Android Memory Inspection Techniques and Tools

Kirill Krinkin, Valeriya Dopira, Olga Kochneva, Sergey Petrov, Maxim Kopylov
Saint-Petersburg Electrotechnical University "LETI"
Saint-Petersburg, Russia
kirill@krinkin.com, leradopira@mail.ru, kochnevaolga74@bk.ru,{serpetr99, maksim.kopylov}@gmail.com

***Abstract*—The paper overviews Android and Linux memory management techniques and compares available tools for memory examination, monitoring and profiling. In the paper, a new tool, *apagescan* for virtual memory page tracking is introduced. A virtual page distribution and fragmentation analysis are discussed as a novel approach to performance tuning for devices with limited resources.**

I. Introduction

Mobile application development and performance optimization are challenging processes. The market is evolving, so developers have to inject new technologies into their software. They update existing tools and even create new languages. Herewith to create a hit application, software engineers must gain detailed verification of mined systems. Diagnostics, analyzing and debugging are being used for optimal resource provisioning for the applications.

Memory management is part of Linux kernel functionality. All applications require resources, therefore for the developer it is important to understand how the resources are being distributed between processes and applications. Understanding memory re-allocation dynamics can give much more clear picture on system load.

The paper organized as follows. In the first section general information about Android memory management is given. The second section discusses available Android and Linux memory management tools. The third part introduces page-level memory inspection approach and *vpagscan*. The last two sections contain evaluation and conclusion.

II. Memory management in Android

In this section general Android memory management and inspection interfaces and techniques are described.

A. Android memory organization

The basis of the Android platform is the Linux kernel. Hence, Android Runtime (ART) relies on the Linux kernel for underlying functionalities such as threading and low-level memory management. The kernel has full access to the system's memory and allows processes to safely access this memory as they require it. A common approach for this is virtual addressing, usually achieved by paging and/or segmentation [1]. Memory management in Android OS is organized using pages. Different architectures may have different page sizes, typically 4 KB. The kernel monitors the state of each page of physical memory presented in system.

Each page could be in one of the following states:

1) *Present / not Present:* Present page is mapped to a page in physical memory. If page is not present, it may be swapped to disk [2].
2) *Dirty / Clean:* Dirty pages for the given process are pages that were modified since they were paged into physical memory from disk memory and are waiting to get written back to the disk. They can only be restored from backing storage. Clean pages are pages that weren't modified by the process [2].
3) *Named / Anonymous:* Named pages mapping is backed by the file. Named clean pages may be restored from this file. Anonymous mapping is backed either by the swap space or by physical memory. Anonymous clean pages may be restored from /dev/zero [2].

B. Pagemap interface

The Linux kernel provides information about page states via pagemap interface. It allows user-space programs to examine page tables and page-related information. Interface includes a set of files available via proc file system (procFS). There is a set of files presented below associated with each process in the system:

- */proc/pid/pagemap:* this file contains information about each virtual page of the process, and which physical frame it is mapped to. Each page is represented as unsigned 64-bit number, containing page frame number (PFN) and a set of flags that provide information about page parameters (present/swapped, dirty/clean etc.) [3].
- */proc/kpageflags:* this file contains a 64-bit set of flags for each page, indexed by PFN, which provide additional information [3].
- */proc/pid/maps:* this file contains information about currently mapped memory regions of the process. This information can be used to determine which areas of memory is actually mapped and skip over unmapped regions.

In order to collect information about the whole system memory map it is required to analyze all these files.

ISSN 2305-7254

C. Control groups (cgroups)

Control groups (cgroups) [4] is a kernel feature which allows monitoring, managing and limiting processes resources. Processes are organized into hierarchical groups, each cgroup is represented by a directory in the cgroup file system containing the following files describing that cgroup [5]:

- *tasks:* list of tasks (by PID) attached to that cgroup. Writing a PID into this file moves the process into this cgroup.
- *cgroup.procs:* list of thread group IDs in the cgroup. Writing a thread group ID into this file moves all threads in that group into this cgroup.

