FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Android core process analysis tool on Raspberry PI platform
Student:	Roman Vaivod
Supervisor:	Ing. Pavel Kubalík, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2019/20

Instructions

1) Explore existing projects that support porting of os Android to the Raspberry PI platform.
2) Select the most suitable project and port it on the Raspberry PI platform.
3) Find a good place in the architecture of Android for the placement of the diagnostic tool.
4) Design a tool for performance analysis of Android components.
5) Tool will be developed in C++ language.
6) Tool will allow to trace events happening with specified process in specified period of time and create a list where each entry is described by at least three parameters: CPU usage, memory usage, system/process event.
7) Based on this list, tool will detect possible performance problems in execution of the process and output problematic sequence of events to user.
8) Tool will be written with respect to the next extension.
9) Implement the proposed tool and write tests for it.
10) Create some examples to demonstrate usage of the tool.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 22, 2018

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Science

Bachelor's thesis

Android core process analysis tool on Raspberry Pi platform

Roman Vaivod

Supervisor: Ing. Pavel Kubalik, Ph.D.

15th May 2018

Acknowledgements

I would like to thank my supervisor Pavel Kubalik for being patient with me while I was slowly delivering my results. I am also very thankful to my mother who constantly supported me during my exam sessions and works on the thesis.

Declaration

Citation of this thesis

Vaivod, Roman. *Android core process analysis tool on Raspberry Pi platform.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

The goal of this thesis is to develop a diagnostic tool for components of Android system ported to Raspberry Pi platform. Project consists of two parts. First one is about portation of Android to Raspberry Pi to create an environment for development. Second part is development of diagnostic tool itself. Solution for both parts is based on existing projects. To port Android was used project android-rpi while for tool development were used process memory statistic tool procmem and system events tracer atrace. Result of this project was finding a way to merge procmem and atrace functionality and develop a new diagnostic tool with both memory and performance measurements.

Keywords android, raspberry pi, porting, diagnostic tool, performance, memory usage, linux, kernel, kernel events, cpu usage.

Abstrakt

Cílem této práce je vyvinout diagnostický nstroj pro komponenty systému Android přeneseného na platformu Raspberry Pi. Projekt se skládá ze dvou částí. Prvná cást je o portování Androidu na Raspberry Pi, která vytváří prostředí pro vývoj. Druhou částí je vývoj samotného diagnostického nástroje. Řešení pro obě části je založeno na stávajících projektech. Do portu Android byl použit projekt android-rpi a pro vývoj nástrojů byl použit procesní paměťový statistický nástroj procmem a nástroj atrace určený pro sledování systémových událostí. Výsledkem tohoto projektu je způsob, jak možné sloučit funkčnosti procmem a atrace a vyvinout nový diagnostický nástroj s měřením pamětí i výkonu.

Klíčová slova android, raspberry pi, portace, diagnostický program, výkon, spotřeba pamětí, linux, jádro, událostí jádra, spotřeba procesorů.

Contents

List of Figures

Introduction

Android OS is a leading operating system on the market of handheld devices. There is a big community of developers that constantly develop new applications for Android. The system itself is always evolving and Google provides new releases each year with significant improvements of previos versions.

Internet of Things is a network of many small devices with broad range of uses. Devices in this network have small computers, however, network allows them to effectively exchange data among themselves and use it to improve their performance.

With emergence of Internet Of Things many projects appear that tries to port Android OS to embedded devices because once ported all these features that make Android so popular became available on the device. The importance of Android porting is marked with announcement of Google to finally introduce official support for embedded devices. Android Things 1.0 could be realised in the year of writing of this paper.

Raspberry Pi is a small cheap computer that has just enough resources to use it in many possible ways and realise IoT ideas. Even though Raspberry Pi was started as a project for educational purposes, now it has a huge community of enthusiasts who are constantly creating new ideas of usage for this device.

Limited resources of embedded devices makes them not easy to program. Raspberry Pi is not an exclusion and therefore it's important to have a good diagnostic tool to analyse performance of developed applications.

There exist already many diagnostic tools that could be used on embedded devices with Android, however, most of them focus on analysing process only from cpu usage or memory usage perspective.

This thesis takes as a goal to explore existing projects of porting Android to Raspberry Pi, use one of them to port Android and develop in this environment a new diagnostic tool that will provide analysis of both cpu usage and memory usage of system or application processes in the provided period of time.

Chapter 1 provides an overview of existing partually similar projects. There is a description of projects about porting Android OS to Raspberry

Pi platform and list of existing tools for analysis of process performance in Android environment.

Chapter 2 describes possible options for solution. In particular deeper analysis of pros and cons of existing porting projects and features of diagnostic tools and how they could be used in solution of this thesis task. Also it includes analysis of possible setups of environment for project development, development methodologies.

Chapter 3 contains conclusions of analysis and provides reasons for them. It states what options discussed in previous chapter were chosen for this project. It covers which project for porting was chosen, which diagnostic tools became a base for solution. It builds a plan for solution and measures approximate time for steps.

Chapter 4 discusses the process of solution. It describes how steps from plan described in previous chapter were fullfilled.

Chapter 5 summarises results of the project. It states a degree to which the task was solved. It reviews what was done to achieve that. It provides some thoughts about future project extensions.

Chapter 1

Research of existing solutions

Project consists of two parts. The first one is porting Android OS to Raspberry Pi platform. The second part is development of a diagnostic tool. Therefore, research of existing solutions was made for both parts of the project.

1.1 Existing solutions for porting Android OS to Raspberry Pi

There was no official support by Android of Raspberry Pi platform until recently. That's why all projects were based on work of enthusiasts or volunteers. Only recently Google started a project called Android Things that is a variation of Android for embedded applications.

Here is a list of projects that are covered further in this section:

- Android Things
- RTAndroid (currently EmteriaOS)
- android-rpi
- RazDroid
- LineageOS

Android Things Android Things is a Google project to provide an OS based on regular Android OS for handheld mobile devices with familiar APIs for application development but with support of additional hardware that can be found only on embedded systems.[1] Project considers several platforms, one of them is Raspberry Pi 3.

Current status of project on time of writing this paper is Developer Preview version 8. Official stable release doesn't exist yet but there are rumours that it will be made in 2018. There is a list of issues that are about to be solved. Source code available, however it wasn't prepared for publishing as other official releases in Android Open Source Project. There is a primary documentation support provided on Google developer website with guide for installation. Experimental build is available to download and test on the user platform.

RTAndroid RTAndroid (or newer name is EmteriaOS) is a project based on research made in RTAachen University that grew into commercial product. RT stands here for real-time. This is a real time extension of Android OS for embedded systems.

Current status of project is support of Android Nougat. The source code is not available, however it's possible to download built image and try it on the board.

android-rpi android-rpi is a github repository that is supported by three enthusiasts (one of them is a founder of RTAndroid project) that provides a basic solution for porting of latest releases of Android for mobile devices to Raspberry Pi.

RazDroid RazDroid is an old project that was maintained by volunteers and included usable port for Android version 2 as well as experimental build for Android 4.

Current version of project is outdated. Source code available online but the latest changes were made several years ago. It seems like project is not supported anymore.

LineageOS LineageOS port for Raspberry Pi is an example of port of flavour of Android OS. It's based on android-rpi repository and LineageOS extensions for Android.

Current version uses Android Nougat port along with latest version of LineageOS. Source code available with documentation how to compile and build it.

1.2 Existing solutions of diagnostic tools for Android OS

There exist a plenty of tools that are developed and maintained by Google for debugging and analysis of system performance.

Android uses Linux kernel and, even though it has some patches applied to it, it's possible to use also Linux diagnostic tools.

Here is a short list of selected tools:

- Android Debug Bridge

- dmesg
- logcat
- dumpsys
- systrace
- atrace
- ftrace
- procmem
- perf
- simpleperf

Android Debug Bridge Android Debug Bridge (ADB) is a software to create a "bridge" between your device and developer. It connects to device using TCP/IP and provides an option to run a remote shell.

Other mentioned tools are executed from the shell provided by ADB. This makes ADB a necessary tool for performing any analysis for system performance on Raspberry Pi.

dmesg dmesg is an internal tool provided in Linux Kernel to see output from kernel boot. It outputs debug information with attached timestamps that are counted from start of system initialization process. It is useful to analyse the performance and functionality of kernel services.

logcat logcat is a built-in Android tool that contains all debugging logs of the system. It can be used to output debug messages from Android System components and developer application as well.

dumpsys dumpsys is a tool that can be executed from shell environment on the target device and it outputs (dumps) information of all system services and running applications.

systrace systrace is a python application that uses ADB to run another tracing tool called atrace on target device for period of time, then reads trace report and generates html with comfortable javascript interface called TraceView to view timespan of processes with their state information. This tool is used for performance issues of Android components and applications.

atrace atrace is a system for analysing performance of Android System components and kernel system. atrace uses ftrace Linux tool to collect information from Android system and kernel at the same time.

ftrace ftrace is a very powerful Linux kernel function tracer. It can trace any function in kernel space and relate them to process that function runs in, specify on which CPU runs this process, display timestamp. It also can record kernel tracepoints that define kernel space events with processes.

malloc debug Malloc Debug is a Android system debugging tool for heap allocations. It can detect user space memory leaks or memory corruption issues in Android applications or native platform code.

meminfo meminfo is a service for collecting process memory information including Virtual Set Size(VSS), Resident Set Size (RSS), Proportional Set Size(PSS) and Unique Set Size(USS). The easiest to analyse are sizes USS and PSS. USS says how much memory will deallocated at the same time as application will be terminated. PSS is shared with other processes and only relative part of it will be freed.[2] It maintains history of process memory statistics in last 3 hours and one of the easiest ways to use is to dump information from this service using dumpsys.

procmem procmem is a memory profiling tool that displays similar information as meminfo for a specified process, however with more details. In particular, it shows memory counters for each shared library in the process.

simpleperf simpleperf is a simple implementation of Linux tool called perf that also includes some fixes specific for Android environment. It's a modern tool for performance analysis of processes running on modern CPUs supplied with PMU.

Chapter 2

Analysis

This section provides analysis of projects and tools that were found during research in terms of their relevance and possible usage for the task of this project.

2.1 Analysis of existing solutions

For the development of Android diagnostic tool on Raspberry Pi platform it's necessary to have ported Android OS that complies with following requirements:

- functional version with regular shell toolbox support
- possibility to connect from host to shell with root access
- working native diagnostic applications
- modern version starting from Android 7 (Nougat)
- available source code
- available documentation of changes to original Android version
- project is up-to-date and supported by its community

For the development of tool it's crusial to have stable system with ability to easy connect to shell with root access. Many diagnostic tools rely on root access for accessing diagnostic files of kernel. Root access in Android gives essential rights to setup system for debugging purposes. It's necessary that system already supports some diagnostic tools that are officially released to be able to verify results of developed tool. It's also could be necessary to use their functionality to implement necessary features. Therefore, ported system should support memory usage and cpu usage measuring tools with available source code and documentation. Project should be supported by community to provide ability to ask possible questions regarding implementation of port.

Similarly, documentation of changes made to Android could be good for understanding better environment and comparing performance of tool in regular supported platforms.

