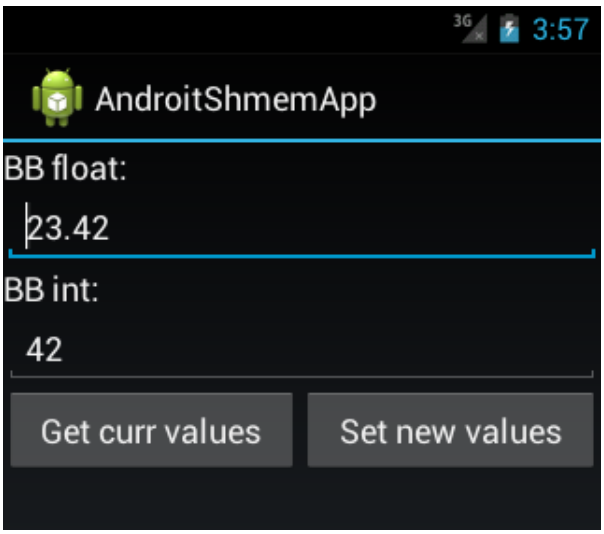

Figure 4.2.: Simple AndroitShmem Java client, implemented for testing purposes

data. The native client representing the real-time and the Java application the non-real-time part, using the corresponding synchronisation mechanisms. Both clients were invoked in several varieties, logged extensive debug messages and included sleep instructions to be able to cause many possible interleaves during execution of the synchronisation technique and make sure everything went fine. These clients were also helpful to show that this technique is practical and to make some general observations.

4.4.2. General observations

Locks were used sparingly, furthermore the real-time and non-real-time part do not share a lock. Therefore, priority inversion is not introduced. Nevertheless, the non-real-time client can delay the real-time client when it commits data to the shared memory while the real-time client is reading. In this case the real-time client has to reread the information. If non-real-time clients commit data very frequently such that the real-time client can never consistently read the data, it would be stuck in an infinite loop caused by the lower priority non-real-time client. While this is a threat in theory on multiprocessor systems, it is not relevant in reality. The non-real-time clients' purpose is to host an easy-to-use user interface to set variables for the real-time clients' executions and observe the system. Therefore, the write access of a non-real-time client is always instantiated by a human and can be assumed to be infrequent enough for the real-time read to take finitely bound time. The user interface could even enforce infrequent writes with a short delay after initiating one, while still remaining practical. In fact, as a real-time client never has to wait for a non-real-time client when writing data, in an AndroitShmem system with a single real-time process the real-time part would be wait-free. Having a single real-time threat is the

case AndroitShmem was intended for.

In the same way as explained above, a too frequently writing real-time process can cause non-real-time processes to infinitely loop while trying to read from or even write to the shared data. Assuming the real-time processes are not faulty, the system would still make progress and thus be lock-free. It is in the responsibility of the designer of the real-time process to handle this danger thoughtfully. It is advisable to avoid this case in general, as on an uniprocessor system this would mean that no non-real-time process gets enough computation time to do reasonable work as long as the real-time processes are running.

An additional concern is the absence of deadlocks. There are only two locks, fulfilling the same purpose for the real-time and non-real-time side respectively. They make sure there is only one writer active of either kind. As the write process itself just depends on the corresponding lock and it is directly released afterwards, no deadlock is introduced by the synchronisation technique as a whole.

4.4.3. Formal verification

As it is not possible to test every possible interleave of executions that can appear while running AndroitShmem clients, the correctness of the synchronisation technique was proven by creating a model in *Promela* and verifying it using *SPIN*.

Promela and SPIN

Promela [3] is short for "Process/Protocol Meta Language". It is a language for writing models of processes. These models can then be used to verify the logic of parallel systems. The verification is done by SPIN [3], short for "Simple Promela Interpreter". It is a tool for automatically verifying the correctness of models written in Promela. For this, SPIN uses model-checking. More precisely, it generates C code of a program that exhaustively visits every state the model can be in, while checking assertions. The programmer defines these assertions and when they should be checked. In simple words, SPIN can be used to verify that given requirements of a program consisting of several processes are true regardless of how the execution of these processes is interleaved.

