Effective Handling of Low Memory Scenarios in Android

M.Tech MTP Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Rajesh Prodduturi

Roll No: 113050076

under the guidance of

Prof. Deepak B Phatak

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Abstract

Android memory management unit plays vital role in great success of android among other operating systems. Android contains modified Linux kernel for handling low memory scenarios and other issues. This report gives brief details on both linux, android memory management systems. It gives complete details on two killers (OOM, low memory killer), these are used for reclaiming memory in low memory scenarios. Finally it describes different set of problems in OOM killer with experimentation and problems in low memory killer which are causes to degrading user experience. It also contains proposed solutions along with problems identified.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my guide, Prof. Deepak B Phatak for the consistent directions towards my work. Because of his consistent encouragement and right directions, we are able to do this project work.

Chapter 1

Introduction to Embedded Operating Systems

Embedded operating system should have small memory footprint, low power consumption and better performance with low hardware configuration. Computer system that performs a dedicated function or a group of functions by using a specific embedded software on a low hardware configuration system is called embedded operating system.
Ex: symbian, windows CE, android, uclinux

1.1 Android

Android is a linux-based embedded operating system for mobile devices such as smart phones and tablets. It is open source. It supports many architectures like ARM, MIPS, X86. It uses modified linux kernel 2.6. Linux Kernel type is monolithic. Even though android uses linux kernel but it varies in some of the core components.

Figure 1.1 depicts the complete architecture of android. Android contains four layers, one more hidden layer (hardware abstraction layer)also exist. The bottom layer represents the modified linux kernel.

1.1.1 Android Features

1. Kernel Modifications[1]

    - Alarm Driver: provides timers to wake devices up from sleep.
    - Ashmem Driver: Allows applications to share memory and manages the sharing in kernel levels. More details are explained in chapter 03.
    - Binder Driver: It facilitates IPC(Inter process communication). Since data can be shared by multiple applications through the use of shared memory. Binder also takes care of synchronization among processes. IPC is being optimized, as less data has to be transferred

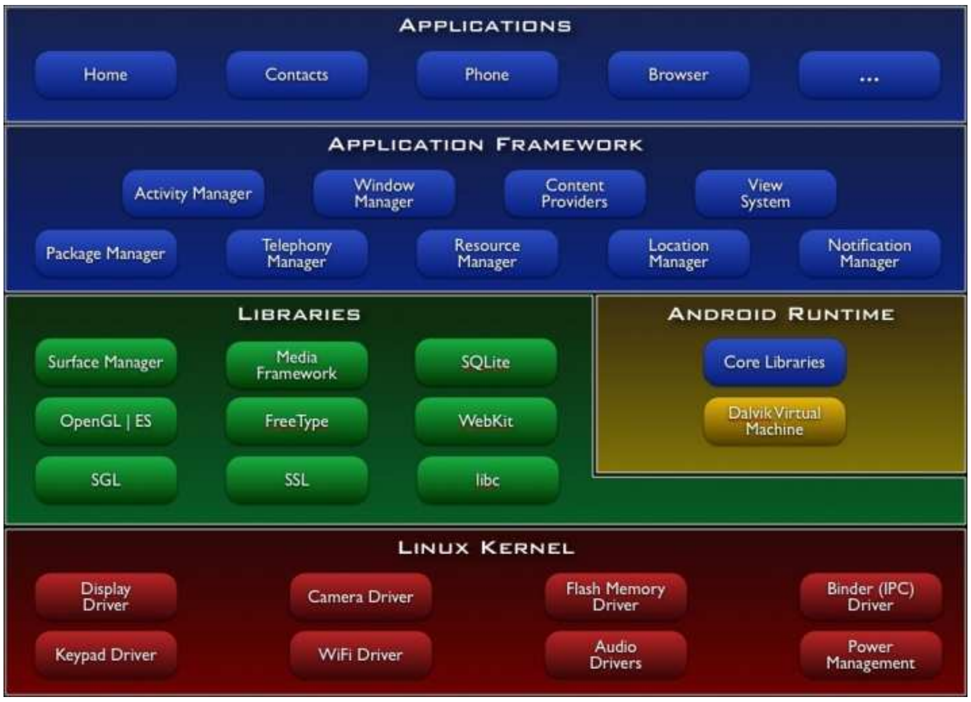


Figure 1.1: Android architecture
[1]

- power management : It takes a more aggressive approach than the Linux PM (power mgmt) solution.

2. SQLite: A lightweight relational database, is used for data storage purposes. Many applications like contacts, calender uses SQLite for storing data in a structural format.

3. WebKit: Android based open source Web browser. It's memory footprint is very less.

4. BIONIC(libc):
It is a Standard C Library. It uses libc instead of GNU glibc. Bionic library has a smaller memory footprint. Bionic library contains a special thread implementation like optimizes the memory consumption of each thread and reduces the startup time for creating a new thread.

5. Dalvik Virtual Machine(DVM):
Many embedded operating systems use JVM(Java virtual machine), but android uses DVM. DVM was prepared to support low config CPU, limited memory, most importantly limited battery power. Application developers write their program in Java and compile java class files. However, instead of the class files being run on a J2ME virtual machine, the code is translated after compilation into a ”Dex file” that can be run on the Dalvik machine. A tool called dx will convert and repack the class files in a Java .jar file into a single dex file. It uses its own byte code, not Java byte code.

6. Storage Media:

Hard disks are too large in size, and consume too much power. So mobile devices uses flash memory. Flash memory is a nonvolatile memory that can hold data without power being supplied.It is usually a EEPROM(Electrically erasable programmable ROM) . Two types of flash memory are available, one is NAND gate based flash memory and another one is NOR gate based flash memory. Some of the silent features of flash memory are it is light weight, low energy consumer and shock resistance. It provides fast read access time, because there is no seek time. Some of the drawbacks of flash memory are five to ten times as expensive per megabyte as hard disk, wear-leveling property[2].

7. File System(**YAFFS**):
   Flash memory uses file system as YAFFS (Yet Another Flash File System). YAFFS is the first flash file system specifically designed for NAND flash. It provides fast boot time. YAFFS uses less RAM than ext4,ext3. YAFFS uses low power consumption than ext4,ext3.

   Each NAND page holds 512 bytes of data and 16 ”spare” bytes. In YAFFS terminology page called as chunk. Collection of chunks is a block (32 chunks). Each file has an id - equivalent to inode. Each flash page is marked with file id and chunk number. Chunks numbered 1,2,3,4 etc - 0 is header.

$BlockId$	$ChunkId$	$ObjectId$	$Delflag$	$OtherComments(Description)$
0	0	100	Live	Object header
0	1	100	Live	First chunk in file
0	2	100	Live	Second chunk in file
0	3	100	Del	Third chunk in file

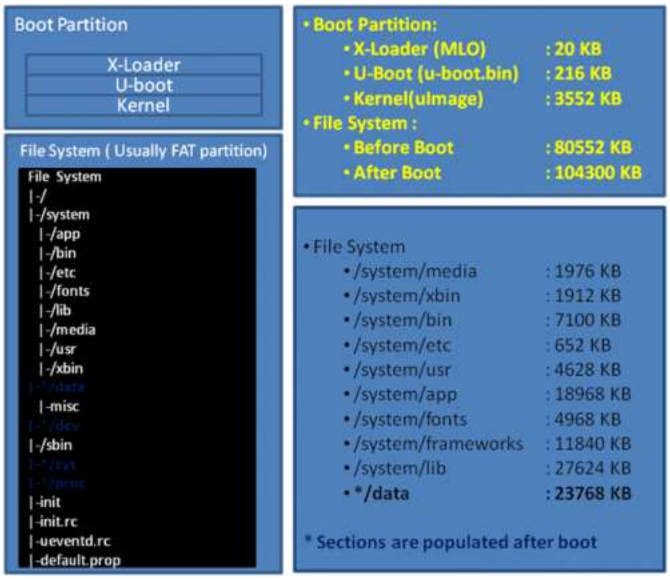
Table 1.1: YFFS File Structure

YAFFS Data structures:

- yaffs_Object: It can be a file, directory, symlink or hardlink. This know about their corresponding yaffs_ObjectHeader in NAND and the data for that object.
- parent: pointer to the parent of this yaffs_Object in the directory structure.
- siblings: link together a list of all the yaffs_Objects in the same directory.
- Children: Points children directories.

1.1.2 Android Memory Analysis

Android contains different components in all layers. Each component occupies different set of sizes in main memory. It is good to know about how much memory needed for each component. Because it helps to reduce the footprint of large memory consume components.



```
File System
|-/
|-/system
  |-/app
  |-/bin
  |-/etc
  |-/fonts
  |-/lib
  |-/media
  |-/usr
  |-/xbin
|-*/data
  |-misc
|-*/dev
|-/sbin
|-*/sys
|-*/proc
|-init
|-init.rc
|-ueventd.rc
|-default.prop
```