III. Memory inspection techniques

Memory inspection techniques can be divided into two main parts - memory debugging for a particular application and overall view of memory state. In this paper, we will focus on the second part. There are several important memory inspection aspects that are useful for the analysis of memory state. A brief description of each of them is given below:

- *Memory usage*. Information about memory distribution.
- *Memory fragmentation*. Page by page statistics showing how distributed the process pages across the physical memory.
- *Real-time executing and dynamics.* Collecting, processing, and providing output data on the fly, with the ability to collect statistics for a specific period of time (to track RAM changes during the execution of a particular large process, for example).
- *Selection of processes.* Collecting statistics for specified processes.
- *Control groups support.* Displaying whether a process belongs to a particular cgroup or collects statistics about processes from a specific cgroup.
- *Scanning zRAM.* Collecting memory info not only for RAM but also for zRAM.
- *Convenient output*. Representation of collected data in convenient, human-readable format.

The following tools have been studied according to memory inspection techniques described above.

1) Android Monitor at Android Studio
2) Eclipse Memory Analyzer Tool
3) Linux ‘/proc/meminfo‘ file
4) Linux ‘free‘ command
5) Linux ‘vmstat‘ command
6) Linux ‘top‘ command

A. Android Monitor

Android Studio memory allocation tracking has data visualization tools to help user identify processes that are allocating memory the most. Tool shows a real-time graph of application’s memory usage and allows capture a heap dump, force garbage collections and track memory allocations. To perform visualization, tool needs to collect data at first.

The memory usage data presented in sunburst chart (by default) or layout chart. The main difference in these charts is the format of view objects allocation sequence. The chart lets to visualize all the other functions a specific function calls and the number of objects allocated.

Tool analyses code from testing application (process) and identifies a reference that might lead to leaking memory. The application can be from the current project or from Google Play Market or, if the device with root privileges, it can be any running app on device.

B. Memory Analyzer Tool

The Eclipse Memory Analyzer [6] is a real-time Java heap analyzer whose main purpose is to find memory leaks and reduce memory consumption for particular application. Tool is used for analyzing heap dumps with hundreds of millions of objects, quickly calculating sizes of these objects, working of a garbage collector, automatically extracting leak suspects. Memory Analyzer tool is based on Eclipse RCP. User should not install a full-fledged IDE on the operating system for the analysis.

Using Memory Analyzer allows inspection of memory contents at a specific address. Tool is comprised of two parts: the first one contains a list of expressions, variables, and registers that user selects for monitoring, and second is used for choosing display data format.

Following steps allow detecting memory issues: an overview of the heap dump, finding big memory chunks, inspecting the content of this memory chunk. They are automated in Memory Analyzer by the Leak Suspects Report [9].

C. Linux ’/proc/meminfo’ file

This file reports statistics about memory usage on the system, can be used to report the amount of free and used memory (both physical and swap) on the system as well as the shared memory and buffers used by the kernel [10].

Contains information about actual memory state in the operating system. Fields that are valuable for memory state analysis described in the following list:

1) MemTotal - a total usable RAM;
2) MemFree - the sum of LowFree+HighFree;
3) MemAvailable - an estimate of how much memory is available for starting new applications, without swapping;
4) Cached - page cache,
5) SwapCached - memory that once was swapped out and swapped back in but still also is in the swap file;

6) HighTotal - Total amount of highmem. Highmem is all memory above 860MB of physical memory. Highmem areas are for use by user-space programs, or for the page cache;
7) LowTotal - Total amount of lowmem. Lowmem is memory which can be used for everything that highmem can be used for, but it is also available for the kernel's use for its own data structures;
8) HighFree - amount of free highmem;
9) LowFree - amount of free lowmem;
10) SwapTotal - total amount of swap space available,
11) SwapFree - amount of swap space that is currently unused;
12) Dirty pages - amount of dirty pages;
13) AnonPages - amount of anon pages;

D. Linux 'free' command

'free' is a Linux console command. It displays the total amount of free and used memory, buffers using by kernel in the operating system. By default, it displays in MB (megabytes). Memory could be physical and swap. It is displayed into 2 strings. Tool also could show statistics of low and high memory if user uses -l option.

