Survey on Android Memory Management System

June 13, 2012

Garza Matteo
Matr. 755295, (matteo.garza@mail.polimi.it)
Tania Suarez Legra
Matr 748927 (tania.suarez@mail.polimi.it)

Report for the master course of Real Time Operative System (RTOS)
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: April, 01 2011

Abstract

Android Operative System[?] is the most diffuse OS in mobile devices. In this paper we will analyze how Android manages memory. We discuss in particular about application memory and some of the most used MMUs used by Android OS.

1 Kernel Memory Management

1.1 Introduction

Android[?, ?, ?] is a Linux-based operative system, written in C and C++. Android application software runs on a framework which includes Java-compatible libraries. Android uses the Dalvik virtual machine with just-in-time compilation to run Dalvik dex-code (Dalvik Executable), which is usually translated from Java bytecode. Figure **??** shows the actual[?] distribution of Android version between devices with this kernel:

C:/home/coach/Dropbox/RTOS/roba su github/Android_chart.png

We can notice that now the common kernel distributions still use Linux 2.6.x kernel.

Figure **??** shows Android System Architecture schema [?].

C:/home/coach/Dropbox/RTOS/roba su github/System-architecture.jpg

Android [?] provides some modification to main Linux kernel, such as an improved power management, ASHMEM virtual memory, some specific-component drivers, and a low memory killer. The latter's mission is to free memory when the system run Out of Memory (OOM).

1.2 Dalvik Virtual Machine

Dalvik is the VM in which Android applications are run. Is structured to work with devices with limited resources:

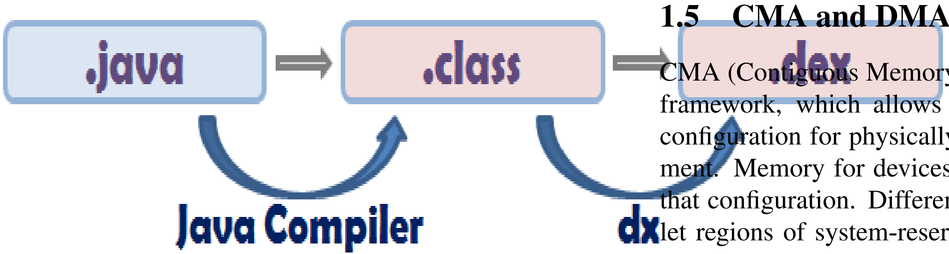
- Relatively slow CPU
- Small amount of RAM
- No swap space

This VM executes Dalvik bytecode, which is compiled from programs written in the Java. But note that Dalvik VM is not a Java VM (JVM).

Every Android application runs in its own process, with its own instance of the Dalvik virtual machine, in this way the applications work in an isolated manner and do not compete with each other.

Dalvik was written so that a device can run multiple VMs efficiently. The Dalvik VM executes code in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.

An uncompressed .dex file is typically a few percent smaller in size than a compressed .jar (Java Archive) derived from the same .class files.



1.3 Low level management and integration with HW resources

In this part, we discuss about how the memory has managed in Android devices. For most of the releases in Android, it was used PMEM and ASHMEM. These kind of libraries was too simple, and was patched with some SoC patches, such as NVMAP for nVidia Tegra devices and CMEM for TI OMAP ones. The most important patch was CMA (Contiguous Memory Access), expecially with DMABUF patch. With the release of Android 4.0 (Ice Cream Sandwich) a brand new driver has released, ION. We discuss about differences between ION and CMA approach, and, in the state-of-art, we discuss of a future integration between them.

1.4 PMEM and ASHMEM

PMEM (Process MEMory)[?] is the first memory driver implemented on Android devices (since G1). It is used to manage shared memory regions sufficiently large (from 1 to 16MB).

This regions must be physically contiguous between user space and kernel drivers (such as GPU, or DSP). It was written specifically to be used in a very limited hardware platform, and it could be disabled on x86 architectures.

ASHMEM[?] (Android SHared MEMory) is a shared memory allocator subsystem, similar to POSIX, but with a different behavior. It also gives to the developer an easier and file-based API. It used named memory, releasable by the kernel. Apparently, ASHMEM supports low memory devices better than PMEM, because it could free shared memory units when it is needed.

PMEM	ASHMEM
Uses physically contiguous addresses	Uses virtual memory
The first process who instantiate a memory heap must keep that till the last one of the users won't free the file descriptor. Thus to preserve contiguity	Memory is handled by instances (object oriented like). It is managed by a reference counter

1.5 CMA and DMABUF

CMA (Contiguous Memory Allocator)[?] is a well known framework, which allows setting up a machine-specific configuration for physically-contiguous memory management. Memory for devices is then allocated according to that configuration. Differently from similar framework, it let regions of system-reserved memory to be reused in a transparent way, letting memory not to be wasted. When an alloc is instantiated, this framework migrates all the system page. Thus to build a big chunk of physically contiguous memory.