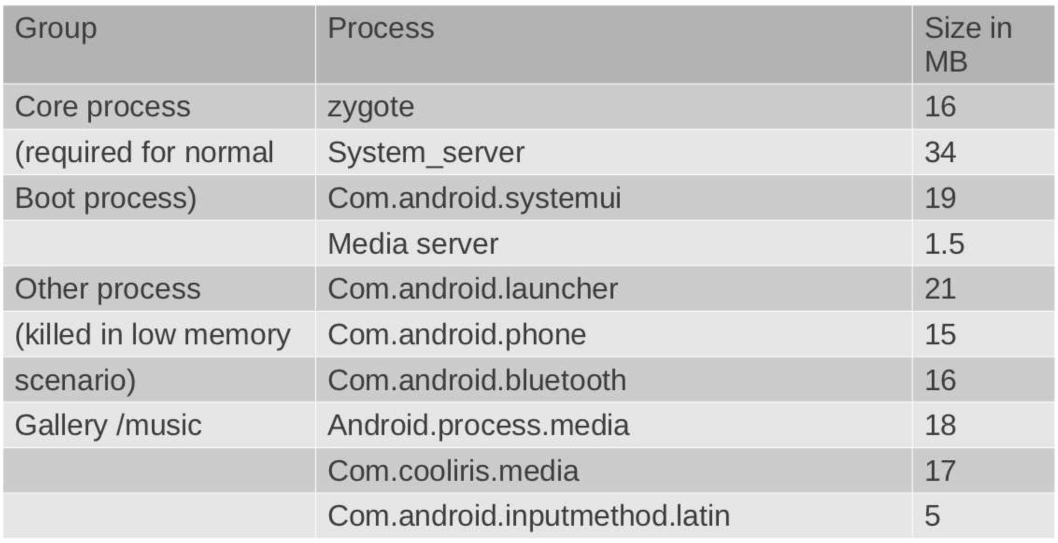
- **Boot Partition:**
  - **X-Loader (MLO) : 20 KB**
  - **U-Boot (u-boot.bin) : 216 KB**
  - **Kernel(uImage) : 3552 KB**
- **File System :**
  - **Before Boot : 80552 KB**
  - **After Boot : 104300 KB**

- File System
  - /system/media : 1976 KB
  - /system/xbin : 1912 KB
  - /system/bin : 7100 KB
  - /system/etc : 652 KB
  - /system/usr : 4628 KB
  - /system/app : 18968 KB
  - /system/fonts : 4968 KB
  - /system/frameworks : 11840 KB
  - /system/lib : 27624 KB
  - ***/data : 23768 KB**

* Sections are populated after boot

Figure 1.2: Android memory footprint
[18]

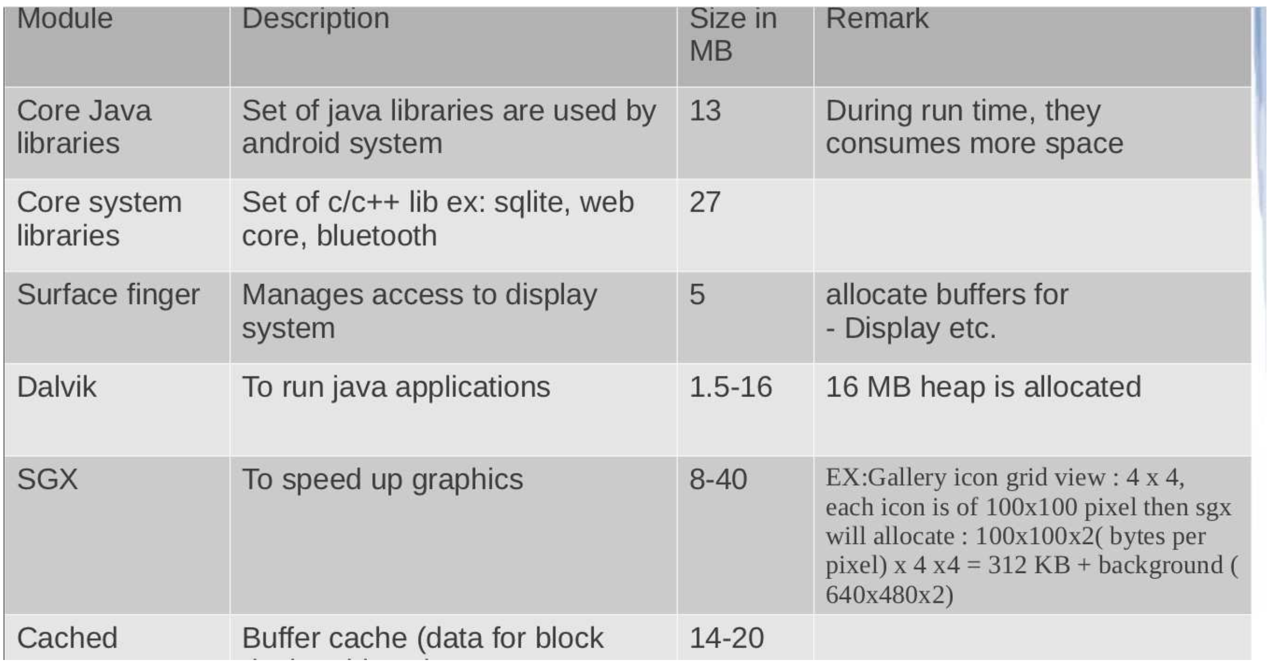
Figure 1.2 shows the memory footprint of Android-Gingerbread-2.3.4. There are many set of open source tools like DDMS(Dalvik debug monitor server) used for analyzing memory footprint of different components.



Group	Process	Size in MB
Core process	zygote	16
(required for normal	System_server	34
Boot process)	Com.android.systemui	19
	Media server	1.5
Other process	Com.android.launcher	21
(killed in low memory	Com.android.phone	15
scenario)	Com.android.bluetooth	16
Gallery /music	Android.process.media	18
	Com.cooliris.media	17
	Com.android.inputmethod.latin	5

Figure 1.3: Android Processes memory footprint
[18]

Figure 1.3 depicts the memory footprint of major process like zygote, system_server etc. And also figure shows different kinds of processes which are killable in low memory situations. More details on killable/unkillable processes are explained in chapter 04. Minimum android boot configuration from figure 1.3, android core process (16 +34+19+1.5) + com.android.launcher (21) - will require around 91.5 MB of memory[18].



Module	Description	Size in MB	Remark
Core Java libraries	Set of java libraries are used by android system	13	During run time, they consumes more space
Core system libraries	Set of c/c++ lib ex: sqlite, web core, bluetooth	27	
Surface finger	Manages access to display system	5	allocate buffers for - Display etc.
Dalvik	To run java applications	1.5-16	16 MB heap is allocated
SGX	To speed up graphics	8-40	EX:Gallery icon grid view : 4 x 4, each icon is of 100x100 pixel then sgx will allocate : 100x100x2(bytes per pixel) x 4 x4 = 312 KB + background (640x480x2)
Cached	Buffer cache (data for block	14-20	

Figure 1.4: Android libraries memory footprint
[18]

Figure 1.4 depicts the memory footprint of libraries, other packages in android layers(frame work, libraries) . Dalvic module consumes 1.5 to 16MB size. And other c/c++ libraries also consume 27MB. Minimal memory requirement to run basic android features (standard android source form Google Inc.) are boot time memory 128 MB, run time memory 256 MB.

1.2 uClinux

uClinux is the Linux distribution, specially designed for embedded microprocessors without the MMU. MMU-less is for low-cost microprocessors (No MMU unit inside processor). MMU-less brings some changes to the kernel.

1.2.1 Memory Management

uClinux provides the memory as a single address space[3]. CPU can directly (sequentially/linearly) address all of the available memory locations. Kernel allocates stack space at the end of the data for the executable. If the stack grows large, it will overwrite the static data and code areas. So developer need to know stack requirements. To support dynamic memory allocation, it reserves some preallocated buffer pool. kernel over commit memory, i.e Kernel allocates large amount of memory as per applications request, without verifying enough memory for back it up. Because normally applications ask for huge memory, they ask for 20MB but uses only 1MB.

It does not have fork( ), brk( ) system calls.It uses vfork( ), mmap( ). Because, fork( ) works on copy-on-write method. Vfork( ) creates a child, if both processes share all their memory space including the stack. vfork() then suspends the parent's execution until the child process either calls exit() or execve().

1. **Advantages**
   - Low kernel size (<512Kb). But linux kernel size (3552Kb).
   - Low size standard C library as uClibc
   - It takes less time for IPC
2. **Disadvantages**
   - Lack of memory protection is that an invalid pointer reference, even by an unprivileged process, may trigger an address error, and potentially corrupt or even shut down the system.
   - Lack of a virtual memory and Security issues

1.3 Organization of the Report

Chapter 01 describes two embedded operating systems android, uclinux. It also gives details on android architecture, memory footprint of android components. Chapter 02 describes memory management unit in linux, it also covers memory allocation/deallocation methods and memory reclaiming algorithm(PFRA). Chapter 03 covers differences between android and linux with respect to OOM vs low memory killer, shared memory etc. Chapter 04 covers process management in android, it covers how android process management is differ from linux process management. Finally Chapter 05 describes different set of problems in OOM killer and low memory killer.

Chapter 2

Linux Memory Management

Memory is a very limited resource especially in embedded systems like phones, tablets etc. So we need a good memory management system, it should provide better mechanisms for allocating and deallocating the memory. Memory management system should offer low memory fragmentation and less time for allocating/deallocating memory. This chapter covers memory addressing in linux in both segmentation and paging. It covers how the memory allocation and deallocation happens in linux. And also covers page frame reclaiming procedure in linux.

2.1 Memory Addressing

In general linux offers three types of addresses. MMU(memory management unit) transfers address from one format to other format. Fig 2.1 represents how the memory translation takes place from one format to other format. Fig 2.1 represents the implementation of segmentation on paging.

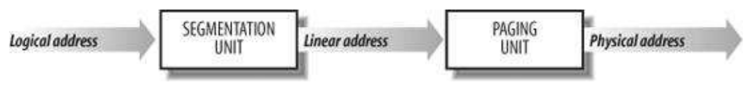


Figure 2.1: Memory addressing in Linux
[5]

1. **Logical address:** It consist of both segment and offset. Offset represents the location of data item with respect to segment base address.

2. **Linear address:** It is also called as virtual address space. In general process address space is represented in virtual address space. Every process can access with in the range of their virtual address space. Every process contains two address spaces one is user address space (0X00000000 to 0X000c0000) and another one is kernel address space(0x000c0000 to 0xffffffff).

3. **Physical address:** It represents the address of physical RAM. If a system contains 32-bit physical address, then the range of address between 0 to 2^{32}.

2.1.1 Segmentation

Linux prefers paging than segmentation. Because segmentation and paging are somewhat redundant. RISC architectures in particular have limited support for segmentation. Many of the phones, tablets are manufactured using RISC architecture. Linux 2.6 uses segmentation only when required by the 80x86 architecture[5].