Describing in terms of memory inspection aspects, 'free' provides overall memory usage, with no information about how this memory is distributed between processes, doesnt provide any info about memory fragmentation, and doesnt support the selection of processes. Tool supports swap usage inspection and provides simple console output.

Tool reads information about memory from /proc/meminfo file. The output of this command is not in real-time. Tool gets an instant snapshot of the free and used memory at that moment.

E. Linux 'vmstat' command

'vmstat' is a Linux console command. Tool provides reporting virtual memory statistics covering memory, swap and processor utilization in real-time. Tool displays information about swapped, free, buff and cache memory. This information is the same with free command, therefore has same advantages and disadvantages. Tool allows showing active and inactive memory.

Tool reads information about memory from /proc/meminfo, /proc/stat, /proc/*/stat files.

F. Linux 'top' command

The 'top' program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes currently being managed by the Linux kernel [11]. To analyze the processes, command 'top' has to be executed from the terminal, and then parameters of interest must be selected.

With 'top' user can get full information about memory state:

- memory distribution - for each task get information about tasks currently resident share of available physical memory, including physical memory and swap;
- supports displaying the name of the control group to which a process belongs;
- supports the selection of processes by providing needed pids;
- provides dynamic real-time statistics, with control over update time.

The 'top' command does not show the paging organization of application's memory, therefore 'top' doesnt provide information about memory fragmentation. Provides console table-like output, which takes some time to analyze for needed information.

IV. Page-level memory inspection

In this section, page-level memory inspection techniques are described. Also, the novel tool *apagescan* is introduced.

In section III Linux pagemap interface has been described. In spite of powerful ability for getting access to each process page status, it is quite tricky to get the whole memory usage map. Especially, it is hard to figure out how physical memory pages are re-distributing between group of processes in dynamic. In general, building page map for one particular process could be presented in the following steps.

1) Reading /proc/pid/maps in order to determine which parts of the memory are mapped to the virtual space.
2) Collect required mapped memory regions.
3) Seeking offsets in /proc/pid/pagemap for the pages for examination.
4) Reading the page state presented as uint64_t for each page in the pagemap. Bits 0-54 of reading value would be PFN [3].
5) Open /proc/kpageflags. Using read PFN seek to that entry in the file (PFN would be the index of a needed uint64_t value in /proc/kpageflags), and read the needed data.

A novel tool *apagescan*[12] has been developed in order to simplify page map data acquisition and collecting statistics from an Android device.

The *apagescan* provides graphical visualization of Android memory state and allows inspecting memory mapping in dynamics. Tool also allows to inspect swapping process, in particular, a 'competition' between apps for the same memory segment - a scenario where multiple processes are located in the same physical memory space and 'struggle' for memory, displacing each other's pages.

The purpose of this project is to visualize processes' memory in RAM and zRAM. Visualization is performed page by page. Page data is provided for each page in RAM.

This article proposes a tool, called *apagescan*, which allows memory examination by displaying memory state in graphical form. Existing applications output statistics of memory usage without providing a graphical representation.

The tool allows scanning processes from a provided process tree of all processes or a control group - Linux interface for grouping processes, so a developed tool is also able to examine processes in available cgroups.

The tool provides the following functionality:

- Collecting data about memory state of multiple processes.
- Both static and dynamic visualization of collected data.
- Selection of processes both from all active processes and from cgroups.

A. Solution design and used technologies

The *apagescan* is developed using Python programming language. PyQt5 module was used to create a graphical user interface. The developed software product supports Model-View-Presenter (MVP) architecture.

The application is based on collecting and processing page data. *apagescan* uses additional data retrievers located on device in order to obtain data. Data retrievers are designed to collect data and save it to a file for further transfer to the application and processing.

B. Device interaction

Data retrievers are written in the C-language due to the need for reading large amounts of memory pages in a short time. They are compiled with gcc-arm-linux-gnueabi compiler and then transferred to a working directory on device using the Android Debug Bridge (ADB [13]) tool. Data retrievers are used in the application, but they can also be executed manually from command line using the ADB shell. Inside the application, data retrievers are being operated using a python 'subprocess' module and the ADB shell.