The model

The model has to be held as simple as possible, as otherwise the number of possible states quickly becomes too big to be checked in reasonable time. In Section 4.4.2 it was already argued

that neither deadlocks nor infinite loops are introduced under reasonable assumptions. Thus, it remains to show that the synchronisation technique ensures consistency of the data. The only purpose of the two locks was to ensure that only one writer of each type can be active at a time. All in all it is a reasonable choice to omit the locks in the model, it suffices to start at most one writer of each type. Furthermore, as SPIN checks every possible interleave of execution and the readers do not change any shared information, it would suffice to start one reader to show that a reader always reads data consistently. In this case there are two different readers for the real-time and non-real-time part which are checked, although they essentially behave the same way.

Listing 4.9 includes an excerpt of the model to illustrate how the Promela model looks like. It shows the part of the model which represents the real-time writer process. The model is as close to the actual code as possible, to ensure the synchronisation technique is correctly verified. Lines 5 to 9 represent `begin_rt_write()`. It is inlined, because Promela does not support procedures. Furthermore, the locking of the real-time writers' lock is omitted, as explained before. Line 7 increases the global sequence counter by two, which essentially is what `begin_rt_write()` does when locking is omitted. It is enclosed in an atomic block, which causes that everything in this block is seen as one atomic operation by the verifier. The block additionally includes an assertion in line 6 and an assignment in line 8. The assertion checks if the second bit is unset when the real-time write begins. The second bit has to be unset, because this bit is reserved to represent an ongoing real-time write and we only allow one at a time. If the assertion is violated, the verifier would show an error. This would mean the second bit was incorrectly set during execution. The assignment in line 8 stores the value of the increased sequence counter to use it in an assertion later on. Lines 6 and 8 were not part of the original `begin_rt_write()`, but were included to verify the synchronisation technique. Line 11 determines which copy of the shared data is active, just like the actual real-time client would do. Afterwards, it can write to it. Lines 14 to 17 represent `end_rt_write()` minus the locking. The assertion in line 15 was added to verify the model. It checks if the bit, indicating which copy of the shared data is active, has changed since it was stored for the write process. It additionally checks if the sequence counter has the same value as previously set. Both values have to be unchanged, otherwise the verifier would show an error as the assertion would be violated. This would mean the sequence was altered although no process was allowed to. Note that there is no payload data being written to. This is unnecessary, because data integrity is ensured by the sequence counter. It has only to be verified that the sequence counter behaves correctly.

The rest of the model can be found on the attached data medium. It would be tedious to discuss the whole model here, because it is an equivalent translation of the implementation, presented at

```
1  proctype rt_writer() {
2      bit active_data;
3      byte assert_seq;
4
5      atomic {
6          assert((sequence & 2) == 0);
7          sequence = sequence + 2;
8          assert_seq = sequence;
9      }
10
11     active_data = sequence & 1;
12     /* Write to active data copy ... */
13
14     atomic {
15         assert((active_data == (sequence & 1)) && (sequence == assert_seq));
16         sequence = sequence + 2;
17     }
18 }
```

Listing 4.9: Excerpt of the model, showing the real-time write process

length in Section 4.3.2, into Promela. It just additionally includes assertions to ensure integrity of the sequence counter at all times and follows Promelas peculiarities.

The result

The model was used as input by SPIN to generate a C program. With this program, the actual verification could be done. Its output is shown in Listing 4.10. As line 10 states there were no errors, therefore no assertions were violated during the full state search. Lines 25 to 34 reveal that every process reached all of its states, meaning there was no dead code. The rest of the output is statistic about the verification process itself e.g. how many states or state transitions occurred (lines 11-13).