Why do an OS have to use chunks of memory?[?, ?] Because virtual memory tends to fragment pages. An intensive use of memory let the system not able to find contiguous memory in a very short time after boot. Recently, the requirement of huge pages in applications raises, especially for transparent huge pages. Another question is devices (such as cameras) that needs DMA over areas physically contiguous. CMA reserve an huge area of memory at boot time, only for huge request of memory. For every region, block of pages can be flaggable as three type.

- movable : typically, cache pages or anonymous pages, accessed by page table or page cache radix tree
- kernel recallable : they can be given back to the kernel by request.
- immovable : these are typically pointer referred pages (such as pages invoked by a kmalloc())

The memory manager subsystem try to keep movable pages as near as possible. Grouping these pages, kernel try to ensure more and more contiguous free space available for further request. CMA extends this mechanism. It adds a new type of migration (CMA). Pages flagged as cma behave like the movable ones, with some differences:

- they are “sticky”
- Their migration type can't be modified by the kernel
- In CMA Area, the kernel cannot instantiate pages not movable.

In other words, memory flagged as CMA keep available for the rest of the system with the only restriction to be movable.

When a driver ask for a huge contiguous allocation of memory, CMA allocator can try to free in his own area some contiguous pages to create a buffer large as needed. When the buffer is no longer requested, memory can be used for other needs. CMA can just take only the needed amount of memory without worrying about strictly request of alignment.

DMA buffers has different request despise of huge pages.

DMABUF	Transparent Huge Pages
Normally larger than Transparent Huge Pages. 10 Mb. It could be needed specific memory area, if underlying hardware is sufficiently "strange"	Almost 2Mb large
DMA requires less alignment than THP	2MB of THP needs 2Mb of Alignment

CMA patches provides a set of function that can prepare regions of memory and the creation of contest area of a well known size using function cm_alloc and cm_free to keep and release buffers. CMA must not be invoked by the driver, but from DMA support functions. When a driver call a function like dma_alloc_coherent(), CMA should be invoked automatically to satisfying the request. This should work in normal condition.
One of the issue about CMA is how to initially alloc this area of memory. Current scheme needs that some of special calls should be done by the board file system, with a very arm-like approach. The idea is to do that without board files. The ending result is that it should be at least one iteration of that patch set before it will be executed by the mainline.

1.6 ION

In december 2011, PMEM is marked as deprecated, and then replaced by ION memory allocator[**?**]. ION is a memory manager that Google has developed from the 4.0 release of Android (Ice Cream Sandwich), mainly to resolve the interface issue between different memory management between different Android device. In fact, some SoC developer implemented different memory manager. We can cite some of them:

- NVMAP, implemented on nVidia Tegra
- CMEM[**?**], implemented on TI OMAP
- HWMEM[**?**], implemented on ST-Ericsonn devices

All this vendor will pass to ION soon
Besides ION being a memory pool manager, it also enables his clients to share buffers (so, it works like DMABUF, the DMA buffer sharing framework). Like PMEM, ION manages one or more pools of memory, some of them instantiated at boot time or from hardware blocks with specific memory needs. Some devices like that are GPU, display controllers and cameras. ION let his pools to be available as heap ION. Every kind of android device can have different ION heaps, depending on device memory. Phisical address and heap dimension can be returned to the programmer only if the buffer is physically contiguous. Buffer can be prepared or deallocated to be used with DMA, or with virtual kernel addressing. Using a file descriptor, it can be also mapped in the user-space. There are three kind of allocable ION heap. Other ones can be defined by SoC producers (like ION_HEAP_TYPE_SYSTEM_IOMMU for hardware blocks equipped with IOMMU driver).

- ION_HEAP_TYPE_SYSTEM
- ION_HEAP_TYPE_SYSTEM_CONTIG
- ION_HEAP_TYPE_CARVEOUT : in this case, carveout memory is physically contiguous and set as boot.

Typically, in the user-space case, libraries uses ION to alloc large continuous buffers. For instance, camera library can alloc a capture buffer to be used from the camera device. Once the buffer is fulfilled with video data, the library gives the buffer to kernel to be processed by jpeg encoder block. A c/c++ program must have access to '/dev/ion' before it can alloc memory thanks to ION. He can alloc data using file descriptors (fd). It can be maximum one client for user process.