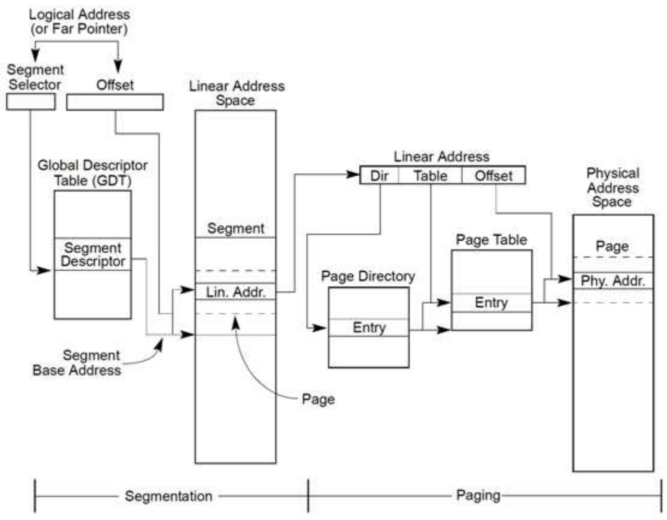


Figure 2.2: Segmentation on paging in Linux
[5]

Four Main Linux Segments. Every process in Linux has these 4 segments(user code, user data, kernel code, kernel data). Figure 2.2 depicts how the segmentation on paging happens in linux. It clearly shows how the conversion of logical to linear, linear to physical address translation takes place. Global descriptor table holds the details about each segment starting address and it's length.

2.1.2 Paging

Linux have 4-level paging from 2.6.11 version onwards for both 32-bit and 64-bit architectures. For 32-bit architecture two-level paging is sufficient to support two level paging linux page folding mechanism. In page folding mechanism linux simply eliminates upper and middle levels by making their sizes as zero. For 64-bit architecture linux uses four level paging.

Fig 2.3 shows the structure of four level paging in linux. The address of page global directory is stored inside cr3(control) register[5]. The following example gives how page folding mechanism works in linux.

1. 32- bit architecture page table with page size 4KB and using page folding method.

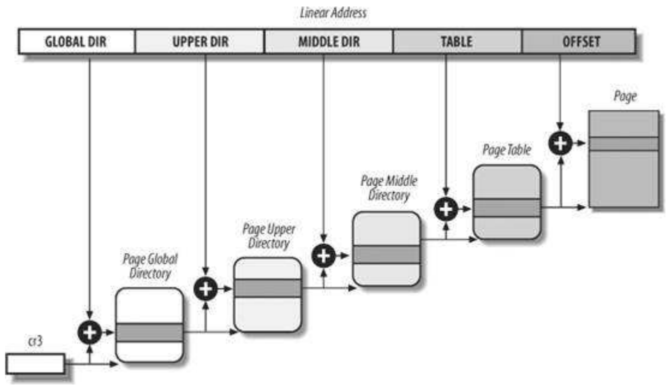


Figure 2.3: Page table in Linux
[5]

- Page Global Directory (10 bits)
- Page Upper Directory 0 bits; 1 entry
- Page Middle Directory 0 bits; 1 entry
- Page Table(10 bits)
- Page offset (12 bits)

2. Some of the methods to access data from page table are given below:

   - pte_read( ) to read the contents of page table entry
   - pte_write( ) to write the contents to page table entry
   - pte_dirty( ) to check page is dirty or not

2.2 Memory Management

This section mainly focus on physical RAM management. All the physical RAM classified into frames. Memory management unit dynamically allocate frames to process. In general linux stores BIOS information on frame-0. It allocates portion of memory for hardware devices. And a portion of memory is allocated for kernel static code data structures. Fig 2.4 represents how the physical RAM is allocated for hardware devices and kernel code.

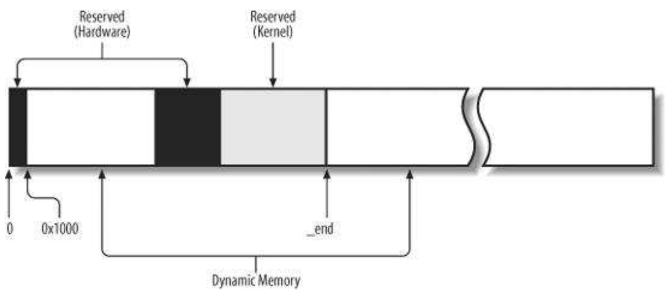


Figure 2.4: Dynamic Memory in RAM
[5]

Page Descriptor structure holds the details of each physical page of memory. This structure contains page flags like locked, dirty, accessed, active/inactive, being reclaimed, etc. And also it contains reference counter to each frame to check how many page tables are accessing this frame.

2.2.1 Zones

In general linux classifies RAM into three zones named as ZONE_DMA ($\leq$ 16 MB), ZONE_NORMAL (16MB-896MB), ZONE_HIGH($\geq$ 896MB). If DMA does not have any hardware constraints, then ZONE_DMA is optional. If RAM size is less than 896MB, then ZONE_HIGH is also optional. Kernel address space is directly mapped into NORMAL_ZONE. Fig 2.5 shows the classification of RAM into zones.

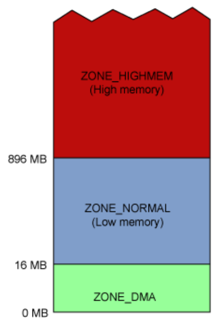


Figure 2.5: Ram classification in Linux

1. **Why we need ZONES:** Zones are introduced due to hardware limitations in some architectures. In some architectures DMA hardware chip can access only 0-16MB address range of RAM. Number of zones in a system is dependent on hardware. If DMA hardware can access any part of RAM, then we don't need DMA_ZONE.

2. **Zone Allocator:** Any memory allocation request sent to zone allocator. It find out the memory zone which satisfies the number of frames required. If no free frames are found then it rises signal to page frame reclaiming algorithm. After selecting the zone it calls buddy allocation algorithm. Fig 2.6 depicts that how the zone allocator runs in each zone. Each per-CPU cache includes some pre-allocated page frames to be used for single memory requests issued by the local CPU. Zone allocator also responsible for freeing memory[5].

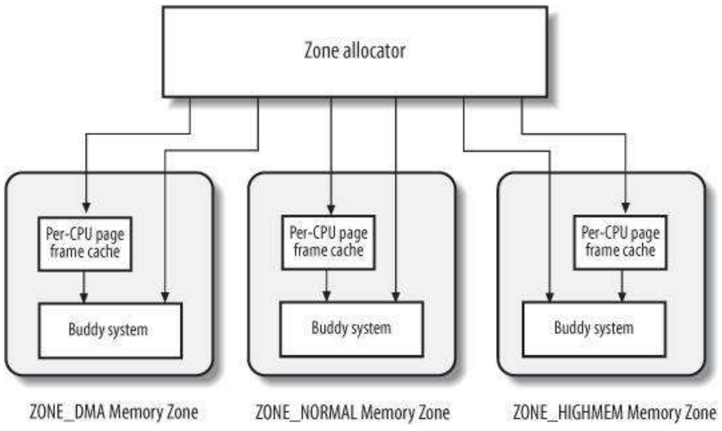


Figure 2.6: Zone allocator in Linux
[5]

2.3 Memory Allocation and Deallocation

Memory management unit could handle allocation and deallocation of memory requests in short time. It also consider the wastage of memory should be very less. Linux have two kinds of memory allocators one is buddy algorithm and another one slab algorithm. In general slab allocator designed to avoid fragmentation with in frames. And slab allocator mainly used for allocating data for kernel data structures[10]. Fig 2.7 shows the overall memory allocation and deallocation process. Fig 2.7 clearly depicts the memory allocation for kernel mode and user mode process.

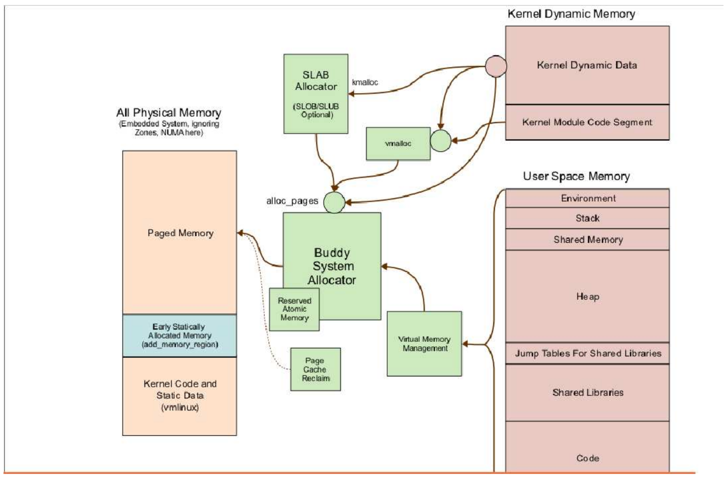

Kernel Dynamic Memory
Kernel Dynamic Data
Kernel Module Code Segment
SLAB Allocator (SLOB/SLUB Optional)
kmalloc
vmalloc
All Physical Memory (Embedded System, ignoring Zones, NUMA here)
Paged Memory
Early Statically Allocated Memory (add_memory_region)
Kernel Code and Static Data (vmlinux)
alloc_pages
Buddy System Allocator
Reserved Atomic Memory
Page Cache Reclaim
Virtual Memory Management
User Space Memory
Environment
Stack
Shared Memory
Heap
Jump Tables For Shared Libraries
Shared Libraries
Code

Figure 2.7: Memory Allocation & deallocation in Linux
[10]

1. **When does new memory get allocated**
    - Process stack grows
    - Process creates (attaches) to shared memory segment(shmat())

- Process expands heap (malloc())
- New process gets created (fork())
- Map a file to memory (mmap())

2. **Allocation at different levels**

   (a) Allocation at pages/frames using alloc_pages() and __get_free_pages(). These requests are handled by Buddy system.