2.1.1 Analysis of porting projects

Most of projects mentioned in previous chapter were excluded from analysis. For RazDroid the reason is that it's out-of-date unsupported project. For LineageOS it is a different flavour of Android that is based on android-rpi and it is a only personal project available on github. RTAndroid is a version of Android for user experience only. There is no documentation for developers and its closed source commercial project. Therefore, two projects remain to consider - Android Things and android-rpi.

Android Things Android Things supports Raspberry Pi 3. There is a good documentation on how to setup a development on the device on Android developers website. It is possible to connect to shell using ADB as long as device is connected to network and assigned some IP address. Shell gives common functionality that could be expected from Linux shell with default toolbox and diagnostic applications. Its code is available in Android Open Source Project repository.

However, one issue is that Android Things is a new project in development preview state with special changes of usual Android mobile concepts from Google for embedded devices. This gives some differences with regular Android development that are described on developer website. For example, one of them is single app experience for user.[1] The project is not stable, interfaces for development are not fixed. Moreover, project is always evolving and it's not possible to rely on things that could be changed in next developer preview after one month.

android-rpi Along with a source code that includes build configuration files, patches for system and Linux kernel, this project also gives a basic guide how to compile, build and run Android image on Raspberry Pi 2 and 3.

Solution of this project includes functionality like Launcher application with several applications (for example, WebView), support of WiFi and Ethernet, HDMI monitor, mouse and keyboard as substitute of navigation buttons. It's possible to connect to shell with root access using ADB through the IP address that can be, for example, viewed in Settings application using monitor.

Current stable version of project is Android Nougat (android-7.1.2r19 branch). There exists also initial version for latest release of Android called Oreo but it doesn't support all features that are mentioned above. Repository is gradually maintained. Changes to kernel are not documented and patches to android system are minimal (just several lines).

2.1.2 Analysis of diagnostic tools

Tools like ADB, dmesg, logcat that were mentioned in previous chapter will be excluded from deep analysis in this section. Instead they will be mentioned in the Development setup section. They are essential for the development process. In this section analysis will cover tools that have common functionality to the task of this thesis.

2.1.3 systrace

As mentioned before systrace is a tool that uses atrace tool from host through ADB to collect trace information of target device and parses it to user in a friendly html interface powered by javascript frontend Trace-Viewer.[3]

Systrace mimics interface of atrace to control what should be included in trace report. These options will be covered in more detail in analysis of atrace. In short, user can specify what kind of events he wants to trace from the ones that are available on target device. There are kernel and android events. Kernel events are defined in documentation of ftrace, while android events are not defined so well, more as a list of categories. This categories correspond to Android components.

This functionality allows user of systrace to see what happens in different Android components in correlation with what happens in kernel.

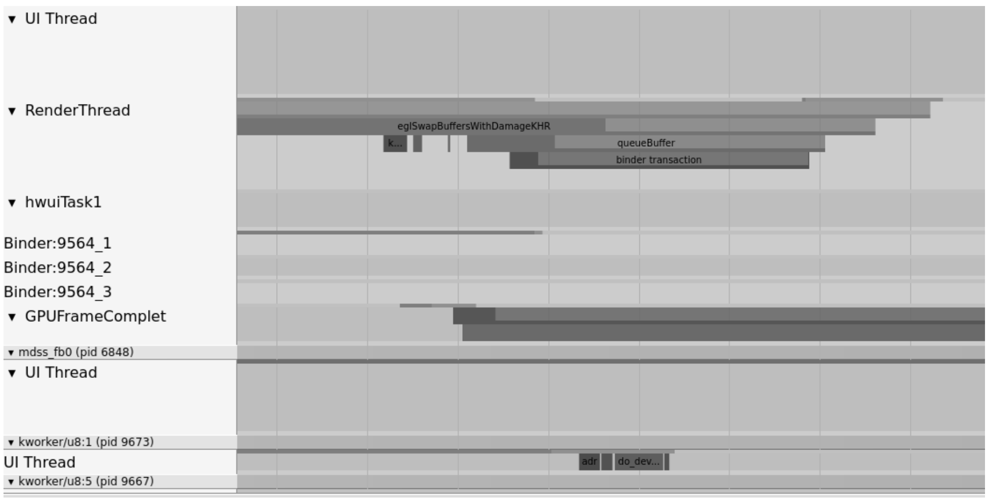


Figure 2.1: Screenshot of Trace-Viewer that runs on trace data collected by systrace

Systrace is written in python, code is available in Catapult project on github as well as in AOSP. It is written in OOP style and is easy to extend. However, most functionality of code is about running ADB commands from shell, collect output from device and save it in report file.

2.1.3.1 atrace

Overview Atrace is a tracing system of Android. The idea of the tool is to run ftrace with some predefined grouped trace events and at the same time capture tracing information from system like, for example, calls of Android service methods and put all together in the same trace time sequence.

Atrace is a shell command. For Android events there are options to specify what kind of event categories to trace in the system. These include:

- Graphics
- Input
- View System
- WebView
- Window Manager
- Activity Manager

This categories are not defined precisely while 'Graphics' defines some group of processes that render graphics, 'Activity Manager' defines events that happen with Activity Manager service. It's possible though to find out what code will log to these categories on existing AOSP code database using this grep command:

```
$ grep -r ATRACE_TAG frameworks
```

It will output many source files that define and use macros corresponding to these categories.

It is also possible to include Android and native applications to trace by putting code functions calls in the beginning and end of methods.[4] For application it's also necessary to specify the package name of the application to trace with the -a or –app command line option of atrace.

Source code As per android 7.1.2r19 source code in AOSP is in 'frameworks/native/cmds/atrace' directory. Dependencies except linux and C libraries are defined in Android.mk file from the same directory and look as follows:

- libbinder
- libcutils
- libutils
- libz

Library libbinder is used to get Service manager through Binder interface and send transactions to services to update their system properties values.

Functions to set system properties are defined in libcutils. Trace tags that define which Android categories to trace are defined in libutils (more about purpose of using these dependencies is in 'Android interaction' paragraph of this section). libz provides functions for compression of trace output.

ftrace role ftrace is used only partially by atrace. Tracing system enables ftrace files that control what kind of event will be traced by kernel and put into buffer. These files are defined in array in source code.

To relate events happening in kernel with the ones that are triggered in Android system, atrace writes event information in the same file as ftrace. This way it's possible to see what was happening in kernel at the time of event in Android system.

Android interaction As was mentioned earlier atrace allows to monitor events that happened in Android system. To achieve that atrace application uses atrace tags that are defined in `system/core/include/cutils/trace.h`. It ORs them in one integer and stores in the property called **debug.atrace.tags.enableflags**.

If user specifies some application packages to trace then the name of package is placed in separate property prefixed with **debug.atrace.app_** and ended with order number of application to trace. Moreover, it will set **debug.atrace.app_number** with number of applications to trace. For example, for option **-a com.example.app,com.example.otherapp** atrace will set property **debug.atrace.app_0** to **com.example.app** then next property **debug.atrace.app_1** to **com.example.otherapp** and, finally, count in **debug.atrace.app_number** to 2. After setting the properties above atrace makes a transaction to services through Binder to update their system properties records.

Execution Application side of atrace system that starts tracing doesn't have to collect information steadily from some pipe unless user wants to stream trace in the standard output directly. It just setups tracing according to user preferences described per arguments and then sleeps until interrupt or time specified by user runs out. Tracing is done behind the scenes by ftrace and Android System. Waked up atrace process stops tracing by cleaning up properties and configuration files and collect everything from ftrace buffer file.

2.1.3.2 ftrace

Overview Being at first a kernel function tracer, ftrace can trace any function in kernel by specifying it's name. It's possible to output call stack, analyse if there was some latency, find out at which CPU and process function was called.

At second ftrace defines tracepoints that are more general then kernel functions and make it easier to trace specific events with CPU, like, for example, switches of processes.

Usage Interface is provided via filesystem that depending on kernel version could be mounted as tracefs or debugfs in `/sys/kernel/tracing` or `/sys/kernel/debug/tracingrespectively`:

```
$ mount -t debugfs debugfs /sys/kernel/debug
```

To control options or start and stop tracing its enough to echo specified values in files under these folders.

trace-cmd In addition to ftrace there exists trace-cmd shell tool that provides friendly interface by specifying options with regular arguments instead writing them in files of debug filesystem.[5]

Format Typical format of ftrace output is a table with headers:

```
# tracer: function
#
# entries-in-buffer/entries-written: 140080/250280   #P:4
#
#                        _-----=> irqs-off
#                      / _----=> need-resched
#                     | / _---=> hardirq/softirq
#                     || / _--=> preempt-depth
#                     ||| /     delay
#   TASK-PID   CPU#   ||||    TIMESTAMP  FUNCTION
#      | |       |    ||||       |         |
    bash-1977  [000] .... 17284.993652: sys_close <-system_call
    bash-1977  [000] .... 17284.993653: __close_fd <-sys_close
```

Figure 2.2: Example of ftrace log in trace buffer available for user

The meaning of data is intuitive. ***tracer:*** shows what kind of tracer is used in ftrace setup. It affects what kind of information is printed in last column that is now called as FUNCTION. Buffer size could be set in options and in this example it's size is not enough to store all 250280 events but only 140080. #P:4 shows number of online cpus. Three first columns are straitforward but forth one has some special values. Their meaning could be found in ftrace documentation, however, as stated there, these values are mostly meaningful for kernel developers and they won't be covered here.[6]

In summary in this example it's possible to find out which function was called on which CPU at what time and in which process. Moreover, also a parent function is specified. ftrace was made to trace kernel functions and it's very powerful for this purpose.

Sched tracepoints One of the interesting tracepoints for ftrace is possibility to trace switches of processes on CPUs by using group of events in `tracing/events/sched` folder (relative path to root folder of debugfs filesystem).

```
sched_wakeup: comm=SensorService pid=1342 prio=89 success=1
  target_cpu=000
sched_switch: prev_comm=swapper prev_pid=0 prev_prio=120
  prev_state=R ==> next_comm=SensorService next_pid=1342
  next_prio=89
sched_wakeup: comm=android.ui pid=1326 prio=118 success=1
  target_cpu=000
sched_switch: prev_comm=SensorService prev_pid=1342 prev_prio=89
  prev_state=S ==> next_comm=android.ui next_pid=1326
  next_prio=118
sched_switch: prev_comm=android.ui prev_pid=1326 prev_prio=118
  prev_state=S ==> next_comm=swapper next_pid=0 next_prio=120
sched_wakeup: comm=atrace pid=2394 prio=120 success=1
  target_cpu=000
sched_switch: prev_comm=swapper prev_pid=0 prev_prio=120
  prev_state=R ==> next_comm=atrace next_pid=2394 next_prio=120
```

Figure 2.3: Example of ftrace trace with sched events only enabled (format was changed insignificantly for illustration)

In this example it's possible to see life events of process on CPU. ftrace tracepoints sched_switch, sched_wakeup give information when CPU switches from one process to another with names of involved processes.

Memory allocations Another kind of tracepoints is kmemm that captures events related to object and page allocation inside the kernel. Here is example of capturing events for kmalloc calls.

In the example on figure 2.4 it's possible how much memory a process requested in kernel space and how much were allocated in reality.