Clients interacts from user-space with ION using ioctl() system interface. Android processes can share memory using their fd. To obtain shared buffer, the second user process must obtain a client handle through a system call open('/dev/ion', O_RDONLY). ION manage user space client through process PID (in particular, the 'group leader ' one). Fd will be instantiated pointing at the same client structure in the kernel. To free a buffer, the second client must invalidate the mmap() effect, with an explicit call at munmap(), and the first client must close the fd, calling ION_IOC_FREE. This function decrements the reference counter of the handle. When it reaches zero, the ion_handle is destroyed, and the data structure that manages ION is updated. While managing client calls, ION validates input from fd, from client and from handler arguments. This validation mechanism reduce the probability of undesired access and memory leaks. Ion_buffers is somewhere similar to DMABUF. Both uses anonymous fd, reference counted, as shareable objects.

	ION buffers	DMABUF
Application level MMU	Alloc and free memory from memory pools in a shareable and trackable way	It focus on import, export and syncronization in a consisten way with buffer sharing solution for non arm architectures
Role of Memory manager	ION replace PMEM as memory pools manager. ION heap lists can be extended by the device.	DMABUF is a buffer sharing framework , designed to be integrated with memory allocator in contiguous DMA mapping framewors, such as CMA. DMABUF exporters can implement custom allocator.
User Space access control	ION offers /dev/ion interface to user space program, letting them to alloc and share buffers. Every user process with ION access can suspend the system overlapping ION heap. Android chech user and groupID blocking non authorized access to ION heap	DMABUF offers only kernel API. Access control is a function of the permissions on device that uses DMABUF feature
Global Client and Buffer Database.	ION has a driver associated to /dev/ion. The device structure has a database that keeps ION buffers allocated, handlers and fd, grouped by user client and kernel client. ION validates all the client calls to be valid for database rules. For instance, an handle can't have two buffers associated.	The debub structure of DMA implements a global hashtable, dma_entry_hash, tracking DMA buffers, but only when kernel is build with CONFIG_DMA_API_DEBUG option.
Cross-architecture usage	ION usage now is limited on architectures that runs kernel Android	DMABUF usage is cross architecture. DMA mapping redesign let his implementation in 9 architectures beside the ARM one.
	ION_buffer	DMABUF
Buffer Syncronization	Ion consider the syncronization problem as an orthogonal problem	DMABUF gives a pair of API for synchronization. Buffer user invokes dma_buf_map_attachment() everywhere he desires to use buffer for DMA. Once he finished using that, signals "endOfDMA" to exporter using dma_buf_unmap_attachment()
Buffer delayed allocation	ION allocs physical memory before the buffer is shared	DMABUF can delay allocation till the first call of dma_buf_map_attachment(). DMA buffer exporter has the opportunity of scans every client attachment, collecting all the constraints and choose the most efficient storage
Integration with Video4 Linux2 API	Processes that uses these API tends to use PMEM. So, the migration from PMEM to ION has a relatively small impact.	DMABUF integration with Video4Linux is hard and asked for lots of modifies in DMABUF. But in a long time that will be a smart choice, because DMABUF sharing mechanism is fitted for DMA, so it is well written for CMA and IOMMU. Both of them reduces carveout memory needs to build an Android smartphone.

2 OOM Killer

2.1 Introduction

Mobile devices become more and more rich of memory over time, due to Moore's Law. However, there's always a limit over wich memory isn't available, and a well form kernel needs some politics to free bunch of memory when needed. Android provides an OOM killer, who kills processes with some heuristics, letting memory to be used from someone else. OOM killer mechanism are implemented in most of Linux kernel.

Major distribution kernels set the default value of /proc/sys/vm/overcommit_memory to zero, which means that **processes can request more memory than is currently free in the system**. This is done based on the heuristics that allocated memory is not used immediately, and that processes, over their lifetime, also do not use all of the memory they allocate. Without overcommit, a system will not fully utilize its memory, thus wasting some of it. Overcommiting memory allows the system to use the memory in a more efficient way, but at the risk of OOM situations. Programs who need lots of memory can consume all the system's memory, stopping the whole system. In such a situation, the OOM-killer kicks in and identifies the process to be terminated.

2.2 OOM Killer parameters

The process to be killed in an out-of-memory situation is selected based on its badness score. The badness score is reflected in /proc/<pid>/oom_score. This value is determined on the basis of four characteristics:

- the system loses the minimum amount of work done,
- recovers a large amount of memory,
- doesn't kill any innocent process,
- and kills the minimum number of processes (if possible limited to one).

