Survey on Android Memory Management System

May 24, 2012

Garza Matteo
Matr. 755295, (matteo.garza@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 lots of devices (expecially smartphones and tablets). In this paper we will analyze how Android manages memory on device. We discuss in particular about application memory and some of the most used MMUs used by Android OS.

1 Introduction

TODO

1.1 Memory Management

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 library 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.

1.2 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

1.3 CMA e DMABUF

CMA (Contiguous Memory Allocator) is a well known patch that let the device to alloc big chunk of memory after the system has booted. 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 mi-

grates 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.
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.

1.4 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 The Second Section

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean magna. Nunc non ante eget nibh condimentum tempor. Nullam ullamcorper lectus eget mauris. Nam neque orci; rhoncus at, pulvinar quis, elementum sit amet, turpis. Mauris posuere nisi ut justo. Morbi non lorem vitae mauris interdum faucibus. Vestibulum ut sapien in augue faucibus fringilla. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Etiam vestibulum fringilla libero. Curabitur libero diam, hendrerit sit amet, ornare eget, imperdiet vel, purus!

2.1 The first subsection of the second Section

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam consectetur ante at eros. Vestibulum mi nisi, venenatis sollicitudin, tempus sed, auctor id, tortor. Fusce orci. Duis tellus arcu, euismod sed, consequat sit amet, elementum vel, mauris. Curabitur leo diam; dapibus quis, condimentum vitae, dignissim ut, diam. Nulla et nulla eget elit volutpat sagittis.

2.2 The second subsection of the second Section

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris eget mauris. Nulla facilisi. Ut condimentum tempor eros? Integer metus mauris, consectetur sit amet, tempor a, facilisis eu, nisl. Vestibulum at turpis. Ut vitae tortor pretium nisl vestibulum blandit. Nulla nibh urna, semper et, elementum at, mattis ut, nisi! Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Morbi vel ligula eget lacus convallis venenatis. Aliquam lacinia tincidunt felis. Ut dui.

References