2.1.3.3 Malloc Debug

Mallog-debug is a debugging tool to deteck memory issues such as, for example, memory leaks or memory corruption. This tool is implemented by

```
 Binder:1543_1-1557  ( 1543) [000] .N.2  9709.880000:
    kmalloc: call_site=ffffffff81432ecf ptr=ffff88000c8a8140
    bytes_req=24 bytes_alloc=32 gfp_flags=GFP_ZERO
 Binder:1543_1-1557  ( 1543) [000] ...1  9709.880000:
    kmalloc: call_site=ffffffff812045a3 ptr=ffff88000cbfa000
    bytes_req=4096 bytes_alloc=4096 gfp_flags=GFP_KERNEL|GFP_ZERO
 Binder:1543_1-1557  ( 1543) [000] ...1  9709.880000:
    kmalloc: call_site=ffffffff811b8e90 ptr=ffff88000c8a8140
    bytes_req=24 bytes_alloc=32 gfp_flags=GFP_KERNEL
 SensorService-1342  ( 1311) [000] ...1  9709.910000:
    kmalloc: call_site=ffffffff81455be1 ptr=ffff88000a510000
    bytes_req=448 bytes_alloc=512
    gfp_flags=GFP_KERNEL|GFP_NOWARN|GFP_REPEAT|GFP_NOMEMALLOC
 SensorService-1342  ( 1311) [000] ...1  9709.980000:
    kmalloc: call_site=ffffffff81455be1 ptr=ffff88000a510000
    bytes_req=448 bytes_alloc=512
    gfp_flags=GFP_KERNEL|GFP_NOWARN|GFP_REPEAT|GFP_NOMEMALLOC
 SensorService-1342  ( 1311) [000] ...1  9710.040000:
    kmalloc: call_site=ffffffff81455be1 ptr=ffff88000a510000
    bytes_req=448 bytes_alloc=512
    gfp_flags=GFP_KERNEL|GFP_NOWARN|GFP_REPEAT|GFP_NOMEMALLOC
```

Figure 2.4: Example of ftrace trace with enabled kmem:kmalloc event

Android System and have been significantly updated starting from Android Nougat. It's possible to record memory issues of native code as well as applications running on Dalvik machine.

Usage Interface of this tool is simple. It's necessary just to set one or two system properties that setup debugging and restart the device with new properties using ADB shell. For example, with the following commands, it's possible to start debug on all processes in the system:

```
adb shell stop
adb shell setprop libc.debug.malloc.options backtrace
adb shell start
```

That makes the debugging starts. Possible issues can be found in logcat output. Also it's possible to dump collected data for a specific process in the file using this command:

```
adb shell am dumpheap -n <PID_TO_DUMP> /data/local/tmp/heap.txt
```

Practice Attempts to use the tool on Android Nougat were not always successful. By searching solutions on the web it seems that Nougat version isn't supported well even though documentation states otherwise.[7]

During analysis basic options that were described above worked but only partially. It wasn't possible to get output in logcat. Dump of heap, however, worked. The format is described in detail in official documentation. Most important are first three lines that give amount of total memory of live allocations in heap and number of allocations records.

There exists API with two functions that could be used to receive dump in the program. Example of usage was found on line 838 in `android_os_Debug.cpp` that could be found in `frameworks/base/core/jni` folder.

2.1.3.4 procmem

Overview procmem is a simple tool that can collect memory information about specified process. It outputs memory statistics for current moment in time.

Data is presented as a table where each row correspond to each component in process that uses memory, for example, shared libraries. Each row contains RSS, PSS, USS, shared clean, shared dirty, private clean and private dirty page memories values.

Source code Source code of the tool is available in 'system/extras/procmem'. The tool is written in C language. Code is short, clean and easy to understand. Main dependency is on pagemap library. This library was written specifically for Android environment. Source code of library can be found in `system/extras/libpagemap`.

Structures that were used from pagemap library in procmem:

```
pm_kernel_t
pm_process_t
pm_memusage_t
```

Structure kernel is the main structure that initializes basic information for a library to function. Using kernel it's possible to obtain a specified by PID process. Process provides access to its memory usage through the structure memusage.

2.1.3.5 simpleperf

Overview simpleperf is a powerful tool to profile processes running on CPUs with PMU for a specific time period.[8]

It's basic output is a list of processes with assigned number of event samples, cpu number and so called overhead amount that is a percentage of events samples count to all recorded event samples during tracing period.

```
Samples: 1554 of event 'cpu-clock'
Event count: 388500000

Overhead  Command     Pid   Tid   Shared Object
99.55%    swapper     0     0     [kernel.kallsyms]
0.39%     simpleperf  2427  2427  [kernel.kallsyms]
0.06%     simpleperf  2427  2427  /system/xbin/simpleperf
```

Figure 2.5: Example of output of simpleperf report on collected perf.data

It supports different kinds of events on CPU including hardware, software and kernel tracepoints.

Source code quality Source code of simpleperf is available at `system/extras`. For android-7.1.2r19 (this is a branch that is used by android-rpi project) it is not easy to understand and hard to read. Code is written in C++, purely structured. On master branch of the project code is partially refactored and cleaned up. More methods and classes are documented. Unfortunately this code is dependent on different libraries that are not present in android-7.1.2r19 and it's not easy to compile it for Android Nougat.

Usage of Linux API Application uses linux perf events interface.[9] It constructs, for each user specified event, perf_event_attr structure that is defined in 'linux/perf_event.h'. This structure apart of containing id of event to trace also specifies different options for which information to collect. Once defined, attr structures are used in system call to kernel with perf_event_open function. It returns a file descriptor that can be used to read samples.

On figure 2.6 there is an example of usage of perf_event. Option mmap enables usage of mmap buffer for samples reading. Option sample_type defines what fields should be included in a sample structure. Option type defines type of event to trace, while config is an id for a concrete event. Option sample_freq enables tracing by frequency. As an alternative, option sample_period enables sampling in periods where period is an amount of event executions. In other words, event will be traced after period number of occurances.

Reading of samples is done using perf_event_mmap_page structure that is instantiated using obtained file descriptor by calling linux function mmap.

Then simpleperf in a loop uses call to poll with created poll_fd structure to wait for available data. Once some data is available code reads it from allocated perf_event_mmap_page structure. This structure defines a ring buffer. After it's done, read raw data is processed based on perf_event_attr and stored in record file. The process repeats in the loop until some signal from system or user fired.

```
struct perf_event_attr pe;
pe.mmap = 1;
pe.sample_type |=PERF_SAMPLE_IP
               | PERF_SAMPLE_TID
               | PERF_SAMPLE_TIME
               | PERF_SAMPLE_PERIOD
               | PERF_SAMPLE_CPU;
pe.type   = PERF_TYPE_TRACEPOINT;
pe.config = 52;
pe.sample_freq   = 0;
pe.sample_period = 1;
pe.exclude_user   = 0; // disable user space events
pe.exclude_kernel = 0; // disable kernel space events

fd = perf_event_open(&pe,
                      0,  // any process
                     -1,  // any cpu
                     -1,  // not useful
                      // close file descriptor on exit
                      PERF_FLAG_FD_CLOEXEC);
```

Figure 2.6: Example of perf_event_open usage

It's crucial to read samples fast because otherwise it's possible to miss some information due to ring buffer size limitations. For this purpose simpleperf creates several processes to collect and read sample data. It also creates it's own record cache to allow to accumulate samples and save some time.

2.2 Development setup

This section will describe options for development kit and software development process setup. It discusses options for choice of Raspberry Pi board and describes building tools in Android environment.

2.2.1 Development kit

Development kit includes the following:

- Raspberry Pi board with power supply and SD card
- host computer with enough resources to store and build AOSP
- (optional) WiFi network, HDMI display, USB mouse and keyboard
- (optional) Ethernet cable

- (optional) USB-USB cable

Android could be built as a light weight version, however, porting projects try to port modern version of Android that requires more resources.

Project android-rpi supports Raspberry Pi 2 and 3, while Android Things support only Raspberry Pi 3.

Collection of debugging information involves execution of additional instructions and that could require better resources from hardware.

To be able to download and build AOSP it's essential to have enough memory on host computer (at least 250 GB). Officially recommended environment is 64-bit Linux, in particular, Ubuntu 14.04 for Android Nougat. Processor should be powerfull enough because builds of Android system could take up to several hours on not enough fast CPUs. It's recommended to have enough RAM for a build (2GB at least).[10]

If target device can connect to Wifi then it's possible to connect to it using IP address. To establish connection and find out IP address it's possible to use Settings application. Therefore, for this setup it's necessary to have WiFi network, HDMI display, USB mouse and keyboard.

Other way to connect is to use Ethernet cable but Android Nougat by default limits connection for Ethernet and futher analysis is needed to find a solution how to setup connection on device.

USB connection could be used if ported solution supports it.

2.2.2 Development process

For development it's necessary to have following options:

- build tool's binary for target device
- distribute tool's binary to target device
- access to shell with root access on device
- run tests both on host and target machines

All these options are provided by Android Open Source project.

repo repo is a python script designed specifically for Android repository that simplifies work with git repositories of different Android projects. It's easy to install.[11] By default it's configured to work with Android source repositories. It's also possible to configure it for other online repository by a configuration file default.xml in '.repo/local_manifests' folder. Configuration allows to define a concrete part of project in directory structure of AOSP.

Android Build System AOSP repository comes with very agile build system. It uses 'make' automation tool to build everything inside its repository. Each executable or library has an Android.mk file that defines for build system a way to build this library or executable. By defining a name of executable in Android.mk it's possible to run make just to build this executable.

System also provides an interface to choose a target for which executable or library should be compiled. This option allows to compile unit tests both for host and target. Running tests on host could save a bit of time for developers.

ADB ADB is an essential for development process. Except possibility to connect to shell with root permissions on target device, it also provides possibility to transfer any file from and out of device. There is also an option to run commands on device without leaving host shell. ADB can easily connects to several devices at a time, including emulators.

Emulator AOSP repository also comes with set of emulators of devices. Developers could choose one of the available ones and use them as a real devices, though some debugging information could be meaningful only for emulator. Emulator allows to continue development even without access to device and that is very valuable with possibly complicated development kit of embedded developers.

2.3 Extension requirements

One of the goals of this project is to develop diagnostic tool that will be easy to extend. Therefore, the following requirements could be defined:

- good documentation
- clearly separated software layers
- tests
- OOP best practices

Good documentation of code will allow developers of extensions to faster understand the code. Moreover, documentation can define some policy for contribution that will help to keep code in a clean state.

Clearly separated software layers will allow to introduce minimum changes for addition of new functionality. By knowing at which layer introduce the change future developers would be sure that they are not breaking old functionality as long as they don't interrupt interface between layers.

Tests provide a resistance of code to new changes. If developer would have to introduce a change in existing piece of code then he will immediately know if this change breaks prior existing functionality or not.

OOP allows to understand and extend code easier by keeping classes small, providing encapsulation concept to preserve behaviour of existing classes and avoid duplication with inheritance and polymorphism, making code more general and abstract.

CHAPTER 3

Design

This section describes decisions and plan for solution with reasons based on results of analysis made in previous chapter.

3.1 Porting project's choice

As a basis for porting Android was chosen android-rpi project. It complies with most requirements stated to porting project.