The badness score is computed using

- the original memory size of the process,
- its CPU time (utime + stime),
- the run time (uptime - start time)
- and its oom_adj value.

The more memory the process uses, the higher the score. The longer a process is alive in the system, the smaller the score.

Any process unlucky enough to be in the swapoff() system call (which removes a swap file from the system) will be selected to be killed first. For the rest, the initial memory size becomes the original badness score of the process. Half of each child's memory size is added to the parent's score if they do not share the same memory. Thus forking servers are the prime candidates to be killed. Having only one "hungry" child will make the parent less preferable than the child. Finally, the following heuristics are applied to save important processes:

- if the task has nice value above zero, its score doubles
- superuser or direct hardware access tasks (CAP_SYS_ADMIN, CAP_SYS_RESOURCE or CAP_SYS_RAWIO) have their score divided by 4. This is cumulative, i.e., a super-user task with hardware access would have its score divided by 16.
- if OOM condition happened in one cpuset and checked task does not belong to that set, its score is divided by 8.
- the resulting score is multiplied by two to the power of oom_adj (i.e. points <<= oom_adj when it is positive and points >>= -(oom_adj) otherwise).

The task with the highest badness score is then selected and its children are killed. The process itself will be killed in an OOM situation when it does not have children.

2.3 lowmemory driver in Android

The Android developers required a greater degree of control over the low memory situation because the OOM killer does not interfere till late in the low memory situation, i.e. till all the cache is emptied. Android wanted a solution which would start early while the free memory is being depleted. **So they introduced the "lowmemory" driver**, which has multiple thresholds of low memory.

In a low-memory situation, **when the first thresholds are met, background processes are notified of the problem**. They do not exit, but, instead, save their state. This affects the latency when switching applications, because the application has to reload on activation. On further pressure, the lowmemory killer kills the non-critical background processes whose state had been saved in the previous threshold and, finally, the foreground applications.

Keeping multiple low memory triggers gives the processes enough time to free memory from their caches because in an OOM situation, user-space processes may not be able to run at all. All it takes is a single allocation from the kernel's internal structures, or a page fault to make the system run out of memory. An earlier notification of a low-memory situation could avoid the OOM situation with a little help from the user space applications which respond to low memory notifications.

Killing processes based on kernel heuristics is not an optimal solution, and these new initiatives of offering better control to the user in selecting the process to be the chosen one to terminate are steps to a robust design to give more control to the user.

2.4 User space OOM control

/proc/<pid>/oom_score is a dynamic value, not so much controllable and checkable i by the administrator. It is difficult to determine which process will be killed in case of an OOM condition. **The system must let the administrator to modify the score** for every process created, and for every process which exits. In an attempt to make OOM-killer policy implementation easier, a **name-based solution** was proposed. With his patch, the process to die first is the one running the program whose name is found in /proc/sys/vm/oom_victim. A name based solution has its limitations:

- task name is not a reliable indicator of true name and is truncated in the process name fields. Moreover, symlinks to executing binaries, but with different names will not work with this approach
- This approach can specify only one name at a time, ruling out the possibility of a hierarchy
- There could be multiple processes of the same name but from different binaries.
- The behavior boils down to the default current implementation if there is no process by the name defined by /proc/sys/vm/oom_victim. This increases the number of scans required to find the victim process.

possible solution is using containers. The patch introduces an OOM control group (cgroup) with an oom.priority field. **The process to be killed is selected from the processes having the highest oom.priority value.**
If you want to create a special control group containing the list of processes which should be the first to receive the OOM killer's attention, create a directory under /mnt/oom-killer to represent it: Add the PID of the process to be added to the list of tasks:
This approach could have some trouble, in presence of multiple cpuset. Consider two cpusets, A and B. If a process in cpuset A has a high oom.priority value, it will be killed if cpuset B runs out of memory, even though there is enough memory in cpuset A.
An interesting outcome of the discussion has been handling OOM situations in user space. **The kernel sends notification to user space, and applications respond by dropping their user-space caches**. In case the user-space processes are not able to free enough memory, or the processes ignore the kernel's requests to free memory, the kernel will kill them.

- the cgroup OOM notifier allows you to attach a task to wait on an OOM condition for a collection of tasks. This allows userspace to respond to the condition by dropping caches, adding nodes to a cpuset, elevating memory controller limits, sending a signal, etc. It can also defer to the kernel OOM killer as a last resort.
- /dev/mem_notify allows you to poll() on a device file and be informed of low memory events. This can include the cgroup oom notifier behavior when a collection of tasks is completely out of memory, but can also warn when such a condition may be imminent.

[?]

References