   (b) Allocation at byte level using kmalloc(size, gfp_mask) and vmalloc(size, gfp_mask). Here kmalloc() request for physically contiguous bytes and vmalloc() requests for logically contiguous bytes. These requests are handled by slab allocator.

   (c) Allocation at user level using malloc()

3. **Some of the memory request methods**

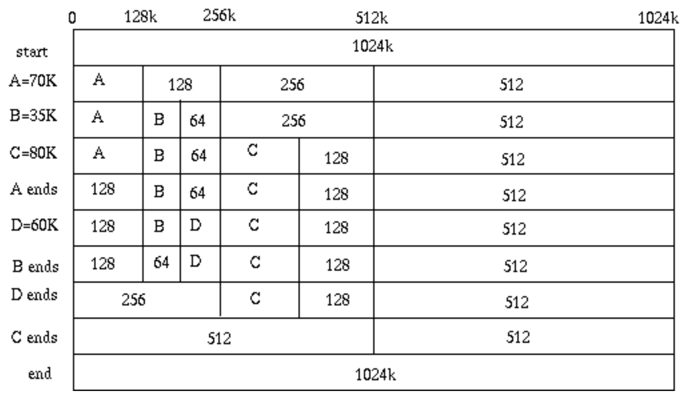
   - alloc_pages(gfp_mask, order), requests for a group of pages of size 2^{order}
   - alloc_page(gfp_mask), requests for a single page
   - _ _get_free_pages(gfp_mask, order)
   - _ _get_dma_pages(gfp_mask, order), get a pool of pages from DMA zone
   - get_zeroed_page(gfp_mask), get a page whose contents are all zeros

4. _ _free_pages( ) is used for deallocating memory.

2.3.1 Buddy Algorithm

It takes less time to allocate and deallocate, it provides less external fragmentation. It maintains different free lists with different sizes in each zone.

Ex: 4,8,16,32,64 group of free lists. Free list 4 contains a set of free blocks(buddies) whose size is 4 pages. Block size should be power of 2.



start	1024k					
A=70K	A	128		256		512
B=35K	A	B	64	256		512
C=80K	A	B	64	C	128	512
A ends	128	B	64	C	128	512
D=60K	128	B	D	C	128	512
B ends	128	64	D	C	128	512
D ends	256			C	128	512
C ends	512					512
end	1024k					

Figure 2.8: Buddy memory allocation & deallocation
[9]

Fig 2.8 explains Buddy algorithm with an example. Initially Buddy algorithm treats entire memory as single unit. Whenever a memory request comes, it checks for smallest possible block (power of 2) and assigns. From fig 2.8 to allocate memory for request $A = 70K$, Buddy splits single block into multiple sub buddies till to get a small buddy which is just greater than memory request and power of 2. And whenever a memory deallocation request comes, it simply deallocates that buddy, and try to merge with sibling buddies to make a large block[9].

2.3.2 Slab Algorithm

Buddy algorithm works fine for large memory requests. To handle low memory (bytes) requests we need an efficient algorithm that supports less fragmentation and less time for initializing an object. It reduces internal fragmentation, inside pages. It works upon Buddy algorithm. Slab algorithm used for handling memory requests for small bytes[5].

Slab allocator creates a cache for each object of unique sizes. It maintains a list of caches for frequently accessed kernel data structures like inode, task_struct, mm_struct etc. Cache is a collection of slabs. Slab size may be one or two pages. Slab contains a group of objects of similar type. Fig 2.9 depicts the clear idea of slab allocator, it shows the relation among caches, slabs and objects.

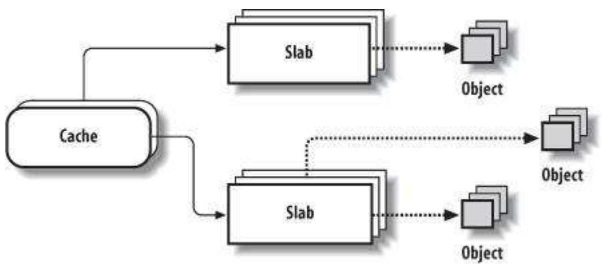


Figure 2.9: Slab allocation in Linux
[5]

1. **Slab Data structures**

    - Cache descriptor depicts the type of object could be cached. Slab maintains a list for all the caches.
    - Slab descriptor contains pointers to actual memory address of object. Every slab contains some state like full, partial, empty.
    - Object descriptor holds two things, first one object is free and holds a pointer to next free object, second holds contents of object.

2. **Methods for allocating and deallocating objects:** Objects are allocated using kmem_cache_alloc(cachep), where cachep points to the cache from which the object must be obtained. Objects are released using kmem_cache_free(cachep,objp).

2.4 Page Cache

Linux logically divides physical memory into two types. One is Page cache another one is anonymous memory. Linux allocates some pages to page cache from any zone. Page cache is nothing but cache for the disk files, and block devices[5]. It improves system performance by caching user mode data in main memory.

some of the silent features of page cache:

- Size of page cache is dynamic.
- It holds the contents of files etc.
- It provides sharing among different process.
- It uses copy-on-write mechanism.

2.4.1 Reading Pages from Disk

Whenever a process requests kernel to load a new file, page cache reads the file from disk. Page cache requests to the memory allocator for new pages, after the allocation of pages succeed it copies contents of file into pages. Finally using mmap() it maps corresponding physical address into process virtual address.

Some of the page cache functions:

- add_to_page_cache( ) inserts a new page in the page cache.
- remove_from_page_cache( ) function removes a page from the page cache
- read_cache_page( ) helps that the cache includes an up-to-date version of a given page

From linux kernel 2.6 onwards, pdflush kernel threads takes care about periodically scanning the page cache and flushing dirty pages to disk.

2.4.2 Anonymous Memory & Page Cache

All pages excluding page cache are anonymous memory. In another form the pages which are swappable to swap space are called anonymous memory. An anonymous page does not belongs to any file. It may be part of a programs data area or stack[8].

Fig 2.10 depicts an example for how page cache shares data among multiple process. In fig 2.10 two process named as render, 3drender both are sharing scene.dat file from page cache. Step2 & step3 in fig 2.10 explains how the concept of copy-on-write mechanism works.

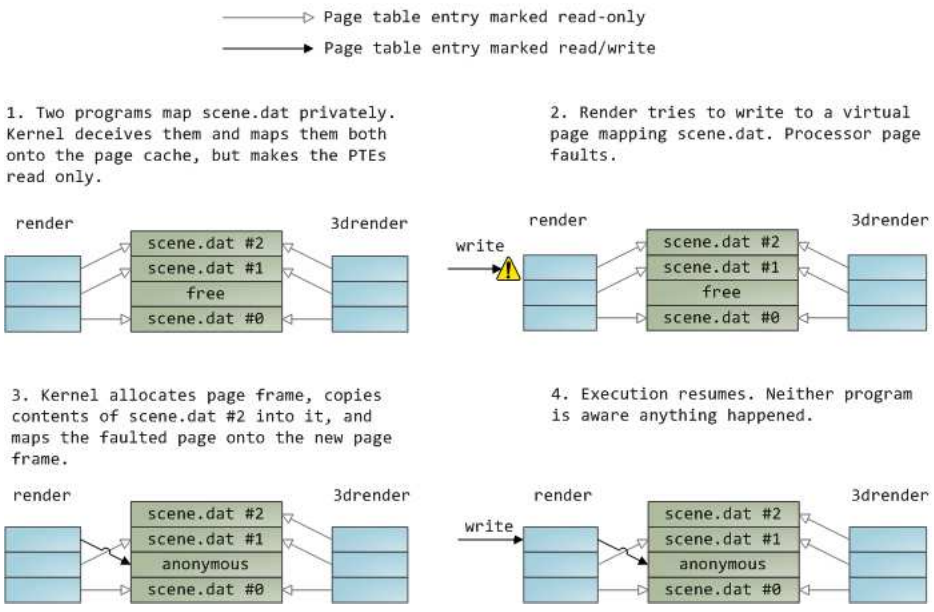

Page table entry marked read-only
Page table entry marked read/write
1. Two programs map scene.dat privately. Kernel deceives them and maps them both onto the page cache, but makes the PTEs read only.
2. Render tries to write to a virtual page mapping scene.dat. Processor page faults.
3. Kernel allocates page frame, copies contents of scene.dat #2 into it, and maps the faulted page onto the new page frame.
4. Execution resumes. Neither program is aware anything happened.
render
3drender
write
scene.dat #2
scene.dat #1
free
anonymous
scene.dat #0

Figure 2.10: page cache in Linux
[8]

2.5 Page Frame Reclaiming Algorithm(PFRA)

PFRA plays vital role in memory management, to reclaim free pages. PFRA works in three modes one is in low memory scenarios, second is hibernation mode and final one is periodic reclaiming. Table 2.1 shows that what are different kinds of pages present inside memory. Some of the pages are reclaimable and some are not reclaimable.

Type	*Description*
Unreclaimable	Kernel code and stack, locked pages,free pages (free_area in Buddy system)
swappable	anonymous memory
Syncable	user mode data(files)
Discardble	unused pages (process unused pages)

Table 2.1: Types of pages

2.5.1 When to Reclaim Memory

From fig 2.11, we can identify the situations when reclaim of memory will be done. And also fig 2.11 clearly draws the sequence of system calls, for reclaiming pages in all three modes (low memory, hibernation, periodic)[5].