The biggest disadvantage of Android Things is that it's an experimental project with new concepts. Possible issues could complicate the solution because this version of Android is not yet well known by people other then developers.

In contrast android-rpi ports well-known version of Android after it's 19th release. It's practically guaranteed that there will be no issues with usage of existing diagnostic tools mentioned in second part of analysis.

3.2 Design of diagnostic tool

Chosen base tools For implementation of diagnostic tool of thesis were chosen atrace tool and procmem.

atrace is a powerful tool to record sequence of events for processes that also allows to trace how much of cpu-time was assigned to concrete process. Its implementation is easy to understand and all tracing is done under the hood. The fact that application just sleeps during the trace gives a space for putting this time to some other work, for example, collecting information about memory usage.

Another candidate for usage could be simpleperf that is a powerful tracing tool of events. However, it's very complicated to understand its source code. Application has to put a lot of effort to record events on CPUs compared with atrace. It's also possible to tweak existing functionality and obtain a sequence of events on process, however, there is no space to put extra functionality like recording of process memory usage. Unfortunately, perf events provide only ability to get user stack size.

procmem is a simple tool with clean source code that gives a good example how to obtain memory statistics for processes from native code.

Solution idea The idea of solution to this thesis is to use available process time of atrace to gather statistics in a similar way as procmem does and log it refered to kernel events in the manner as atrace does for Android components. This way trace will contain sequence of events in kernel merged with events in Android system and memory statistics of processes. To get cpu usage statistic it's possible to use sched_switch tracepoint and obtain time intervals when process was actually running on CPU. Summing these intervals and dividing by total amount of tracing time gives us a CPU usage of process.

3.3 Development kit

The chosen development kit for this project includes Raspberry Pi 3 and all other possible options that were described in the analysis section.

android-rpi project provides two ways to connect to shell through ADB and it's very likely that at least one of these options will work.

3.4 Development process

Development process will highly rely on tools provided by AOSP repository. Source code of application should be stored in a separated available online repository like github and pulled into Android using repo.

For building will be used Android Build system because it allows to use any native Android libraries that are necessary.

To distribute application and run it for testing on the target device will be used ADB tool.

There is no emulator for Raspberry Pi 3 in AOSP but most of the time it will be enough to test tool functionality on emulator with arm platform. Functionality of diagnostic tool is not bound only to Raspberry Pi 3 platform.

3.5 Development plan

Proposed plan to implement diagnostic tool with approximate workload per step is as follows:

1. porting of Android to Raspberry Pi 3 using android-rpi project (one-two weeks)

2. testing of tools procmem and atrace (one week)
3. refactoring of atrace for extension (two weeks)
4. refactoring of procmem for extraction of necessary functionality (two weeks)
5. merging functionality of procmem into atrace (one week)
6. testing of developed tool, fixing possible bugs, completing unit tests (one week)
7. development of trace output processing layer (one week)
8. development of algorithm to calculate problematic sequences of events (one-two weeks)
9. testing of developed tool, fixing possible bugs (one week)

3.6 Development approach

For development steps several approaches will be used.

To refactor existing code of atrace and procmem, an approach of working with legacy code will be used as both atrace and procmem doesn't have any tests. Therefore, at first there will be created some approach to test their functionality. Then pieces of relevant logic will be gradually encapsulated in classes until all code will be refactored.

Along with refactoring will be maintained simplified UML class diagrams. For Unit tests will be used Google Test framework.

Chapter 4

Solution

This sections describes how steps of the plan developed in previous chapter were implemented.

4.1 Porting of Android to Raspberri Pi 3

All steps were made according to guide of android-rpi3 project.

4.1.1 Downloading Android source with kernel patches

To download Android source it was necessary to install repo first. This was done using guide in Android Source website.

Once repo was installed, the first serie of commands from android-rpi3 guide was used:

```
$ repo init -u https://android.googlesource.com/platform/manifest
-b android-7.1.2_r19
$ git clone https://github.com/android-rpi/local_manifests
.repo/local_manifests
$ repo sync
```

Init command initialises repo to prepare it for sync with Android 7.1.2 release 19 branch. This command downloads some configs for repo into .repo folder.

Next command clones repository from android-rpi3 project that contains new config for repo to download kernel patched for Raspberry Pi 3.

Last command runs repo and downloads Android repository into directory where commands were run. This command could take for about several hours. To make things quicker -j option was used to specify a number of threads for repo:

```
$ repo -j8 sync
```

4.1.2 Building Kernel

After repo downloads all sources next step was to build kernel. To compile kernel for Raspberry Pi it was necessary to install first gcc-arm-linux-gnueabihf and then run following commands as per android-rpi3 guide:

```
$ sudo apt-get install gcc-arm-linux-gnueabihf
$ cd kernel/rpi
$ ARCH=arm scripts/kconfig/merge_config.sh
arch/arm/configs/bcm2709_defconfig
android/configs/android-base.cfg
android/configs/android-recommended.cfg
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make zImage
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- make dtbs
```

Note that commands that are present on one line should be copied and executed this way. For example, ARCH=arm is not a separate command but arguments for make or script merge_config.sh.

First command is to move to kernel/rpi directory that contains kernel source code cloned from repository of android-rpi3 project.

Second command uses merge_config.sh to merge kernel configs defined specifically for Raspberry Pi's BCM2709 board with base and recommended flags from android. This defines what flags will be used in build of kernel.

Third command compiles zImage that is compressed self-extracting image of patched Linux kernel. Fourth command compiles device tree binary - a low level description specific to the device.

For all commands ARCH parameter is used to specify architecture of CPU to compile kernel to. Moreover, for make commands CROSS_COMPILE command is given to specify cross compiler for target. Here is used gnueabihf where gnu stands for linux, eabi stands for embedded ABI and hf stands for hard float. This choice of compiler means that code will be compiled for, so called, bare metal arm core with hardware floating point support. make dtbs stands for compiling multiple dtb or device tree binaries from dts files or device tree sources that make takes from standard location 'arch/arm/boot/dts'.

To make things quicker a number of threads were provided to use for 'make' in the same way as for repo.

4.1.3 Building Android System

Patches Several patches as it is proposed by project were applied to Android System source code.

Mesa VC4 Mesa VC4 is a graphics driver for Raspberry Pi 3 used in build of kernel. As it doesn't support RGBA8888 format but uses BGRA8888 with Depth 24 or 0 and Stensil 8 or 0 it's necessary to apply following patches:

```
// frameworks/native/opengl/libs/EGL/eglApi.cpp:478
EGLSurface eglCreateWindowSurface(  EGLDisplay dpy,
                                    EGLConfig config)
-            format = HAL_PIXEL_FORMAT_RGBA_8888;
+            format = HAL_PIXEL_FORMAT_BGRA_8888;
        } else {

// frameworks/native.opengl/libs/EGL/eglApi.cpp:1843
EGLClientBuffer eglCreateNativeClientBufferANDROID
                                  (const EGLint *attrib_list)
        if (alpha_size == 8) {
-            format = HAL_PIXEL_FORMAT_RGBA_8888;
+            format = HAL_PIXEL_FORMAT_BGRA_8888;
        } else {

// frameworks/native/opengl/java/android/
                                opengl/GLSurfaceView.java:976
public class GLSurfaceView extends SurfaceView
                           implements SurfaceHolder.Callback
      public SimpleEGLConfigChooser(boolean withDepthBuffer) {
-          super(8, 8, 8, 0, withDepthBuffer ? 16 : 0, 0);
+          super(8, 8, 8, 8, withDepthBuffer ? 24 : 0, 0);
      }
```

Figure 4.1: Mesa VC patch

HW video decoder As HW video decoder is not supported in this porting project, it's necessary to use SW video decoder instead with limited performance. Patch is presented on figure 4.2.

Eluminate screen flashing There exists a Strict mode of operating system that flashes screen when applications do too long operations in main thread. This mode is enabled by default is 'eng' build variant that will be used to build Android System. Little modification to StrictMode class could fix this issue. Patch is presented on figure 4.3.

Bluetooth timeout fix Add new bigger timeout for Bluetooth. Patch is presented on figure 4.4.

Build system dependencies Per guide it's necessary to install python-mako for building system. However, after several attempts of building android source on Ubuntu 16.04, several errors were encountered that demanded in-

```
// frameworks/av/media/libstagefright/colorconversion
                                     /SoftwareRenderer.cpp:113
void SoftwareRenderer::resetFormatIfChanged
                                   (const sp<AMessage> &format) {
     case OMX_TI_COLOR_FormatYUV420PackedSemiPlanar:
     {
-            halFormat = HAL_PIXEL_FORMAT_YV12;
-            bufWidth = (mCropWidth + 1) & ~1;
-            bufHeight = (mCropHeight + 1) & ~1;
         break;
     }
```

Figure 4.2: Switch to SW video decoder

```
// frameworks/base/core/java/android/os/StrictMode.java:1068
public final class StrictMode {
     if (IS_ENG_BUILD) {
-            doFlashes = true;
     }
```

Figure 4.3: Screen flashing patch

```
// hardware/broadcom/libbt/src/hardware.c:232
 static const fw_settlement_entry_t fw_settlement_table[] = {
     {"BCM43241", 200},
     {"BCM43341", 100},
+    {"BCM43430A1", 1000},
 };
```

Figure 4.4: Bluetooth patch

stallation of bison and libxml2-utils. For installation of bison it's also necessary to install libraries libc6 i386, libncurses5 i386 and libstdc++6 i386.

Environment setup Android build system provides two commands to setup environment for the build that should be used always before running make.

The first command is running a shell script build/envsetup.sh. It adds new functions to run from shell for developer. Among them the second command called 'lunch' as well as some other useful commands for quickly navigation to root of AOSP repository, some custom build commands and grep commands for a specific purposes.

'lunch' is used to specify a target device to build source for. To use it's necessary to call command with a combo name. These combo names correspond to target devices and are added by `build/envsetup.sh` that finds all vendorsetup.sh files in 'device' directory and execute them. Each script vendorsetup.sh defines what combos to add for 'lunch'.

Building After environment is set and necessary dependencies are installed it's possible to start the build. It necessary to build ramdisk and system image. Android ramdisk has the same purpose as Linux ramdisk - it's a filesystem that is mounted first for kernel that allows to start init to mount all other system. System image is basic file system for Android that includes all necessary files, applications, frameworks, the Dalvik VM. However, this image doesn't include kernel or ramdisk. That's why it's necessary to build those separately

Out of memory error One issue during the build was caused by the fact that Jack Server Virtual Machine was running out of memory. Several options were tried to set bigger memory limit for Jack Server, however, only exporting environment variable JACK_SERVER_VM_ARGUMENTS with necessary arguments and restarting the server if it's already running to pick up this variable worked.

Commands summary On first build 'make' step took the most time from all building process commands even though -j option was used as before.

```
$ sudo apt-get install python-mako
$ export JACK_SERVER_VM_ARGUMENTS="-Dfile.encoding=UTF-8
                     -XX:+TieredCompilation -Xmx4096m"
$ out/host/linux-x86/bin/jack-admin kill-server
$ out/host/linux-x86/bin/jack-admin start-server
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
$ sudo apt-get install libc6:i386 libncurses5:i386
                     libstdc++6:i386
$ sudo apt-get install bison
$ sudo apt-get install libxml2-utils
$ source build/envsetup.sh
$ lunch rpi3-eng
$ make ramdisk systemimage
```

Successful build should create ramdisk.img and system.img files in `out/target/product/rpi3` folder. First time files were missing and turned out that there was an error in setup (environment wasn't set correctly to rpi3-eng and images were built for some other target).