For acquiring data from the mobile device a set of special programs (data retrievers) has been developed in C-language in order to reach maximum performance.

Data retrievers[1]:

1) *get_pid_list.c:* This program is designed to get PIDs that are currently running on mobile device. Each running process has its own folder in the procFS. Folder's name is the same as processes ID. get_pid_list scans the procFS and returns a .csv file with each process ID and name.
2) *read_cgroup.c:* This program is designed to collect data to visualize a hierarchy of processes in CGroup. It receives a path to a CGroup tasks file, reads it and for each PID in CGroup scans it's a status file to get processes' parent PID and name. Tree of dependencies could be built using pairs of PID and it's parent PID. It returns a .csv file with PID itself, it's parent PID and PID's name for each PID in tasks file.
3) *page_tool.c:* This program is designed to collect data for further analysis. It receives PID and a path for a file to save data and then uses a kernel pagemap interface to collect process page data. The tool scans files "pagemap", "maps" and "kpageflags" located in the procFS in a folder of a process to get information about each page. Page information includes page address (the PFN or swap offset depending on page's type - present or swapped) and flags data - bitwise representation of page-attributes: swapped, dirty, anon, etc. The data retriever creates a binary file with all collected information, located at the given path.

[1]All sources are available in repo github.com/OSLL/apagescan

C. apagescan *interface*

The *apagescan* provides a GUI interface. Elements of the interface are described below.

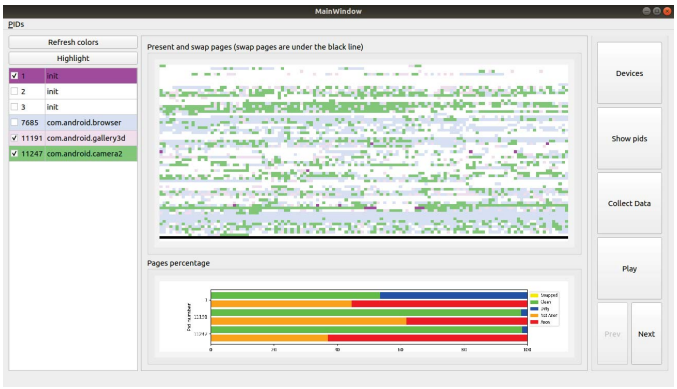

Fig. 1. Screen of the application

Control panel contains the following buttons:

- *"Devices"* button opens a dialog to choose a working device from the list of all devices connected to PC. All further actions will affect only the chosen device. Device dialog is automatically opened when *apagescan* launches.
- *"Show PIDs"* button opens a dialog that allows a user to select several PIDs from the list of all PIDs currently running on device. Selected PIDs will be available in Active PIDs widget on the left panel.
- *"Collect data"* button opens a dialog window asking a user to enter two parameters: time between iterations of data collecting and total time of work. Using these parameters, *apagescan* collects memory usage data from device for each PID from ActivePIDs widget and then plots graphics. Amount of graphics is calculated using two parameters described above. Graphics are being displayed at the center of the screen.

- *"Play"* button shows all iterations graphics one-by-one with 0.5 seconds delay. 'Prev' button and 'Next' buttons show the previous and next iteration graphic respectively.

"Active PIDs" panel, placed on the left side, contains information about each PID that was chosen by opening *"Show PIDs"* or *"Show CGroups"* window. *"Active PIDs"* panel supports right-clicking PID, checking PID in the table and refreshing PID's color. By right-clicking PID a user can call context menu with two options:

- *"Show full information about PID"* action opens a table with information about each page of a process. The page contains page offset and flags: present, shared, anon.
- *"Change PID color"* action opens a dialog, which allows changing PID color on all graphics.

Each PID can be selected using checkbox. Selected PIDs can be highlighted using *"highlight"* button. PIDs colors can be re-generated by clicking *"Refresh colors"* button.