As the verification did not produce any errors and all assertions were positively checked, the model of the synchronisation technique is correct. As the model was equivalently translated from the original implementation, it can be concluded that the implementation is also correct. All in all the synchronisation technique was formally verified in this section.

```
(Spin Version 6.2.2 -- 6 June 2012)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 44 byte, depth reached 95, errors: 0
     5387 states, stored
     5664 states, matched
    11051 transitions (= stored+matched)
      391 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.370   equivalent memory usage for states (stored*(State-vector +
        overhead))
    0.479   actual memory usage for states
  128.000   memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  128.925   total actual memory usage


unreached in proctype rt_reader
   (0 of 19 states)
unreached in proctype nonrt_reader
   (0 of 18 states)
unreached in proctype rt_writer
   (0 of 9 states)
unreached in proctype nonrt_writer
   (0 of 28 states)
unreached in init
   (0 of 7 states)

pan: elapsed time 0.01 seconds
```

Listing 4.10: Verification output

4.5. Conclusion

This chapter first introduced many important notions and concepts in the context of synchronising cooperative processes in Section 4.1. It explained locking mechanism and the associated problems. The presented way to avoid these problems was non-blocking algorithms, more specifically transactional memory as a fitting mechanism for this thesis' purposes in the context of AndroitShmem. Afterwards, in Section 4.2, a sophisticated concept to synchronise AndroitShmem clients as suggested in [13] was explained using the basic concepts introduced in the previous section. In Section 4.3 the concept was slightly adjusted for implementation. Implementation choices and the realisation of the implementation were explained. Section 4.4 analysed the implementation in terms of practicability and did a formal verification of the synchronisation mechanism. In summary, a sophisticated synchronisation mechanism for AndroitShmem was comprehensively introduced, successfully implemented and formally verified in this chapter.

5. Conclusion and prospect

5.1. Conclusion

This thesis introduced fundamental notions and concepts of Android in Chapter 2. Among these were the Dalvik Virtual Machine in conjunction with Zygote, Linux kernel extensions made during the development of Android and the Android Init Language, which is used to configure Android's custom init system. Additionally, real-time systems were explained as a basis to present Androit, the combination of Android and a real-time capable Linux kernel. AndroitShmem, the part of Androit enabling communication between real-time processes and Android applications, was presented in its proof-of-concept state.

Chapter 3 thoroughly discussed the first improvement made to AndroitShmem during this thesis which was enabling AndroitShmem to use the main instead of secondary memory as shared storage. Before the concept was developed and presented, required notions and concepts were introduced. This included memory-mapping a file and the tmpfs file system, which in conjunction were one of the two possible approaches to realise the improvement. The second possible approach was to use Android's own shared memory allocator ashmem. During the development of the concept both approaches were discussed. Storing a file in a tmpfs mount and memory-mapping it exposed to be the superior approach for the purposes of this thesis, because ashmem appeared to be virtually not documented and therefore obscure. The concept using tmpfs was implemented by extending Android's init configuration and the AdroitShmem code. It was followed by performance measurements, which showed a considerable improvement in Androit-Shmems performance when compared to the initial implementation, and underlined that using tmpfs was the superior approach.

The second improvement made to AndroitShmem was comprehensively introduced in Chapter 4. The improvement was to implement a sophisticated synchronisation mechanism. Different basic synchronisation mechanisms were explained in this chapter. Among these were several types of locks. The negative aspects of locks were pointed out e.g. the possibilities of deadlocks and lifelocks or the complexity the handling of locks causes. This lead to the introduction of

non-blocking algorithms with the example of transactional memory. Transactional memory is an important part of the synchronisation mechanism for AndroitShmem proposed in [13]. This synchronisation mechanism was explained in detail. Considerations on how to implement it and choices on which techniques to use were explained. Afterwards, the implementation of the synchronisation mechanism developed in the course of this thesis was profoundly presented. To evaluate the implementation, it was tested and analysed. An equivalent model of the implementation was written for this thesis. It was presented and formally verified. Therefore, it could be concluded that the implementation was correct and the shared data provided by AndroitShmem is kept in an consistent state at all times. Overall, this chapter enabled AndroitShmem clients to access the shared data in an efficiently synchronised way.

In sum, this thesis successfully improved AndroitShmem and has taken it from its proof-of-concept state to being practically usable. Therefore a missing crucial requirement of making Android real-time capable was fulfilled in this thesis.