4.1.4 Installing Android on Raspberry Pi 3

Raspberry Pi 3 doesn't have ROM and instead it boots from SD card.[12] To boot Android from SD card on Raspberry Pi it had to be first partioned in a specific way that was decribed in android-rpi guide. Then it remained only to write Android image files and kernel that were built in previous steps.

Partitioning of SD card There should be overall four partitions placed on SD card:

- W95 FAT32(LBA) Bootable partition with 512 MB size with label BOOT
- Linux EXT4 partition with 512 MB with label system
- Linux EXT4 partition with 512 MB with label cache
- Linux EXT4 partition with remaining space with label data

To prepare this partitions was used fdisk and mkfs tools. First it was necessary to delete previous partitions and define new ones with fdisk. Then mkfs was used for a specific filesystem (mkfs.vfat, mkfs.ext4) to format partitions of SD card. Here is a comprehensive list of steps that were made:

- SD card was inserted in the computer using card reader
- 'df' command was used to see in which directory SD card was mounted
- let's assume that it was mounted in /dev/sdb
- fdisk console was used on /dev/sdb with root privileges as 'sudo fdisk /dev/sdb'
- all existing partitions were deleted by entering 'd' until error
- FAT32 partition was added with options 'n','e','+512M','t','b','a'
- two more partitions were added with options 'n','p','+512M'
- last partition was added with options 'n', 'p' and using default values for other options
- table was checked with 'p' and exit with 'w' if table is correct

Resulting table was looking similar to this:

```
Device     Boot   Start      End  Sectors  Size Id Type
/dev/sdb1  *       2048  1050623  1048576  512M  b W95 FAT32
/dev/sdb2       1050624  2099199  1048576  512M 83 Linux
/dev/sdb3       2099200  3147775  1048576  512M 83 Linux
/dev/sdb4       3147776 30449663 27301888   13G 83 Linux
```

After to format created partitions and assign labels to them were use the following commands:

```
$ sudo mkfs.vfat -n BOOT /dev/sdb1
$ sudo mkfs.ext4 -L system /dev/sdb2
$ sudo mkfs.ext4 -L cache /dev/sdb3
$ sudo mkfs.ext4 -L userdata /dev/sdb4
```

Writing Android System to Raspberry Pi To write Android system to SD card it was necessary to write kernel and Android components that were built before.

To write system.img file from 'out/target/product/rpi3' folder following commands were used (assuming that system partition is mounted at /dev/sdb2):

```
$ cd out/target/product/rpi3
$ sudo dd if=system.img of=/dev/sdb2 bs=1M
```

To write another files to Raspberry Pi it was enough to simply copy them to SD card:

```
$ cp device/brcm/rpi3/boot/* /dev/sdb2
$ cp kernel/rpi/arch/arm/boot/zImage /dev/sdb2
$ cp kernel/rpi/arch/arm/boot/dts/bcm2710-rpi-3-b.dtb /dev/sdb2
$ mkdir /dev/sdb2/overlays
$ cp kernel/rpi/arch/arm/boot/dts/overlays/vc4-kms-v3d.dtbo
     /dev/sdb2/overlays
$ cp out/target/product/rpi3/ramdisk.img /dev/sdb2
```

Booting Android Finally it was possible to boot Android. It was enough to insert SD card in Raspberry Pi board, connect monitor using HDMI cable, connect mouse and keyboard. First boot took a bit of time, however, next boots took only around one minute.

Setup ADB connection To setup ADB connection was used option with WiFi network. Device with booted Android was connected to network using Settings application. Then IP address was taken in Status section of Settings and used in ADB tool to connect to shell with the following command:

```
$ adb connect ip-address
```

4.2 Testing of procmem and atrace

All tools that were discussed in Analysis chapter were available and functionable in the shell of target device.

procmem procmem was fully functional and worked without any issues.

atrace Before using atrace it was necessary to mount debugfs filesystem to enable ftrace. Then atrace worked as expected for several kernel categories but for Android categories it didn't output anything.

Atrace was tested on 'eng' build not on 'userdebug' of rpi3. However, atrace worked well on emulator both eng and userdebug.

Grep on aosp for atrace tags lead to file that could be responsible for enabling atrace tracing. It's trace_dev.c located in system/core/libcutils. This module turned out to have LOG_TAG defined as cutils-trace. Search in the logcat output revealed an error with this tag - fail to write in file because it doesn't exist (from source it was revealed that file is trace_marker of ftrace).

As it was mentioned before there was an issue with debugfs to make ftrace working. It seems that system needs to mount it on boot to be able to trace Android components.

init.rc and init.device_name.rc files are the one that specify how to boot the system[13]. Analyzing init.goldfish.rc in device/google/atv folder that is usually the one that is used by vendors[13], it was noticed that first command that is defined is the following:

```
on early-init
    mount debugfs debugfs /sys/kernel/debug
```

This command surely mounts ftrace filesystem and it is missing in init.rpi3.rc. After adding it atrace started to show events of different Android components and issue was resolved.

However, in logcat there was still an error even though now about insufficient permission to write file for example for camera or audio services. Using chmod to enable write permission to trace_marker file of ftrace on early-init didn't resolve the problem.

4.3 Refactoring of atrace

atrace source code is contained in one file and compiled as C++. The style of code is procedural and its pieces of logic are divided into separate functions. A lot of global variables make code not easy to read and understand. Many condition branches that change the flow of the program make the separation of functionality difficult.

Nevertheless, all functionality of this tool is useful for implementation of the thesis task. It includes a minimal interface for setup tracing in ftrace as well as logic to turn on tracing in Android system.

Absense of any tests makes this code legacy[14] and it's hard to introduce new changes. Therefore, refactoring process should be done in several stages to minimize space for addition of new bugs.

The designed plan of refactoring of atrace contains the following stages:

- Encapsulate logic from details of environment and tracing systems
- Add tests using mocks to substitute dependencies of environment
- Refactor core logic

The implementation of these stages is described in detail in the following sections.

4.3.1 Initial encapsulation

It was essential to write tests at the beginning of refactoring process. To make writing of first tests easier all interactions of application with ftrace and Android system as well as console arguments should be extracted and exchanged by interfaces. Once API would be set, it would be much easier to exchange these dependencies by mock or dummy objects and write basic tests.

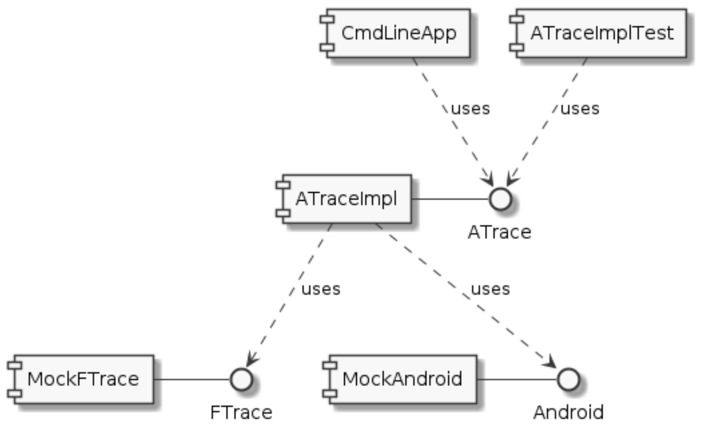


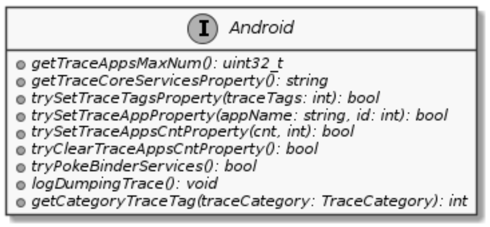
Figure 4.5: ATrace initial component model

4.3.1.1 Encapsulation of Android logic

In a nutshell, to enable tracing atrace writes values in special Android system properties and makes a call through Binder to notify all available services about these changes.

After separating all logic and data related to details of properties handling and notification of services the following interface for Android was designed:

It was important for design to encapsulate as minimal logic as possible. For example, setting of tags properties to enable categories could be encapsulated in Android class to remove from user of class responsibility of handling tags. However, in this stage it's not our goal. Moreover, handling of tags is also logic that could be tested and it would be easier to do if it will based on a designed interface.



```
Android
getTraceAppsMaxNum(): uint32_t
getTraceCoreServicesProperty(): string
trySetTraceTagsProperty(traceTags: int): bool
trySetTraceAppProperty(appName: string, id: int): bool
trySetTraceAppsCntProperty(cnt, int): bool
tryClearTraceAppsCntProperty(): bool
tryPokeBinderServices(): bool
logDumpingTrace(): void
getCategoryTraceTag(traceCategory: TraceCategory): int
```

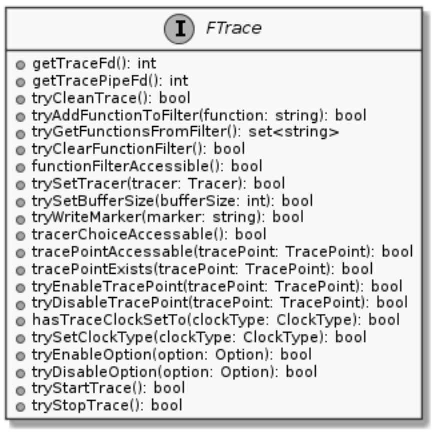
Figure 4.6: Android interface

TraceCategory class was emplemented using enum data structure.

4.3.1.2 Encapsulation of ftrace logic

ftrace provides its interface through filesystem. atrace, therefore, uses a lot of predefined filenames that correspond to ftrace configuration files and writes and reads strings from these files to setup a desirable trace.

Here designed interface does cover some nuances of setting options because it would be too tedious to write tests using only some filesystem abstraction. However, filesystem interface would be introduced in next stages and will simplify testing of ftrace interface.



```
FTrace
getTraceFd(): int
getTracePipeFd(): int
tryCleanTrace(): bool
tryAddFunctionToFilter(function: string): bool
tryGetFunctionsFromFilter(): set<string>
tryClearFunctionFilter(): bool
functionFilterAccessible(): bool
trySetTracer(tracer: Tracer): bool
trySetBufferSize(bufferSize: int): bool
tryWriteMarker(marker: string): bool
tracerChoiceAccessable(): bool
tracePointAccessable(tracePoint: TracePoint): bool
tracePointExists(tracePoint: TracePoint): bool
tryEnableTracePoint(tracePoint: TracePoint): bool
tryDisableTracePoint(tracePoint: TracePoint): bool
hasTraceClockSetTo(clockType: ClockType): bool
trySetClockType(clockType: ClockType): bool
tryEnableOption(option: Option): bool
tryDisableOption(option: Option): bool
tryStartTrace(): bool
tryStopTrace(): bool
```

Figure 4.7: FTrace interface

These interface uses enums in the same manner as Android interface to represent ftrace options, tracepoints, tracers and clocks.