Memory pages are being plotted on the *"offset graph"* in blocks with specified size. Blocks are plotted according to pages offset in memory. Therefore, if the upper left corner on the plot is offset of 0 and the lower right is a maximum offset (the "last" page in RAM), then each dot on canvas could be connected with a page in memory. Having pages of process scanned (using *"Collect data"* button), page in blocks are displayed using a color from the table in *"ActivePIDs"* widget. Offset graph contains two areas related to RAM and zRAM. Areas are divided by black line. For closer examination, *"offset graph"* has a zoom option.

Pages' states percentage widget contains two graphs:

- *Swapped-present-dirty graph* shows the percentage of swapped, present and dirty pages (each page is either swapped, present or dirty).
- *Shared-anon graph* shows the percentage of anon and shared pages (each page is either shared or anon).

PID's menu contains the following options:

- *"Show CGroup tree"* option shows a widget containing information about available CGroups on connected device. There is an option to choose a control group and display it as a tree. PIDs in a cgroup's tree can be selected for closer examination.
- *"Iterations"* option opens the table containing time information about previous iterations of data collecting. Table has three columns: system time of the beginning of the iteration, the same for the end of the iteration, and the last column contains duration of the iteration.

V. Evaluation

A. Application usage scenario

The application was tested on Nexus 5 with 2GB of RAM, modified native core 3.4.0. with swap support and Android 6.0.1. On Nexus 5, three applications were launched and used total for two minutes, while *apagescan* was collecting data with 0 sec. delay between measurements. Finally, 60 physical-memory snapshots were made. Camera (blue color - 5837 PID), browser (fuchsia color - 5307 PID) and gallery (green color - 5390 PID) applications were used in this experiment. Fig. 2 presents the start state of memory when browser app was used, and other applications were in the background.

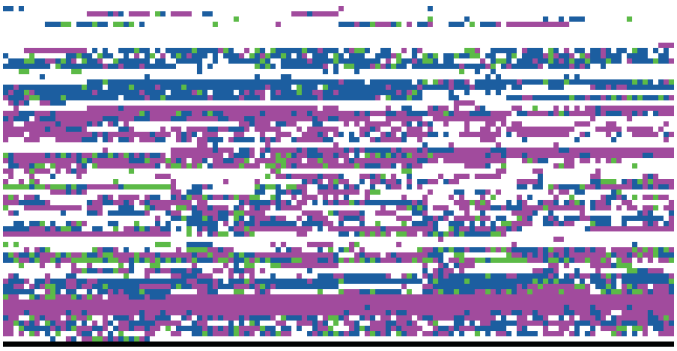

Fig. 2. Physical memory snapshot. Browser is in the fore-ground, gallery and camera are in the background. Browser -fuchsia. Gallery - green. Camera - blue.

On Fig. 2 the big part of RAM is being occupied by browser and much less by camera and gallery. Also, there are no pages in the swap area. While browser was used, there were some changes in memory mapping but they became noticeable only when browser app went to the background and camera app was chosen (See Fig. 3). Some pages were displaced by camera's pages but most of them stayed at their places and the camera's pages just took free space.

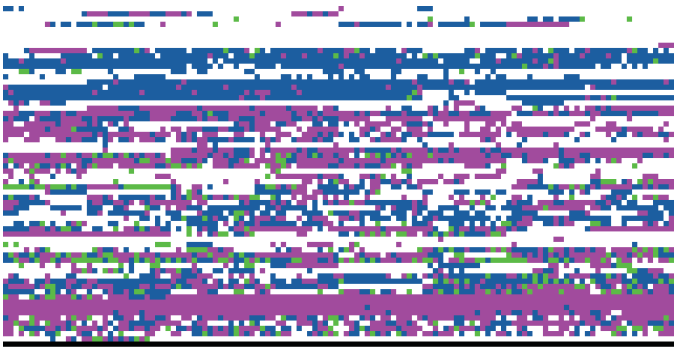

Fig. 3. Physical memory snapshot. Camera is in the fore-ground, gallery and browser are in the background. Browser - fuchsia. Gallery - green. Camera - blue.

Fig. 4-5 show 'competition' between camera app and gallery app (red area) and the process of evicting pages, when the app goes to the background (blue area).