- **Low on memory:** If there is no free and no reclaimable memory, the system is in trouble. So now we have to identify process which to be killed. Out_Of_memory()

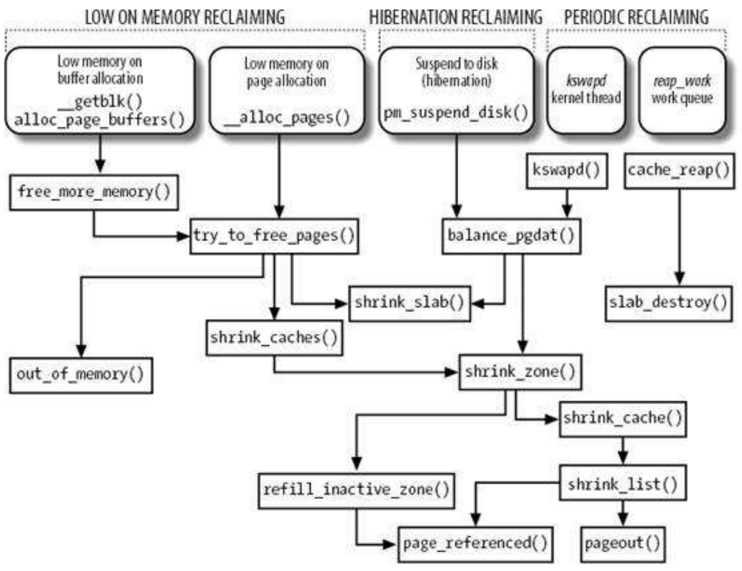


Figure 2.11: Frame Reclaiming system calls in Linux
[5]

functions starts to kill some of the process, based upon heuristics, more details about OOM(out of memory) described in next chapter.

- **Periodic Reclaiming:** A kernel thread kswapd runs periodically to reclaim memory. Kswapd traverses each process and reclaims pages until free memory size > some threshold. After reclaiming 32 pages, kswapd yields the CPU and calls the scheduler, to let other processes run. It checks the free memory in each zone, if free memory in any zone less than pages_min, than it reclaims pages until to reach pages_high number of free pages.

Each process maintains two lists(Active, Inactive). Active list holds frequently used pages. Inactive list holds less frequently used pages. In linux if a page is accessed by a process two times then it is placed in active list. If a page is not accessed by any page from a long time, then it is moved to inactive list.

Chapter 3

Linux vs Android as Memory Management

Even though android uses linux kernel, but there are some bit differences in android, linux memory management units. Android designers modified some of the drivers like ASHMEM (android shared memory, PMEM process memory). Android introduced a new driver called low memory killer to handle low memory scenarios effectively. This chapter covers what are the differences among linux and android with respect to different drivers. One of the major difference between android and linux is whenever a process gets terminated in linux, all details about that process would be removed from main memory. But in case of android even though a process terminated, contents of that process are not removed from main memory. Because to reduce response time, when the user access same process in next time. These kind of processes are called "Empty processes".

3.1 Out Of Memory (OOM) Killer in Linux

In low memory scenarios we have to kill a bad process or leave the system to be crash. OOM would be called when the system is full of multiple large memory consumed applications (or) if applications running in a system have memory leaks. Most of the times OOM killer was called when the virtual memory of all the process is > (physical memory + swap) . In low memory situations, if page frame reclaiming algorithm fails to free free_pages > some threshold, then it calls OOM[12]. Most of the times OOM killer tries to kill one process.
Order of system calls:

- _alloc_pages() − > try_to_free_pages().
- In low memory situations, if try_to_free_pages() function fails to free the pages then it calls out_of_memory()

3.1.1 How Does OOM Kills Processes

OOM calculates score for each process using badness(), and remove the process which have highest scores. Sequence of system calls are out_of_memory() − > select_bad_process()

$->$ oom_badness(). OOM killer calculates score for each process based upon heuristics.

Some of the heuristics are given below:

1. High memory occupied process gets high score
2. Long running process gets low score
3. Process which have large no of child process, to be killed.
4. Do not kill super users process

3.1.2 OOM Killer Implementation Details

OOM receives request to free a set of pages of order (2^n)[17].

- **select_bad_process()**
  This function traverses through all the running processes and calculates score for each process, using some heuristics. Finally it returns the pid of bad process, which have highest score. Internally it calls oom_badness() function.
  oom_badness() working procedure:

  1. It calls oom_unkillable_task() to check whether this process can be killable or not. OOM killer does not kill some kernel and root processes. If the process is unkillable this function returns 1 otherwise 0.
  2. If the process is killable then it calculates score for this process.
     score= size of process in RAM + size of process in Swap + process page table size.
  3. If the process belongs to root, then it gets some(3%) reduction in score. And finally it returns total score.

  After traversing all the process, select_bad_process() returns pid of bad process. Now if is there any process found to be killed then out_of_memory() calls oom_kill_process().

- **oom_kill_process()**
  This function traverses through all the threads of killable process and calls oom_badness() for all threads. And it tries to kill the threads with highest score. This function helps to minimize loss of work through killing threads instead of killing entire process.

More details about OOM killer will be discussed in later chapter 5. In chapter 5 some of the problems in OOM killer are explained.

3.2 Low Memory Killer in Android

Android have another new killer along with OOM killer to avoid some of the problems in OOM killer. OOM killer does not give any priority to frequently accessed applications. And priority value of a process is static during the lifetime of a process. Android main memory contains so many empty(terminated) process, so order of killing should be different unlike OOM killer.

3.2.1 How it Differ from OOM

Low Memory Killer of the Android behaves a bit different against OOM killer.It classifies process into different groups. In low memory situation it starts killing the process from low priority groups[14].
List of different groups:

- Foreground(active), the application which user currently focused.
- visible process which is not in foreground but bounded to foreground process.
- Service Process which is running on background like playing music etc.
- Hidden Process which is not visible.
- Content Provider which provides structural data like calender, contacts.
- Empty processes which were already terminated. But they are still present in main memory.

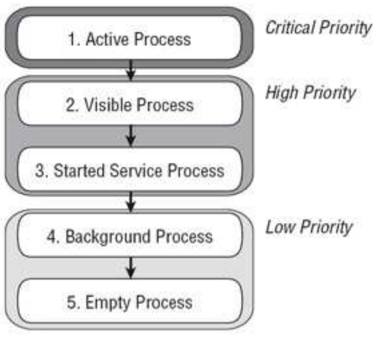


Figure 3.1: Process groups in android
[14]

More details on how a process can be moved from one group to other group, would be explained in chapter 04. Figure 3.1 depicts order of importance of groups, order of priority decreases from foreground(active) to empty.

3.2.2 How does Low Memory Killer Works

It stores oom_adj (out of memory adjust) values for each group of processes, oom_adj value ranges from -17 to 16. It also stores memory(minfree) thresholds for each group of processes at which this group processes gets killed. All these oom_adj and minfree thresholds are stored in init.rc file. Kernel reads these minfree, oom_adj values from init.rc during booting time[13].

From table 3.1, low memory killer starts killing processes whose oom_adj value > 15 when free memory falls down less than 32MB (8192*4KB). Similarly low memory killer

GroupName	*oom_adjvalue*	*Memory(minfree)Threshold*
Foreground Processes	0	2048
visible Processes	1	3072
Secondary Processes	2	4096
Hidden Processes	7	6144
Content Provider Processes	14	7168
Empty Processes	15	8192

Table 3.1: Order of killing process

starts killing processes whose oom_adj value > 14 when free memory falls down less than 28MB (7168*4KB) like that for all. Memory thresholds are varies based on RAM size. The values in table 3.1 are taken from a android OS mobile,with RAM size as 512MB. These static minfree values causes to many problems regarding responsive time of a process and battery life time. More details on these problems explained in chapter 05.

3.3 POSIX shared memory in Linux

From linux kernel version 2.6 onwards linux uses POSIX shared memory. It provides set function to create, access shared memory among multiple processes.

Some of the functions in POSIX shared memory are given bellow.

1. fd = **shm_open**(myfile, O_CREAT | O_RDWR, 0666), this function creates a shared memory file in /dev/shm/myfile and returns file descriptor of that file.
2. **ftruncate**(fd, SHM_SIZE), it is used to set the shared memory file size.
3. **mmap**(0,SHM_SIZE, PROT_WRITE,MAP_SHARED, fd, 0), using mmap() function we can map the shared memory address into process virtual address space. PROT_WRITE(pages can be written) used for putting protection on shared memory. MAP_SHARED means updates to the region are visible to other process. The files which are placed in page cache have MAP_PRIVATE which means copy-on-write princple. Any number of processes can access shared memory through mmap().
4. **mlock**(addr, SHM_SIZE), it applies lock on given given address region. This mlock() function prevents the kernel to do not swap these pages into swap space. But kernel can reclaim these pages.
5. **shm_unlink**(myfile) this function helps to delete shared memory region. kernel maintains the reference count for shared memory, whenever count goes to zero kernel automatically deletes the shared memory region.

POSIX shared memory implementation not suitable for embedded devices. Because POSIX does not support for reclaiming pages in low memory scenarios.

3.4 Ashmem - Android shared memory

Android shared memory sub system is a little bit different from POSIX shared memory. Ashmem introduced the concept of pining and unpinning of concept to handle low memory scenarios. Pinned pages of shared memory can not be reclaimable in memory pressure, but unpinned pages can be reclaimable. It is suitable for low memory devices, because it can reclaim unpinned shared memory pages under memory pressure[13].

1. **shmem(Linux) vs ashmem(Android)**

   - Ashmem classifies shared memory pages into two types.
   - One is pinned pages. another one is unpinned pages.
   - In low memory situations kernel can evict unpinned pages.

   Ashmem is implemented in mm/ashmem.c. It contains two major structures to support shared memory. One structure is ashmem_area, it holds list of shared memories and their sizes, protection information. Another structure is ashmem_range, it holds list of unpinned pages of all shared memory regions.

2. **Creating a shred memory region**

```
        fd = ashmem_create_region("my_shm_region", size);
        if(fd < 0)
            return -1;
        data=mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, 0);
        if(data == MAP_FAILED)
                goto out;
```

   If another processes wants to access shared memory, it does not access through file name. Because of security issues. Process share the fd of shared memory through IPC binder.