4.3.1.3 Abstraction from console interface

To abstract from console interface it was necessary to encapsulate all atrace core functionality by wrapping it into a class called ATrace with setter methods for applying commandline arguments, run method to execute functionality and a way to inject dependencies on Android and FTrace interface that were designed in previous sections.

To make testing easier some further classes and interfaces were introduced in this stage to wrap calls from application environment. atrace code was registering signal handler to change a boolean variable and control its behaviour (break from infinite loop of tracing, for example). This boolean was wrapped into class Signal and passed to atrace wrapper class. Moreover, to abstract also from arguments parsing were introduced ATraceArgs class that contains all necessary arguments to run atrace.

Different approaches were tried to implement dependency injection that is needed to efficiently test core functionality and also to encapsulate logic of constructing ATrace instance from ATraceArgs values. At the end a version of Builder pattern was used. It was implemented by introduction of new classes that contained logic of injecting dependencies and applying command line arguments.

After applied changes the enter point of application was looking like this:

```
int main(int argc, const char ** argv) {
  cmdLineApp.setArgs(new CmdLineArgs(argc, argv));
  cmdLineApp.setActionCmdLineBuilder(
    new ATraceCmdLineBuilder(
      new ATraceArgsCmdLineBuilder(),
      new ATraceBuilder(
        new FTraceBuilder(),
        new AndroidBuilder()
      )
    )
  );
  registerSignalHandler();
  return cmdLineApp.run();
}
```

CmdLineApp instance uses args to build new Action instance that is implemented by ATrace. ATraceCmdLineBuilder builds ATrace using ATraceArgsCmdLineBuilder that builds ATraceArgs from CmdLineArgs. ATraceArgs are used by ATraceBuilder to build ATrace and provide it with two dependencies built by FTraceBuilder, AndroidBuilder.

4.3.2 Testing core

Testing of core logic was implemented using Google Mock objects. Once the logic was wrapped into class with two dependencies on FTrace and Android interfaces it was easy to make mocks for FTrace and Android to inject them in the tested class object.

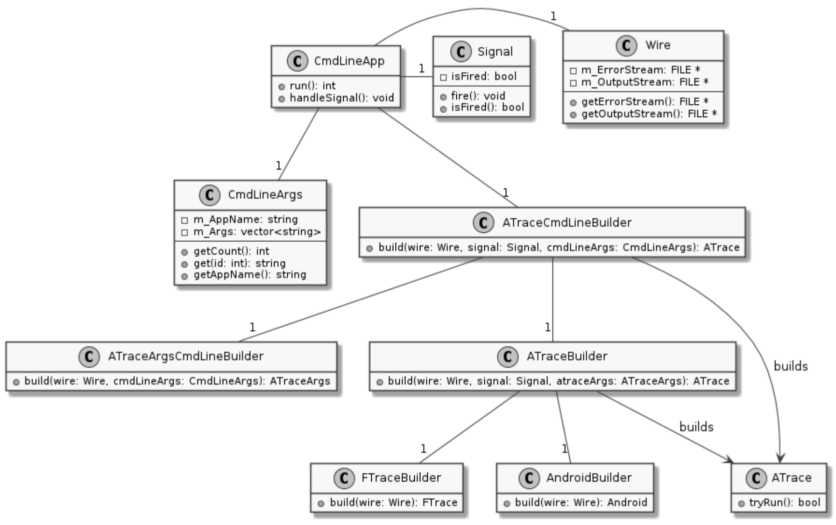


Figure 4.8: CmdLineApp class diagram

For example, MockFTrace was injected into tested ATrace class object and then by calling methods of this object, it was tested which methods of MockFTrace were called and with which arguments.

```
class ATraceTest : public ::testing::Test {
  void testRun() {
    myMockFTrace = new MockFTrace();
    myMockAndroid = new MockAndroid();
    myATrace = new ATrace(myMockFTrace(),
                          myMockAndroid());
    myATrace->addCategory("sched");
    myATrace->setTime(0);
    EXPECT_CALL(*myMockFTrace,
                 tryEnableTracePoint(FTracepoint::SCHED_SWITCH))
                .WillOnce(Return(true));
    EXPECT_CALL(*myMockAndroid, trySetTraceTagsProperty(0))
                .WillOnce(Return(true));
    EXPECT_TRUE(myATrace.tryRun());
  }
}
```

Figure 4.9: Example of ATraceTest with Google Mock framework

4.3.3 Refactoring of core logic

It was necessary to continue refactor the inner logic of class to make application easy to extend, in particular, for inclusion of missing process memory profiling logic and processing trace buffer dump.

Concept The basic structure of original atrace application is built by a sequence of calls to procedures that setup trace systems, both Android and ftrace, then start or stop tracing, dump trace buffer and clean up environment to defaults. Main method of application was in charge of which ones of these action procedures to envoke and in which order using complicated checks for different user flags.

Based on that observation it was proposed to separate actions into different classes and extract logic that builts a sequence from them.

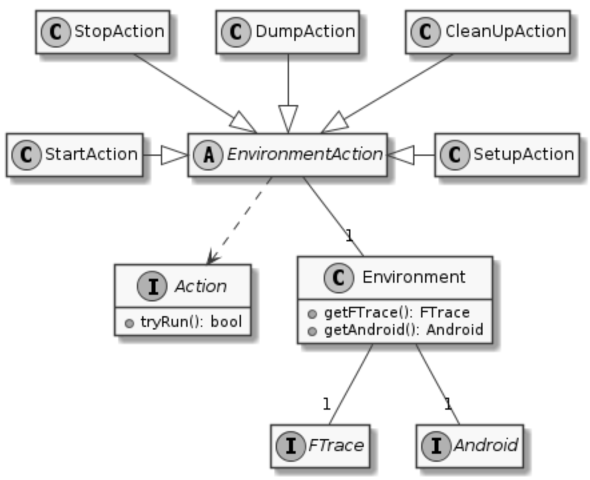


Figure 4.10: Action class diagram

Each action runs in so called Environment or set of interfaces to different application components or Android and FTrace implementations. This approach makes application easy to extend by just including new service in environment or adding new action that uses this service from environment. This way was inserted procmem functionality that is described more in "Merging of procmem into atrace".

Environment is injected in each action and makes actions easy to test using the same approach as before with mock objects.

Component Model As a result of refactoring process a three layered architecture was formed (see Figure 4.11) First layer, presentation layer, interacts with user providing to him console interface through arguments to executable. It also parses user arguments and uses them create new action environment from second layer. This environment is then injected in each action and a sequence of actions is built. Once everything is ready actions are instructed to run.

Second layer includes application components that are used to implement environment interfaces. It constructs an environment for required actions using prepared by first layer arguments. This layer interacts with last layer that was separated at first step of refactoring: FTrace and Android. Another interface was added to this layer to abstract file system operations called FileSystem.

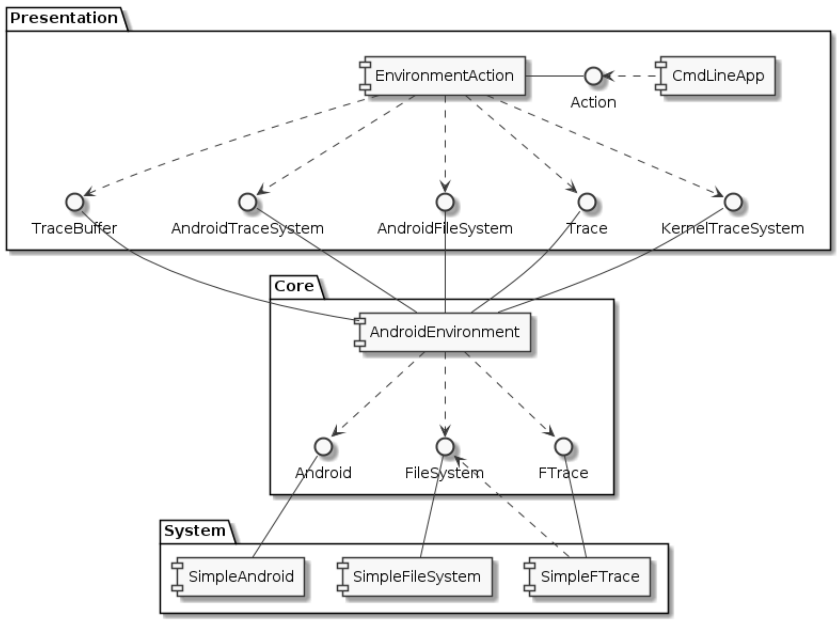


Figure 4.11: Component model of atrace

Issues It was a challenge to determine what kind of actions to extract and how separate behaviour between them. Same challenge was with separation of environment components. A big part of implementation process was spent on understanding the original code that mostly was written in C style without separation of responsibilities.

Console arguments interface A significant effort was made to provide an easy to use classes for definition of application console arguments. As a result two classes were created that are called CmdLineArgsParser and Arguments.

The caller uses CmdLineArgsParser to register a specific type of argument with option and id. Then it calls parseTo method with provided command line arguments and Arguments object. While parsing, CmdLineArgsParser uses registered types to correctly parse and convert arguments. Then it's put them to Arguments object by the same ids that were provided in registration step. After parsing is done, caller can access arguments using its ids.

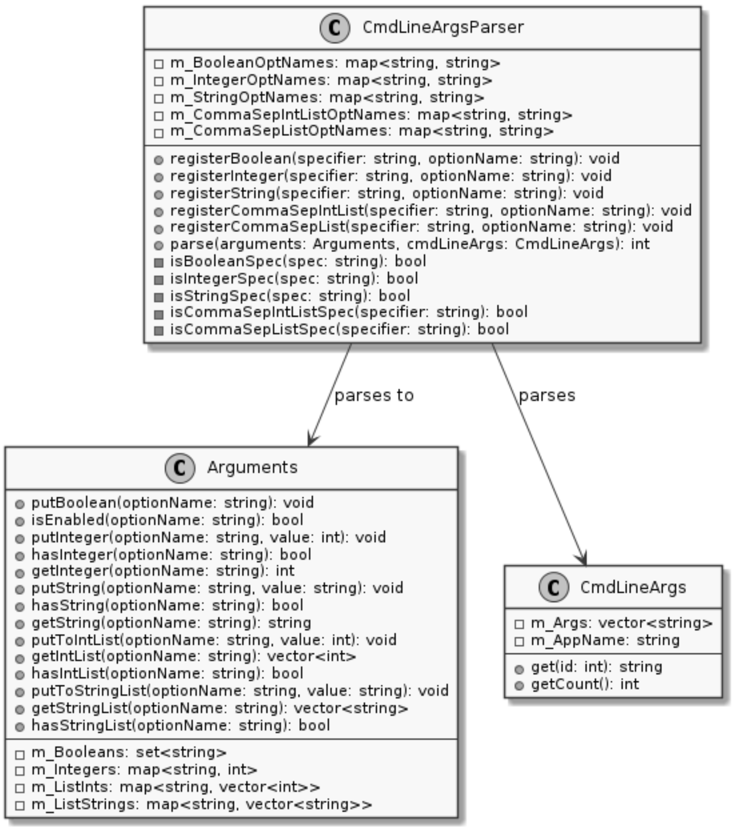

CmdLineArgsParser
m_BooleanOptNames: map<string, string>
m_IntegerOptNames: map<string, string>
m_StringOptNames: map<string, string>
m_CommaSepIntListOptNames: map<string, string>
m_CommaSepListOptNames: map<string, string>
registerBoolean(specifier: string, optionName: string): void
registerInteger(specifier: string, optionName: string): void
registerString(specifier: string, optionName: string): void
registerCommaSepIntList(specifier: string, optionName: string): void
registerCommaSepList(specifier: string, optionName: string): void
parse(arguments: Arguments, cmdLineArgs: CmdLineArgs): int
isBooleanSpec(spec: string): bool
isIntegerSpec(spec: string): bool
isStringSpec(spec: string): bool
isCommaSepIntListSpec(specifier: string): bool
isCommaSepListSpec(specifier: string): bool
parses to
parses
Arguments
putBoolean(optionName: string): void
isEnabled(optionName: string): bool
putInteger(optionName: string, value: int): void
hasInteger(optionName: string): bool
getInteger(optionName: string): int
putString(optionName: string, value: string): void
hasString(optionName: string): bool
getString(optionName: string): string
putToIntList(optionName: string, value: int): void
getIntList(optionName: string): vector<int>
hasIntList(optionName: string): bool
putToStringList(optionName: string, value: string): void
getStringList(optionName: string): vector<string>
hasStringList(optionName: string): bool
m_Booleans: set<string>
m_Integers: map<string, int>
m_ListInts: map<string, vector<int>>
m_ListStrings: map<string, vector<string>>
CmdLineArgs
m_Args: vector<string>
m_AppName: string
get(id: int): string
getCount(): int