For these two states percentage graphs were represented below (see Fig. 6-7). On these graphs, the amount of not anonymous and dirty pages reduced for gallery and increased for camera.

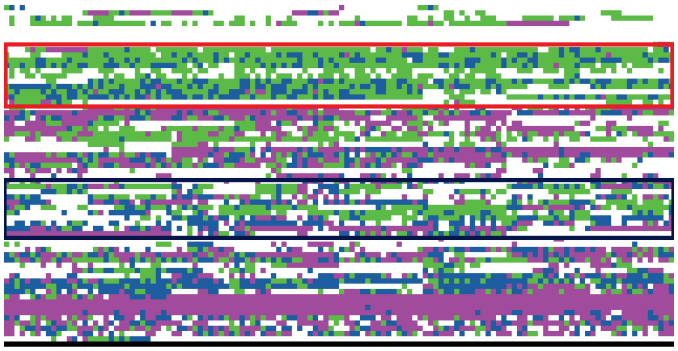

Fig. 4. Physical memory snapshot. Gallery is in the fore-ground, camera and browser are in the background. Browser - fuchsia. Gallery - green. Camera - blue.

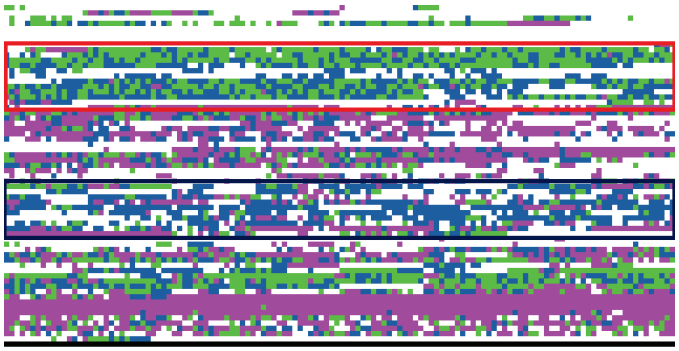

Fig. 5. Physical memory snapshot. Camera is in the fore-ground, gallery and browser are in the background. Browser - fuchsia. Gallery - green. Camera - blue.

Often it's necessary to trace memory usage of the system in order to determine the program that consumes most of the memory resources or the program that is responsible for slowing down other processes. The *apagescan*'s graphical representation of device's physical memory in such convenient way allows to detect memory consumers instantly - user can visually see processes that take the most memory space, and see how fragmented process's memory is. Barplot graphics provide page statistics that allow seeing which pages mostly make up the process memory. Statistics can be customized by changing the set of flags used to determine the pages' type.

Graphical representation of memory state also allows detecting processes that are trying to use the same segment of memory. Inspecting memory segment several times can show, that two or more processes are allocating memory from this segment and 'compete' for pages: after the first process allocates a page, another process re-allocates it to itself and the first process has to allocate memory again. Such cases slow system's work and may come to trashing. *Apagescan* provides the ability to scan memory state every time interval. Inspecting collected graphs can help to detect 'competing' processes.

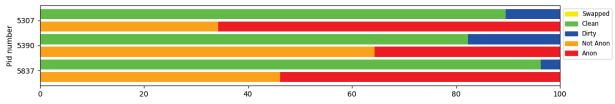

Fig. 6. Percentage plot for state 1. Browser - 5307 PID. Gallery - 5390 PID. Camera - 5837 PID.

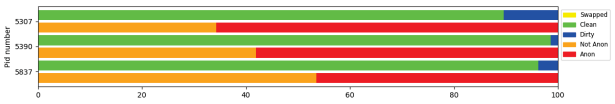

Fig. 7. Percentage plot for state 2. Browser - 5307 PID. Gallery - 5390 PID. Camera - 5837 PID.