3. **Some of the functions available in android shared memory**

   (a) **ashmem_create_region**("myfile",size) this function creates shared memory file in /dev/ashmem/myfile

   (b) **ashmem_set_prot_region**(fd,prot) It helps to set protection mechanism on shared memory.

   (c) **ashmem_pin_region**(fd, offset, length) this helps to process protect the pages from kernel to reclaim in low memory.

   (d) **ashmem_unpin_region**(fd, offset, length) this function helps tell to android shared memory subsystem to reclaim these pages in low memory scenarios. A process can request for pining on already unpinned pages to android shared memory sub system.

Android shared memory varies with linux shared memory in only pining and unpinning concept only. Linux provides mlock() to get lock on virtual address of a process, but it only protects that pages could not be swapped into swap space. In android user can develop an applications such that it offers some of the unused pages to kernel to reclaim on low memory pressure. In memory pressure situations kernel requests shared memory subsystem to release unpinned pages. Then shared memory subsystem releases pages least recently used pages.

3.5 PMEM (process memory allocator)

pmem is used to manage large (1-16+MB) physically contiguous regions of memory shared between user space and kernel drivers (dsp, gpu,etc). Due to some hardware limitations in devices like MSM7201A, they need continuous memory. Ashmem and pmem are very similar, both are used for sharing memory between processes. Ashmem uses virtual memory, whereas pmem uses physically contiguous memory[13].

Ashmem maintains reference count for each shared region, it represents how many process are currently accessing shared memory region. If reference count is zero, then no process accessing that shared memory region. Pmem does not work in above manner, because it needs to maintain a physical to virtual mapping. The process which creates shared memory through pmem should hold file descriptor until all the references are closed.

User can Choose ashmem or pmem depending upon the size of physically shared contiguous memory.

Chapter 4

Android Process Management

In android processes are grouped into multiple groups to handle low memory scenarios. This chapter covers what are the fundamental components of an applications. It also covers role of each component in a brief manner and life cycle of android process. The state of a processes depends upon the components of that process.

4.1 Application Fundamentals in Android

Android is multi-user OS, each application has it's own user id(because of security reasons). Each applications has it's own VM(virtual machine), to provide isolation from other process. Each applications has it's own process. Application is a combination of components. List of all components should be placed in manifest file. Every component plays different role in applications. List of components of an application are activity, services, Broadcast Drivers, Content Providers[15].

4.1.1 Activity

Activity is a single visual user interface(UI) on the screen. In general activity is the main component in an applications for interacting with user. Some of the examples for activities are reading/sending SMS, single snapshots in a game. An activity can occupy complete screen or it can share screen with multiple activities. An activity in one application can interact with activities in other applications. When a camera application takes a photo then it can start email application activity to send this image to someone else.

- **Activity Stack:** All the activities in a system are placed in stack. Whenever a new activity starts, i.e placed on top of stack. whenever user press back button, activity on the top of stack would be removed. An activity which is placed on top of stack is always belongs to foreground process and remaining activities belongs to either foreground or visible processes[16].

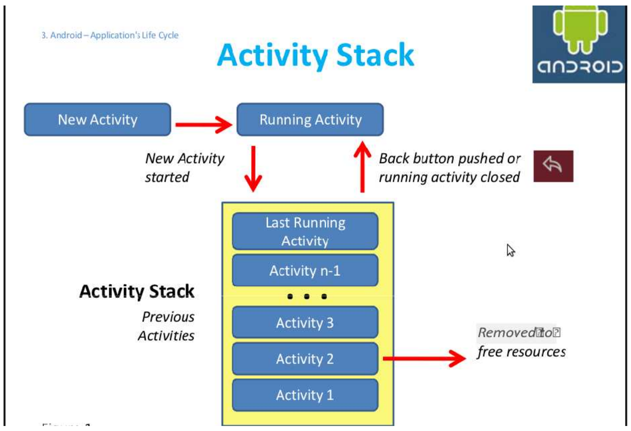


Figure 4.1: Activity Stack in android
[16]

Figure 4.1 represents the pictorial view of activity stack of android. Whenever a new activity starts it is placed on top of stack android if user wants to go back then top activity is popped. In memory pressure situations low memory killer can kill some of the processes, so the activities belongs to corresponding processes are removed from stack.

- **Activity Life Cycle:** Every activity have create and destroy states, in between these two states it will be fall into multiple states(running, pause, stop). If an activity have user focus (on screen) then it is in running state.

1. **onCreat()** is responsible for initialization of an object and also it restores the saved state of a activity.
2. **onStart()** after this method is called activity is visible to user.
3. **onResume()** is puts the activity into foreground, currently this activity is on top of the activity.
4. **onPause()** methods called whenever user creates a new activity, so it saves the state of current activity and then puts activity into pause state. But this activity still visible to user.
5. **onStop()** method is called before putting an activity state to non visible(hidden).
6. **onDestroy()** method causes to release resources of that activity. This is the final state of a process.

Activity class provides multiple methods to implement an activity. An application class can extends activity class to override methods like oncreat(), onstart() etc. Figure 4.2 shows complete life cycle of an activity. Every activity need not to be go through all states, because in memory pressure situations low memory killer can kill an activity.

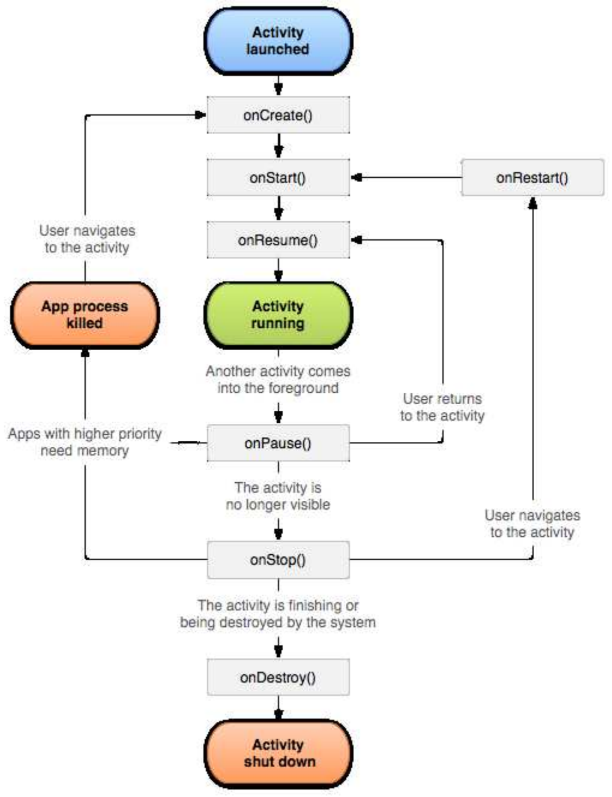

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 4.2: Activity life cycle in android
[15]

4.1.2 Services

Service is a component many times which runs on background, but we can start service in foreground also. Once an applications starts a service on background, even though user switches to other applications, it still runs in background. Some of the examples are playing music, downloading a file from Internet. Services do not provide any user interface.

Figure 4.3 depicts the life cycle of a service. Service total life time lies between create and destroy methods. We can start service in two modes one is running on background using onStartCommand() or bound a service to another component using onBind() method.

4.1.3 Content Providers

It provides a data for different applications. Content providers support to store data in a structural format in a file or database(sqlite). Other applications can access the data provided by content providers through content resolver. Content resolver is an interface for client applications to access data provided by content providers. Some of the examples for contact information, calender information and audio, video information also.

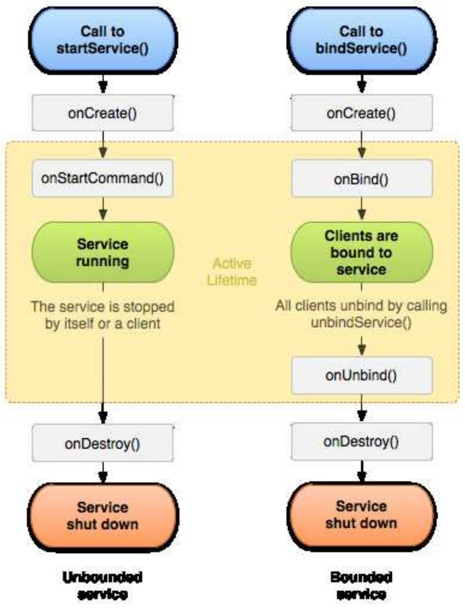


Figure 4.3: Services life cycle in android
[15]

4.1.4 Broadcast Receiver

This component used for sending broadcast announcements in system-wide. An applications can initiate broadcast message to other applications like battery is low so try to close unnecessary events or file download completed etc. It does not have any user interface but it generates a notification messages on status bar.

Each application contains a manifest file, it holds list of components available in an application. And manifest file also contains permission for each component.

4.2 Process Life Cycle in Android

For each applications android creates a new process with single thread of execution(i.e main thread). By default all the components run in same thread(main thread). If not developer explicitly creates a new thread. In low memory situations a process is decide to kill based upon the number of inactive components of that process. A process which have more number of inactive components should be killed first.

Processes in android have five states like foreground, visible, service, background, and empty. The process current state depends upon the states of all components present in that process[15].

4.2.1 Foreground Process

A process is called as foreground if any of it components are in running(active) state. In another manner the process which have user focus currently is called foreground process.

A process is treated as foreground if any of components of process holds following states.

- An activity which is interacting with user. When an activity calls methods like onresume().
- When a service is interacting with foreground activity.
- When a service is started using startForeground().

4.2.2 Visible Process

If a process is not belongs to foreground but it is still interacting with user indirectly is called visible process. In another manner the foreground process covers partial screen, so another process is visible to user.

A process is treated as visible if any of components of process holds following states.

- An activity which is not interacting with user. But it is still visible to user, when an activity running method like onPause().
- When a service is interacting with visible activity.