Figure 4.12: Arguments, CmdLineArgsParser and CmdLineArgs classes

This design makes it easy to extend application with new user arguments. It is just enough to register necessary option and obtain from Arguments object after.

4.4 Refactoring of procmem

procmem uses pagemap library that provides interface using structures and functions that run on this structures. For the purpose of this thesis functionality that involves collecting memory pages was not needed. It was enough to be able to get RSS, VSS, PSS or USS values for a concrete process.

procmem provides statistics only for one specified process. It is necessary to extend it to make statistics on a list of processes.

4.4.1 Example of usage of pagemap

An example of usage of library could be found on figure 4.13. This code retrieves memory usage of a process with PID 2018.

```
typedef struct process_usage {
  pm_process_t * process_ptr;
  pm_memusage_t memusage;
} process_usage_t;

pm_kernel_t * kernel_ptr;
process_usage_t process_usage;
pid_t pid = 2018;

if (!pm_kernel_create(&kernel_ptr)
    || !pm_process_create(kernel_ptr, pid,
                           &(process_usage.process_ptr))
    || !pm_process_usage(process_usage.process_ptr,
                          &(process_usage.memusage))
{
  exit(EXIT_FAILURE);
}
printf("rss: %ld, vss: %ld", process_usage.memusage.rss,
                                process_usage.memusage.vss);
exit(EXIT_SUCCESS);
```

Figure 4.13: Pagemap library usage example

Note One interesting issue with functions that retrieve memusage structure is a necessity to place 'pm_memusage' structure next to pointer of structure for which memory usage is requested. This behaviour was hard to realize and there are no comments about this in function documentation. That's why in code example it was necessary to put 'pm_process_t' pointer with 'pm_memusage_t' at the same structure.

4.4.2 Implementation issues

Nested classes To encapsulate structures with functions from pagemap library into classes it was necessary to use nested classes friend feature of C++. The reason is the hierarchy of structures that makes one structure need another one to initialize itself. This requires to uncover library structures of

”parent” class. With nested classes it can be avoided because inner class can access private members of outer.

Static constructor methods Classes have to contain inside structures provided by library. However, the problem is that this structures has to be initialized and there exists an option that it will produce an error. Therefore, regular constructors are not suitable because it can lead to initialization of invalid instance. On other hand, static creator methods are a good solution with ability to return null pointer in case of errors.

It is also necessary to put structure inialization code in private method of class and call it in static creator after construction. The problem is that library function needs a pointer to memory structure for initialization.

4.4.3 Final implementation

To provide extra options for extension, API of PM_Kernel was modeled using interfaces and dependency injection. For caller it's enough to use Kernel-Builder interface to create object with Kernel interface and use it to create Process. After from Process it's possible to create two different MemoryUsage implementation - one that returns only working set of process and another that returns all memory. MemoryUsage interface provides 'tryUpdate' method to update information for current time.

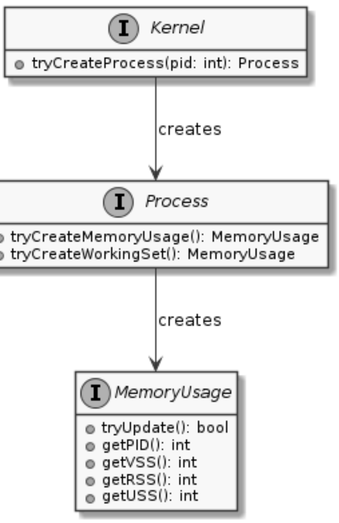


Figure 4.14: Procmem interface

4.5 Merging of procmem into atrace

One feature that atrace doesn't have and that is essential for purpose of this thesis is logging of memory statistics for a process to trace buffer.

Design The idea of gathering and writing in trace buffer memory statistics for a process was to substitute sleeping of process in the original application to writing of memory statistics.

In refactored atrace application it means to substitute SleepAction in a sequence of actions to new action with desired functionality. In its tryRun method it will obtain pagemap API, get statistics for a certain PID and log them to trace buffer.

TraceBuffer already can be accessed through Environment, however, it is necessary to add to environment new getter to get Kernel interface that is a creator for Process with certain PID. Using this Process interface it is possible to get MemoryUsage and obtain statistics.

Implementation For usability of new feature it was necessary to provide new user options. They are period, number of periods and list of PIDs to take memory measurements from. Period is specified in nanoseconds and controls at what intervals application should make measurements of process memory. Number of periods says how many periods should trace run overall. List of PIDs specifies PIDs to take memory measurements from.

New added action was called MemorySampleAction. It uses Kernel getter from Environment to acquire MemoryUsage instances for each of specified PIDs. Once started action logs memory usage contents, goes to sleep for a specified period, wakes up, logs memory usage and again goes to sleep. It repeats this cycle number of times equal to number of periods that was specified by user.

4.6 Testing

Unit tests using mocks were written for main core components of applications. FTrace tests use FileSystem mock and AndroidTraceSystem uses Android interface mock. Main actions such as StopAction, StartAction were also tested.

4.7 Development of trace processing layer

Design To collect results from trace it was necessary to parse ftrace file. Tool ftrace creates entries in the format that was briefly specified earlier. Here is another example:

Buffer contains one entry for each kernel event or write event (tracing mark write) to buffer. Each entry has common fields in the beginning and specific

```
android.display-1332  ( 1311) [000] d..3   287.760915:
 sched_switch: prev_comm=android.display
 prev_pid=1332 prev_prio=116 prev_state=S
 ==> next_comm=Binder:1036_2 next_pid=1083 next_prio=120
  Binder:1036_2-1083  ( 1036) [000] dN.5   287.760923:
 sched_wakeup: comm=android.display
 pid=1332 prio=116 success=1 target_cpu=000
android.display-1332  ( 1311) [000] ...1   287.760932:
 tracing_mark_write: E
android.display-1332  ( 1311) [000] ...1   287.760944:
 tracing_mark_write:
 B|1311|com.android.server.wm.WindowManagerService$H: #8
android.display-1332  ( 1311) [000] ...1   287.760958:
 tracing_mark_write: E
```

Figure 4.15: ftrace buffer dump format

content in last column with preceding identifier. The content is usually a pair of values and their names.

The parsing functionality should be easy to extend in a purpose of adding new events. New statistics of processes could be written, plenty of ftrace events with their own format could be supported in future application versions.

Parsing should output a list of events for each process with PID from a list specified by user. Each entry in this list should contain CPU usage, memory usage statistics, event name and timestamp.

Implementation Several abstractions were used on the way from converting a line of trace buffer to process record.

First of all application tries to create an FTraceEntry instance from each line of file. FTraceEntry is a base class and each separate event defines it's own class that inherits common fields from FTraceEntry. Moreover, for creation of each FTraceEntry child class there are FTraceEntryCreator child classes that contain logic to parse content column value specific for it's entry. These creators are then used in Factory Pattern when during parsing a necessary creator is chosen by identifier column (see Figure 4.16).

FTraceEntry besides representing an information from line also contains logic for parsing its information in ProcessChange instances. These process changes are then applied to Process instances that represent states of system processes. For each change a ProcessRecord is created. Its CPU usage is calculated by dividing process total run on total time of tracing at moment of recorded change. Other fields like memory usage are just copied from last Process instance state (see Figure 4.17).

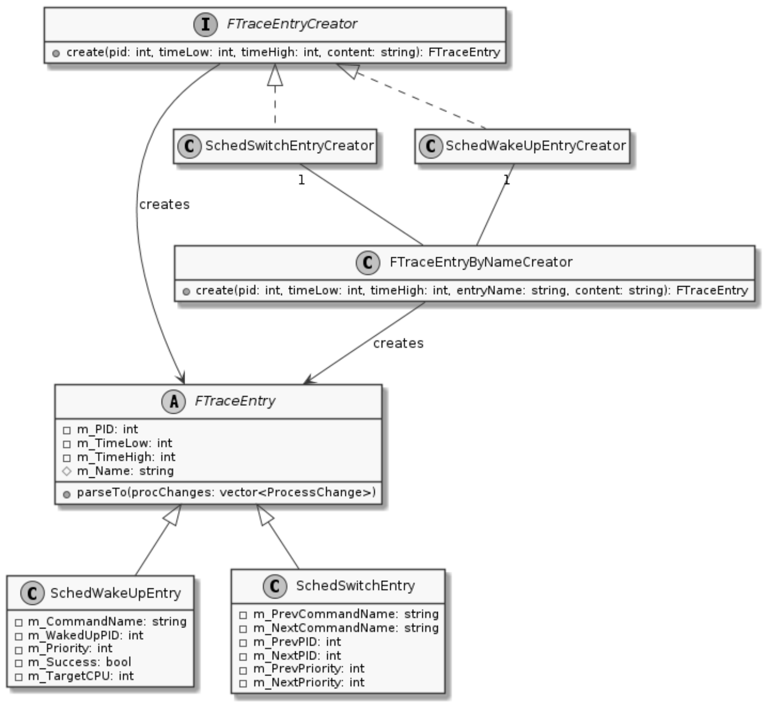


Figure 4.16: FTraceEntry creators class diagram

Calculation of problematic sequences of events Calculation of problematic sequences is based on determining events that pass a user specified threschold of CPU usage or memory value.

A new action was added to application called InterpretDumpFileAction. It uses ProcessRecordFile interface to obtain all records from the trace buffer. Then these records are referenced to user defined processes and output to user with filtered records that passed threschold.