B. Data collection time analysis

The number of memory snapshots depends on how long data is collected and pulled from device. As mentioned in the section 3.3. clicking on the "Collect data" button proposes to enter two parameters: time between iteration of data collectings (iterations_time) and total time of data collecting (total_time).

apagescan can't predict time of data collection for chosen PIDs for several reasons:

- accessibility of data for the current process
- existence of the current process. For example, if the process finished after data collecting started
- a number of pages for the current process

On the plot below the dependence of time for data collecting and pulling it to computer on the number of pages is shown. The sample for this graph was collected by choosing all processes and running data collection algorithm 100 times. On the graph, there is decent time spread for the same number of pages, as well as emissions. The main cause of emissions is the ADB tool working and there is no way to influence it. Therefore, to observe the dynamics of time change depending on the number of pages, an interpolation curve was drawn.

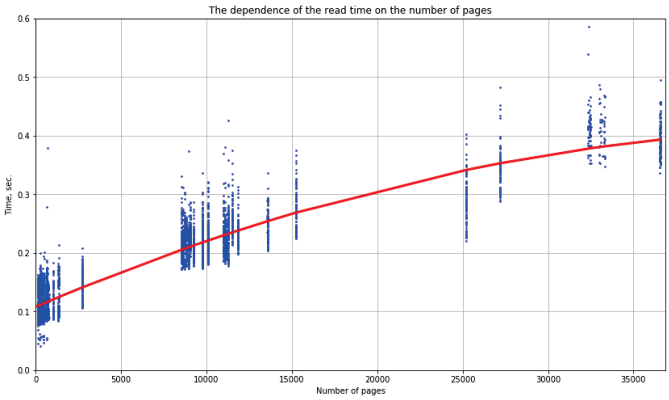

Fig. 8. The dependence of read time on the number pages

Analyzing this graph, it can be noted that time of data

reading for inaccessible or finished processes takes 0.11 sec. For processes with 36000 pages, it takes 0.4 sec. This graph allows to predict time taken to collect data for certain number of pages, but the general problem of the undefined number of pages for the current process still exists.

C. Comparison

This article introduces *apagescan* - a new tool for memory state inspection and visualization. The comparison between *apagescan* and described above existing tools is presented below.

TABLE I. COMPARISON OF TOOLS

Criteria\Tool	*apagescan*	Android Studio	MAT	free	/proc/meminfo	vmstat	top
Memory usage	+	+	+	+	+	+	+
Memory fragmentation	+	-	-	-	-	-	-
Real-time executing and dynamics	+	+	+	-	+	+	+
Selection of processes	+	+	-	-	-	-	+
Control group support	+	-	-	-	-	-	-
Scanning zRAM	+	-	-	+	+	+	+
Convenient output	+	+	+	-	-	-	-

IDE tools (Android studio and Memory Analyzer Tool) are determined to application debugging, therefore lacking information about memory fragmentation and unable to show swap usage or memory distribution between processes, while they are providing detailed info about physical memory usage for a single process. Console Linux (and Android) tools mostly give an overall overview of memory usage (total, free and etc.), unable to show dynamic memory changes, don't support cgroups or selection of processes for inspection. The 'top' program is the most useful for the whole memory state analysis, it provides real-time dynamic information about memory usage (including swap) for multiple processes, supports the selection of processes, but unable to show memory fragmentation and provides console table output, which is hard to process on the fly.

Comparing to that, *apagescan* collects information about memory distribution (including swap) between multiple processes, memory fragmentation, allows to create memory snapshots and capture dynamic memory changes, supports processes selection (from cgroups as well), and provides output in form of convenient graphical representation, which allows user to perform a quick analysis of memory state.

Advantages of *apagescan*:

1) Memory usage information for multiple processes
2) Visualization of memory distribution between processes
3) Memory fragmentation statistics (page by page information)
4) Inspection of zRAM (swap)
5) Graphical user-friendly interface
6) Convenient, human-readable output
7) Selection of any available process for inspection
8) Selection from cgroups for inspection

Disadvantages:

1) *apagescan* must be installed additionally on the user's computer. Other tools are provided by Android Studio, Eclipse or Linux OS, which may be convenient for developers. They can investigate the performance inside IDE, where they write and fix code instantly.