4.2.3 Service Process

If a process does not fall in any of foreground, visible states and it have some active services which are running in background is called service process. In another manner the process which does not have any active components and it have services which are running in background like playing music etc.

A process is treated as Service if any of components of process holds following states.

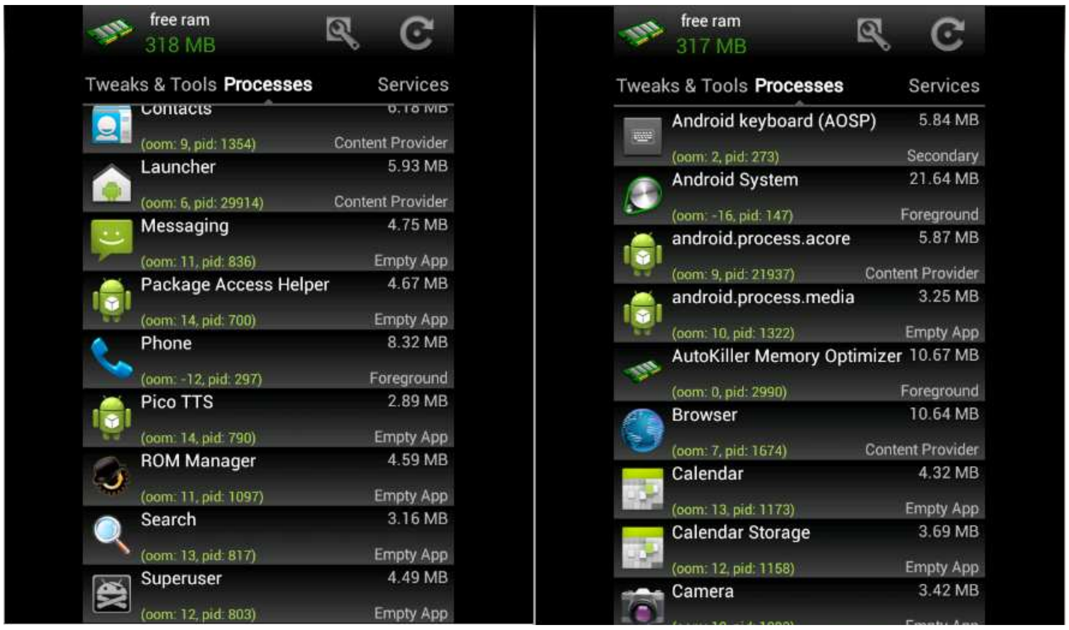
- When a service is started using startService().
- Eg: playing music in background, downloading file from Internet.

4.2.4 Background Process

It is also called as hidden process. The processes which are not visible but still alive are called background processes. A process is treated as background if any of components of process holds following states, when an activity running method like onStop(), i.e currently user is not interacting with that activity. System maintains LRU list of background process. Whenever a process decided to kill, it stores the state of activity. So whenever next user wants that activity, we can restore the previous state of activity.

4.2.5 Empty Process

A process is treated as Service if any of components of process holds following state when a process does not have any active component. Caching of empty process inside memory, reduces relaunching of applications once again if user wants that applications in future. Some of the examples for empty applications are phone call app, messages app, and settings app which are frequently used by user.



free ram
318 MB

Tweaks & Tools **Processes** Services

Contacts	6.18 MB	
(oom: 9, pid: 1354)		Content Provider
Launcher	5.93 MB	
(oom: 6, pid: 29914)		Content Provider
Messaging	4.75 MB	
(oom: 11, pid: 836)		Empty App
Package Access Helper	4.67 MB	
(oom: 14, pid: 700)		Empty App
Phone	8.32 MB	
(oom: -12, pid: 297)		Foreground
Pico TTS	2.89 MB	
(oom: 14, pid: 790)		Empty App
ROM Manager	4.59 MB	
(oom: 11, pid: 1097)		Empty App
Search	3.16 MB	
(oom: 13, pid: 817)		Empty App
Superuser	4.49 MB	
(oom: 12, pid: 803)		Empty App

free ram
317 MB

Tweaks & Tools **Processes** Services

Android keyboard (AOSP)	5.84 MB	
(oom: 2, pid: 273)		Secondary
Android System	21.64 MB	
(oom: -16, pid: 147)		Foreground
android.process.acore	5.87 MB	
(oom: 9, pid: 21937)		Content Provider
android.process.media	3.25 MB	
(oom: 10, pid: 1322)		Empty App
AutoKiller Memory Optimizer	10.67 MB	
(oom: 0, pid: 2990)		Foreground
Browser	10.64 MB	
(oom: 7, pid: 1674)		Content Provider
Calendar	4.32 MB	
(oom: 13, pid: 1173)		Empty App
Calendar Storage	3.69 MB	
(oom: 12, pid: 1158)		Empty App
Camera	3.42 MB	
[illegible]		Empty App

Figure 4.4: Applications states in android

Autokiller memory optimizer application helps to display states of currently running processes. For knowing the states of processes, So i installed this application in android SDK emulator. Figure 4.4 shows the memory footprint, oom_adj, state of different applications in a particular situation. This figure is captured through Autokiller memory optimizer third party application on android SDK emulator(android 4.0 version, RAM size 512MB). This figure shows different kinds of processes(foreground, empty, content providers) with minfree parameters of low memory killer are 20MB, 24MB, 32MB, 40MB, 44MB, 56MB.

Activity Manager in the android is responsible for maintaining life cycle of processes, adjusting oom_adj value of a processes, permisssions and other tasks. Activity manager runs in android framework layer. AndroidManagerservice.java file holds all the implementation details of how to adjust oom_adj value of each process. computeOomAdjLocked(), updateOomAdjLocked() are holds the code for updating oo_adj value of a process based upon the process currently state. In computeOomAdjLocked() method computes the oom_adj value based upon the states of activities, services and other components[19].

Chapter 5

Low Memory Problems in Linux, Android

Low memory problems in linux, android causes to degrade the system performance, if we do not handle them in proper manner. This chapter gives experimental analysis of OOM killer and more details on low memory killer working. This chapter describes different set of problems in both OOM, low memory killer. It also contains proposed solutions to handle problems in both OOM, low memory killer.

5.1 Analysis of OOM Killer in Linux

Out of memory killer in linux kills the process in memory pressure situations. Some of the details on OOM killer are already explained in section 3.1. This section gives details about behavior of OOM killer in memory pressure situations. This section proves some of the flaws in OOM killer through experimentation. All the below experimentation work done on kernel version 3.0.0-12. This entire experimentation done on system with memory configuration as 4GB ram + 8GB swap space. In general OOM killer was called when memory occupied by all the processes exceeds total memory (4GB ram + 8GB swap) and it also depends upon two kernel parameters (overcommit_memory, overcommit_ratio).

5.1.1 How does OOM Kills Process

OOM killer traverses through each process and calculate score for each process using badness(), and kills the process which have highest score. The order of system calls are out_of_memory() $->$ select_bad_process() $->$ oom_badness(). Finally oom_badness() function calculates score for each process using some heuristics and kills the process which have highest score.

1. **some of the heuristics are given below**
   (a) High memory occupied process gets high score
   (b) Long running process gets low score
   (c) Process which have large number of child process, to be killed.

(d) Do not kill super users process

2. **Implementation Details of oom_badness()** This function receives a process, total number of pages to be free and returns the score for this process. The main objective of this function is to return score for a process based upon some set of heuristics[17].

```
oom_badness( process, totalpages )
  {
          // Check this process is killable or unkillable(kernel process)
          if(oom_unkillable_task(process))
             return 0;

          //get lock on process mm(memory) structure
          if(!find_lock_task_mm(process))
             return 0;

          /* Calculate the score of process based on rss(resident set size) + swap
          + page table size. All the below functions returns number
          of pages occupied by coresponding portions */
          score = get_mm_rss(p->mm) + p->mm->nr_ptes;
          score += get_mm_counter(p->mm, MM_SWAPENTS);

          //Performs some normalisation
          score *= 1000;
          score /= totalpages;
          task_unlock(process);

          /* oom_score_adj ranges from -1000 to +1000. This oom_score_adj value is
          calculated by kernel based upon the other heuristics like how long this
          processes is running, number of child processes running */
          score += p->signal->oom_score_adj;

          // returns the final score for this process
          if (score <= 0)
            return 1;
          return (score < 1000) ? score : 1000;
  }
```

5.1.2 Scenario 01

In this scenario, experimentation is done with nine 1GB memory consume processes, one 2GB memory consume process on a system with 12GB total memory(swap + main memory) to create low memory scenario. From figure 5.1 except process nine all are 1GB processes. All the processes are started at same time in seconds(varies in micro seconds). Figure 5.1 shows the memory footprint of all the processes before OOM killer was called. From figure 5.1 we can simply say process nine is the bad process because it occupied large memory as compared with other process. This same is scenario 10 times, process nine gets killed 9/10 times and process ten gets killed 1/10 times. The reason to kill process ten is that process started lately so it OOM killer gives priority to long running processes.