4.8 Diagnostic tool

Features A result of solution section is a diagnostic tool that supports the following features:

- enable tracing of Android
- enable kernel ftrace tracepoints
- profile given set of pids and log them in common trace buffer
- process file with dump of trace buffer and create process event sequences
- based on event sequences and user threshold output problematic sequences to user

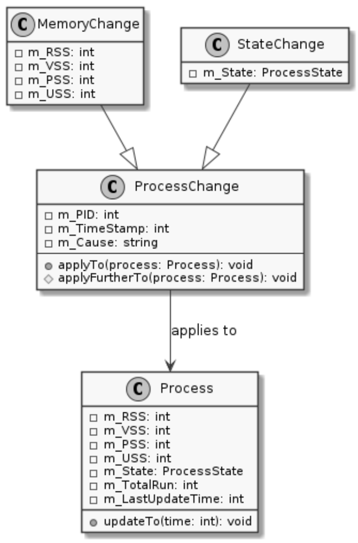


Figure 4.17: Process and Process Change class diagram

Features and usage of atrace was preserved with minor changes in console arguments, especially, android trace categories and kernel trace categories should be explicitly divided by using separate options in contrast with previous interface.

There were added options how to specify memory measurements, especially period, number of periods to run trace and list of pids to take memory measurements of. All these parameters are necessary for a memory tracing to start.

Moreover, it's possible to filter records using two options. One option defines the minimal border of CPU usage in percents that should be overrun for a record to display. For example, by specifying a value 20, only records with value of 21 percent and more will be displayed.

Another option specified how much from initial record value of RSS should other records deviate to be shown to user. For example, by saying 1000 it would mean for application to filter all records that didn't deviate from first records by more than 1000 KB.

Implementation Application consists of two parts: collection of tracing information and processing of collected information. Second part strongly depends on first, in particular, on what kind of information was collected.

To get calculations of CPU usage it is necessary to specify for first stage to collect information about scheduling events. CPU usage is calculated based on process state switches and without this information nothing will be displayed to user.

Similar applies to memory usage of process. It's crucial to specify period for memory collection and how many periods should be made. This would create required information in buffer for further processing.

And at last for processing it is necessary to specify file destination to dump buffer in. Then processing will be able to take place after using dumped data.

Therefore to get full table of statistics for the process it is necessary to provide the following options:

```
$ atrace -e sched -p 100000 -m 1000 -o /data/trace -pids PID
```

where 'e' specifies kernel categories, here, in particular sched group of categories. 'p' option specifies period of 100000 and 1000 of repetitions.

Format of memory tracing To provide statistics about process memory application make records in the following format to ftrace buffer:

```
M: VSS=UINT64 RSS=UINT64 PSS=UINT64 USS=UINT64 PID=INT
```

Format begins with letter M to differentiate from Android system entries. Then are printed memory statistic values and PID of traced process.

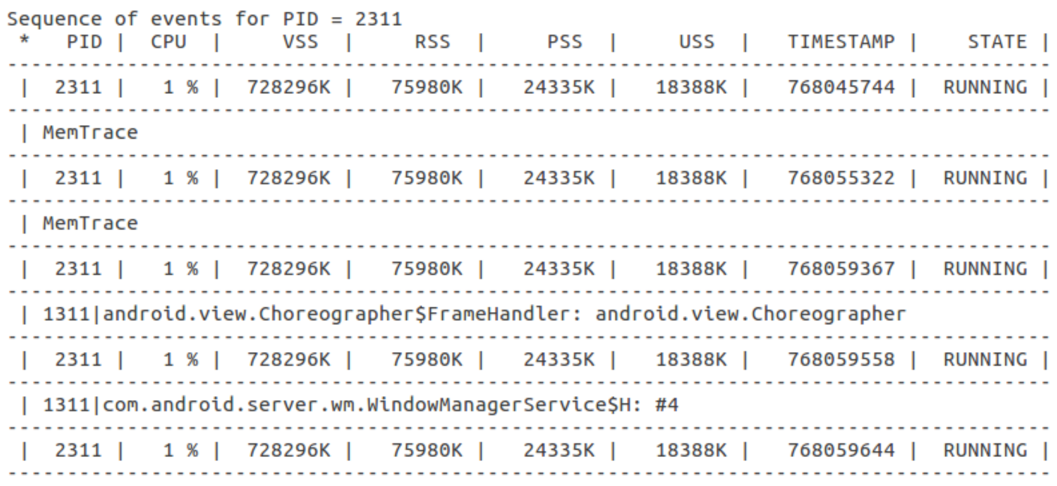
Example of usage This example will describe how to find out whether application android.settings allocates for its process not less than 100 KB during interaction with user and not less than 1% of CPU time.

First of all it's necessary to find out PID of android.settings. It's easy to do using, for example, 'top' command.

Assuming that PID of android.settings is 2311 the following command will be able to answer our above question with also appending some android events in the sequence of measurements:

```
adb shell /data/extrace -d am -e sched -p 100000 -m 1000
                        -minCpu 1 -minUss 100
                        -pids 2311 -o /data/trace
```

The output will look as it's shown on figure 4.18. Filtered records are displayed in two lines. First line contains statistics such as (from left to right) PID of the process, CPU usage percentage and memory values for VSS, PSS, USS and RSS ending with timestamp and process state. Second line contains content or name of the event. Android uses content to display package of service and its event description. For memory measurement records made by application used 'MemTrace' identifier. For kernel events such as process switches are used SchedSwitch or SchedWakeup identifiers.



```
Sequence of events for PID = 2311
 *   PID |  CPU  |      VSS  |      RSS  |      PSS  |      USS  |   TIMESTAMP |    STATE |
-------------------------------------------------------------------------------------------
 |  2311 |   1 % |  728296K |   75980K |   24335K |   18388K |   768045744 |  RUNNING |
-------------------------------------------------------------------------------------------
 | MemTrace
-------------------------------------------------------------------------------------------
 |  2311 |   1 % |  728296K |   75980K |   24335K |   18388K |   768055322 |  RUNNING |
-------------------------------------------------------------------------------------------
 | MemTrace
-------------------------------------------------------------------------------------------
 |  2311 |   1 % |  728296K |   75980K |   24335K |   18388K |   768059367 |  RUNNING |
-------------------------------------------------------------------------------------------
 | 1311|android.view.Choreographer$FrameHandler: android.view.Choreographer
-------------------------------------------------------------------------------------------
 |  2311 |   1 % |  728296K |   75980K |   24335K |   18388K |   768059558 |  RUNNING |
-------------------------------------------------------------------------------------------
 | 1311|com.android.server.wm.WindowManagerService$H: #4
-------------------------------------------------------------------------------------------
 |  2311 |   1 % |  728296K |   75980K |   24335K |   18388K |   768059644 |  RUNNING |
-------------------------------------------------------------------------------------------
```

Figure 4.18: Example of output after tool execution

Chapter 5

Conclusion

The goal of this thesis was to develop a diagnostic tool for components of Android system running on Raspberry Pi platform. Application should be able to compute memory and cpu consumption of system processes related to sequence of events happening with them. Based on these statistics tool should be able to mark a possibly problematic sequence of events and display it to the user.

As a result of this project was developed a tool that base its functionality on Android tracing system called atrace and Android library pagemap. Atrace turned out to be a very suitable system for the goals of this project because it uses ftrace to trace kernel events and also instructs Android components to log their behaviour in the same trace buffer. That gives an option to see for a specific period of time what was happening in Android system and kernel at the same time. Pagemap provides an easy to use C library to obtain memory usage statictics for a process.

One of the major conclusions of this project was that tracing memory and performance is not so common and usually these tasks are done separately. However, during research it was found that atrace interface application doesn't do a lot of work except that sending signals to system and kernel to start tracing or stop. During tracing the application just put itself to sleep. This spot turned out to be a good place to extend atrace and instruct process instead of sleeping to collect memory usage statistics and log them into the same ftrace buffer. That produced a trace output that contained events from kernel, Android components and memory usage statistics with timestamps. It wasn't so hard to parse this information after and using some rules like limit barriers for memory or CPU time consumption display problematic sequences to the user. Tool supports such profiling for multiple processes at the same time that could be specified by user.

Tool was designed in OOP style with documentation and tests that made it easy to extend.

In future tool could be extended to gather more information about memory, specify more events to trace in kernel. These extensions could increase the amount of data to analyse. Analysis also can be improved by introduction of more rules, such as for example, relation of memory allocation in kernel space with allocations in user space.

Provided by tool information about memory consumption in different points of time could be also used to improve Trace-Viewer interface that uses original atrace output to show to the user a visual interactive graph to analyse but lacks memory usage statistics.

Bibliography

[1] Google. Android Things - Get Started [online]. April 2018, [cit. 2018-05-14]. Available from: https://developer.android.com/things/get-started/

[2] Wiki, E. L. Android Memory Usage [online]. [cit. 2018-05-14]. Available from: https://elinux.org/Android_Memory_Usage

[3] Google. Systrace Documentation [online]. April 2018, [cit. 2018-05-14]. Available from: https://developer.android.com/studio/command-line/systrace

[4] Google. Instrument your app code [online]. April 2018, [cit. 2018-05-14]. Available from: https://developer.android.com/studio/command-line/systrace

[5] Evans, J. ftrace: trace your kernel functions! [online]. March 2017, [cit. 2018-05-14]. Available from: https://jvns.ca/blog/2017/03/19/getting-started-with-ftrace/

[6] Rostedt, S. ftrace - Function Tracer [online]. 2017, [cit. 2018-05-14]. Available from: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[7] Google. Malloc Debug [online]. 2018, [cit. 2018-05-14]. Available from: https://android.googlesource.com/platform/bionic/+/master/libc/malloc_debug/README.md

[8] Google. Simpleperf Documentation [online]. April 2018, [cit. 2018-05-14]. Available from: https://developer.android.com/ndk/guides/simpleperf

[9] man pages, L. Linux Programmer's Manual - PERF EVENT OPEN [online]. February 2018, [cit. 2018-05-14]. Available from: http://man7.org/linux/man-pages/man2/perf_event_open.2.html

[10] Google. Requirements [online]. 2018, [cit. 2018-05-14]. Available from: `https://source.android.com/setup/build/requirements`

[11] Yaghmour, K. *Embedded Android: Porting, Extending, and Customizing.* O'REILLY, 2013.

[12] Molloy, D. D. *Exploring Raspberry Pi.* John Wiley and Sons, Inc., 2016.

[13] Levin, J. *Android Internals::A Confectioner's Cookbook*, volume 1. Technologeeks.com, 2015.

[14] Feathers, M. C. *Working Effectively with Legacy Code.* Prentice Hall PTR, 2005, 15 pp., legacy code definition.

APPENDIX A

Acronyms

OS	Operation system
IoT	Internet of Things
ADB	Android Debug Bridge
AOSP	Android Open Source Project
VSS	Virtual Set Size
RSS	Resident Set Size
PSS	Proportional Set Size
USS	Unique Set Size
CPU	Central Processing Unit
PMU	Processor Monitoring Unit
OOP	Object Oriented Programming

Appendix B

Contents of enclosed USB disk

```
readme.txt ............ the file with USB disk contents description
exe ............................ the directory with executables
src ........................ the directory of source codes
  app ............................... implementation sources
  thesis ......... the directory of LaTeX source codes of the thesis
text ............................ the thesis text directory
  thesis.pdf ..................... the thesis text in PDF format
  thesis.ps ....................... the thesis text in PS format
```