VI. CONCLUSION

In some cases, page-level memory analysis is required. Well known memory analysis tools described in III do not suggest any solution based on pagemap interface. *apagescan* is a new tool to scan RAM and zRAM of any rooted Android device. *apagescan* supports two interfaces: CGroup and Pagemap. Pagemap simplifies investigating Android management of physical memory. *apagescan* works with the procFS to read any amount of pages in RAM, providing information about each page, including present/swapped/dirty and anon/not anon flags. *apagescan* is connected to the CGroup interface, so it can be used to analyze processes from any chosen by user control group.

The developed application allows user to examine the particular state of memory at some point in time and to see dynamics of changes in the memory state for a certain period. *apagescan* provides information that is easy to understand, graphs plotted by the tool can be customized for better understanding. All pictures are also available in a png format. The tool is designed to show RAM changes in dynamics. The tool has options to scan RAM every time interval to see how it changes during work with device.

The advantage of the tool is a convenient visualization of information in form of pages, combined in blocks with specified size, selection of processes from cgroups, scanning zRAM. On the other hand, the disadvantages are slowness, difficulties with interpreting a large number of pages or processes, lack of memory leak detection.

Different processes can occupy the same sections of memory, so the points on the graph overlap each other. A selection of these overlay areas can be added to the application. It is also possible to add other flags about pages to C programs and supplement the tool with an analysis of data received.

REFERENCES

[1] Wikibooks, Linux kernel Memory, Web: https://en.wikibooks.org/wiki/The_Linux_Kernel/Memory
[2] Open Source and Linux Lab, Linux memory management summary, Web: http://wiki.osll.ru/doku.php/etc:users:jcmvbkbc:linux-mm
[3] Linux kernel documentation, pagemap, Web: https://www.kernel.org/doc/Documentation/vm/pagemap.txt
[4] Linux kernel documentation, cgroups, Web: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.
[5] Linux Programmer's Manual, cgroups, Web: http://man7.org/linux/man-pages/man7/cgroups.7.html
[6] Brahler, Stefan, *Analysis of the android architecture*. " Karlsruhe institute for technology 7.8. ", 2010.
[7] Android Device Monitor, Web: https://developer.android.com/studio/profile/monitor
[8] Vogel, Lars, *Eclipse Memory Analyzer (MAT)Tutorial*. , 2013.

[9] Maxwell, Evan K., Godmar Back, and Naren Ramakrishnan, *Diagnosing memory leaks using graph mining on heap dumps.* " Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. ", 2010.
[10] Linux Programmer's Manual, proc pseudo-filesystem, Web: http://man7.org/linux/man-pages/man5/proc.5.html
[11] Linux Programmer's Manual, top, Web: http://man7.org/linux/man-pages/man1/top.1.html
[12] Android page scan tool, Web: https://github.com/OSLL/apagescan
[13] Android developers official website, Android Debug Bridge (adb) Web: https://developer.android.com/studio/command-line/adb
[14] Android developers official website, Platform description, Web: https://developer.android.com/guide/platform
[15] M. Gorman *Understanding the Linux virtual memory manager*. Upper Saddle River: Prentice Hall, 2004.
[16] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*. " O'Reilly Media, Inc.", 2013.
[17] S.S. Hahn, S. Lee, I. Yee, D. Ryu, J. Kim, "Fasttrack: Foreground app-aware i/o management for improving user experience of android smartphones", *In 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, Aug. 2018, pp. 15-28.
[18] K. Vimal, A. Trivedi, "A memory management scheme for enhancing performance of applications on Android", *In 2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, Dec. 2015, pp. 162-166.
[19] K. Baik, J. Huh, "Balanced memory management for smartphones based on adaptive background app management", *In The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, Oct. 2014, pp. 1-2.
[20] Park, Jihyun, and Byoungju Choi, *Automated memory leakage detection in android based systems.* " International Journal of Control and Automation 5.2. ", 2012, pp. 35-42.
[21] Seyfried, Stefan, *Resource management in linux with control groups.* " Proceedings of the Linux-Kongress. ", 2010.
[22] Seo, Jaebaek, *FLEXDROID: Enforcing In-App Privilege Separation in Android.* " NDSS. ", 2016.