5.2. Prospect

5.2.1. Further improvements

Version 4.7 of GCC introduced a new set of built-in atomic functions which supports the same operations as the previously existing one, but additionally allows the programmer to choose the memory model. [1] Previously, the memory model was very strict i.e. the order of loads from and stores to memory locations was strictly set by the compiler when using the built-in atomic functions. From GCC 4.7 on, the programmer can choose a more relaxed memory model, allowing the architecture and the compiler to reorder loads an stores more freely. This could potentially be used in AndroitShmem and may increase its performance further. Unfortunately as of this writing, the Android toolchain does not support GCC in version 4.7. Therefore, this improvement requires an updated Android toolchain.

5.2.2. Practicability

The synchronisation mechanism implemented in Chapter 4 was not tested on a real real-time system yet. Furthermore, it was only tested with sample clients accessing primitive variables. Future work has to observe how AndroitShmem and the synchronisation mechanism behave in a real environment with full-featured applications sharing data using it. Especially the assumptions made in Section 4.4.2 and the mentioned issues when having too frequent writing clients

in the same section have to be examined.

A. Contents of the attached data medium

The attached data medium includes the commented source code of AndroitShmem in its initial proof-of-concept state and in its form after the improvements of this thesis were implemented and evaluated. Furthermore, snapshots of all referenced websites, as well as copies of all referenced text files are included. In addition, a digital copy of this thesis is included.

Bibliography

[1] The C++11 Memory Model and GCC. Website: `http://gcc.gnu.org/wiki/Atomic/GCCMM`. [Online; accessed 05-September-2012].

[2] What is Android? Website: `http://developer.android.com/guide/basics/what-is-android.html`. [Online; accessed 19-June-2012].

[3] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer London, 1st edition, 2008.

[4] Patrick Brady. Anatomy & Physiology of an Android. Google I/O Session: `https://sites.google.com/site/io/anatomy--physiology-of-an-android`. [Online; accessed 20-June-2012].

[5] Free Software Foundation, Inc., `http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/`. *GCC 4.6.3 Manual*. [Online; accessed 03-August-2012].

[6] Aeleen Frisch. *Unix System-Administration*. O'Reilly, 2nd edition, 2003.

[7] Dianne Hackborn. Re: [patch 1/6] staging: android: binder: Remove some funny && usage. Mailing List: `https://lkml.org/lkml/2009/6/25/3`, June 2009. [Online; accessed 15-June-2012].

[8] Brian Hall. *Beej's Guide to Unix IPC*. 2010.

[9] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *In Proceedings of the 16th International Symposium on Distributed Computing*, pages 265–279. Springer-Verlag, 2002.

[10] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 1st edition, 2008.

[11] Robert Love. *Linux System Programming*. O'Reilly, 1st edition, 2007.

[12] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox, 1st edition, October 2008.

[13] Wolfgang Mauerer, Gernot Hillier, Jan Sawallisch, Stefan Hönik, and Simon Oberthür.

Real-Time Android: Deterministic Ease of Use. 2012.

[14] Android Open Source Project. Android Open Source Project license. Website: `http://source.android.com/source/licenses.html`. [Online; accessed 15-June-2012].

[15] Goldwyn Rodrigues. Taming the OOM killer. LWN Article: `http://lwn.net/Articles/317814/`, February 2009.

[16] Dave MacLean Satya Komatineni. *Pro Android 4*. Apress, 1st edition, 2012.

[17] Thorsten Schreiber. Android Binder - Android Interprocess Communication. 2011.

[18] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice Hall, 6th edition, 2009.

[19] John Stultz. Anonymous shared memory (ashmem) subsystem. LWN Article: `http://lwn.net/Articles/452035/`, July 2011. [Online; accessed 20-June-2012].

[20] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall International, 2nd edition, 2001.

[21] Wikipedia. Android (operating system) — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=497572426`, 2012. [Online; accessed 15-June-2012].

[22] Wikipedia. Google play — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Google_Play&oldid=497633198`, 2012. [Online; accessed 15-June-2012].

[23] Wikipedia. init — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Init&oldid=500486077`, 2012. [Online; accessed 3-July-2012].