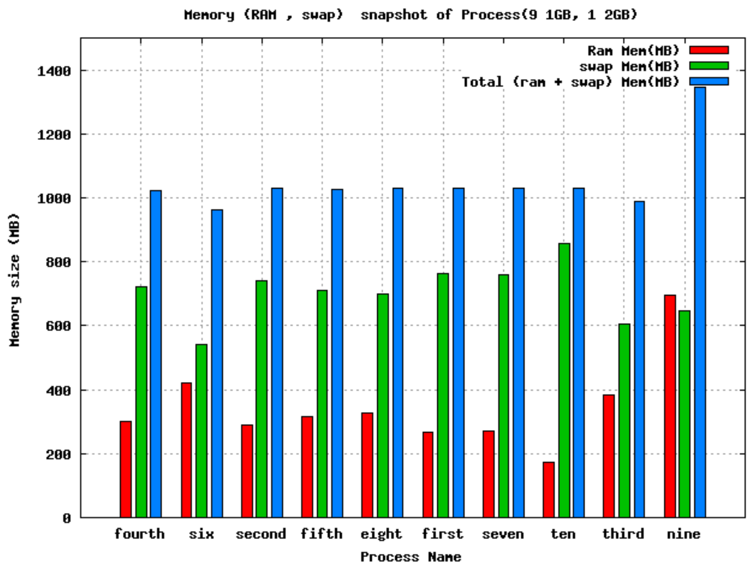


Figure 5.1: Processes(9 1GB, 1 2GB) footprint in low memory scenarios

5.1.3 Scenario 02

In this scenario, experimentation is done with nine 1GB memory consume processes, one 1136MB memory consume process on a system with 12GB total memory(swap + main memory) to create low memory scenario. From figure 5.2 except process nine all are 1GB processes. All the processes are started at same time in seconds(varies in micro seconds). Figure 5.2 shows the memory footprint of all the processes before OOM killer was called. From figure 5.2 we can not simply say which process is the bad process because all the processes occupied equal amount of memory. This scenario is repeated 5 times, process ten gets killed 4/5 times and process one gets killed 1/4 times.

According to my expectation all the time process nine gets killed but due to scheduler process nine got less memory before OOM killer was called. If we look into the figure 5.2 almost all the processes have same total memory and ram memory. So it is not better to kill every time process ten. Finally we can say there is no fairness in the order of killing the process, because single process(ten) gets killed many times, even though all the processes are almost equal in size.

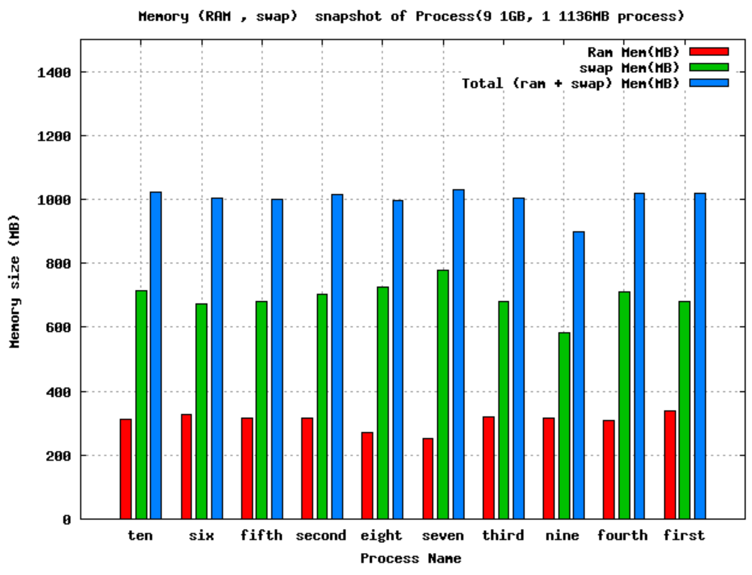


Figure 5.2: Processes(9 1GB, 1 1136MB) footprint in low memory scenarios

5.1.4 Scenario 03

In this scenario, experimentation is done with all equal size processes of size 1GB each. All the processes are started at same time in seconds(varies in micro seconds). Figure 5.3 shows the memory footprint of all the processes before OOM killer was called. From figure 5.3 we can not simply say which process is the bad process because all the processes occupied equal amount of memory. This scenario is repeated 5 times, process ten gets killed 5/5 times.

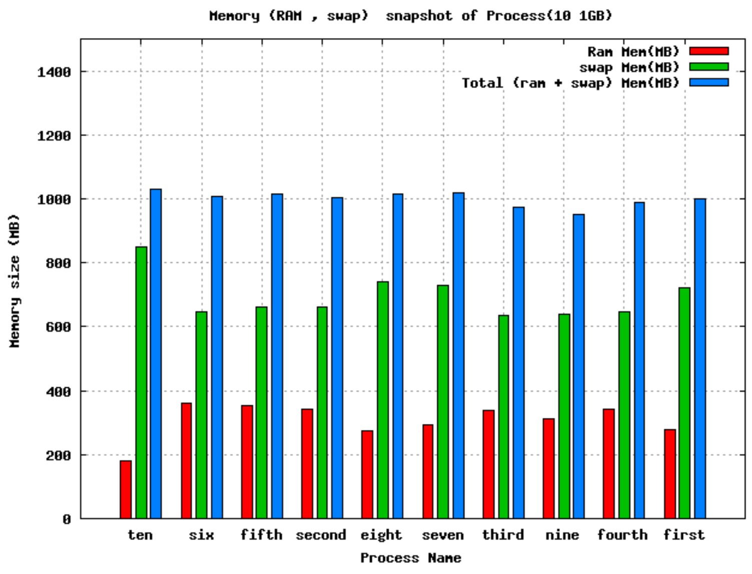


Figure 5.3: Processes(10 1GB) footprint in low memory scenarios

One good observation from figure 5.3 is process ten occupied less amount memory in RAM as compared with the other processes. So if we kill a process which occupied more amount of RAM rather than killing based upon total memory, it gives more free memory in RAM.

5.1.5 Processes Killed list

Figure 5.4 shows that how many times a process gets killed in all above three scenarios. In scenario 01 process nine gets killed 9/10 times and process ten gets killed 1/10 times. In scenario 02 process ten gets killed 4/5 times and process one gets killed 1/5 times. In scenario 03 process ten gets killed 5/5 times.

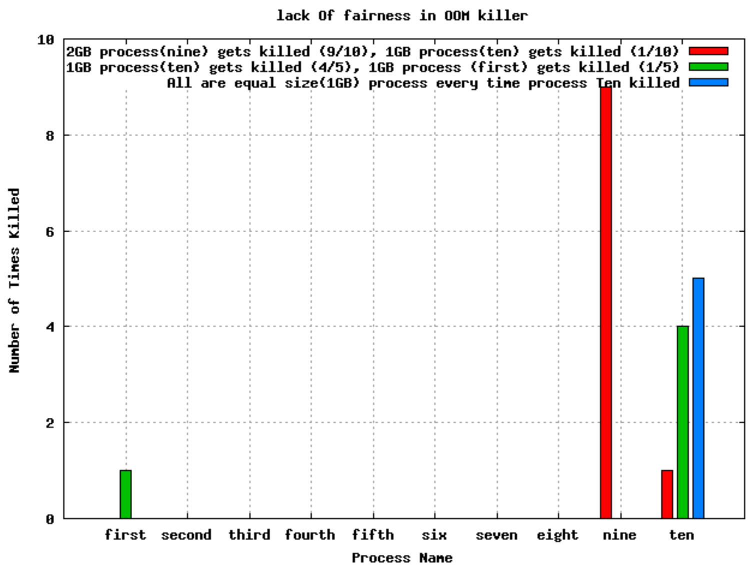


Figure 5.4: List of processes killed in low memory situations

5.1.6 Problems Identified in OOM Killer

1. **Problem 01**
   From section 5.1.2 we can identify there is no priority for frequently accessed applications. In case of embedded operating systems like android, we need to give priority to frequently accessed applications. Embedded operating systems demands more priority for frequently accessed applications. Finally OOM killer do not give any priority to frequently accessed applications. We can simply maintain frequently accessed applications list from log history. So if we give priority to the frequently accessed applications in low memory scenarios, we can achive better user experience.

   Solution for the above problem is maintain the list of frequently accessed applications. Here we are adding a new heuristic to the OOM killer such that it gives more priority to frequently accessed application. From table 5.1, if we maintain information about frequently accessed application, then we can implement this heuristic in OOM killer. Table 5.1 gives sample details, what kind of data we need to

Process_Name	*Number_of_Times_Accessed*	*Last_Access_Time*
contacts	20	10-12-12:3:12
browser	10	10-12-12:9:30
music player	8	11-12-12:11:10

Table 5.1: List of frequently accessed processes

implement priority for frequently accssed processes. If we implement frequently accessed list using heap data structure, we could improve performance for searching/adding/deleting processes in frequently accessed list.

2. **Problem 02**
From section 5.1.3 no fairness in the order of killing processes. Because from section 5.1.3 even though all the processes have almost same total memory and all the processes are started at same time in seconds, but every time same process gets killed. It proves that killing the processes based upon some heuristics leads to unfairness in order of killing.

Process_Name	*NumberOfTimesKilled*	*First_Time_Killed*	*First_Time_Killed*
Database App	10	10-12-12:3:12	13-12-12:9:45
3D game(wow)	8	10-12-12:9:30	15-12-12:4:23
HD video player	8	11-12-12:11:10	14-12-12:10:25

Table 5.2: List of processes killed by OOM killer

Solution for above problem is simply maintain log history of previously killed applications. If any application gets killed more than some threshold in a period of time then do not kill this application. If any process gets killed more than a threshold then it will not gets killed in future. If we maintain data like in table 5.2, we can solve this problem. Whenever OOM killer decides to kill a processes, it checks whether this processes gets killed many times or not. If yes then it searches for next bad process.

3. **Problem 03**
From section 5.1.4, we can identify OOM killer giving equal priority to all memories (RAM + SWAP + PTS) rather than giving wighted priority to all memories. From section 5.1.4 we can identify process ten gets killed every time even though it occupied less memory in RAM. So if OOM killer consider to kill a processes which occupies more space in RAM improves system performance.

Solution for the above problem is assign weights to RAM, swap, page table instead of giving equal priority to all memories. For this purpose we need to change code in oom_badness() function. Inside oom_badness()

- Existed: score = (sizeof(RSS) + sizeof(swap) + sizeof(PageTable))
- Proposed: score = (w1*sizeof(RSS) + w2*sizeof(swap) + w3*sizeof(PageTable))
  Let us take w1=0.75, w2=0.20, w3=0.05. We can figure out these weights through experimentation.

5.2 Compressed Cache

Android does not have any swap space, due to wear-leveling property of flash memory. In flash memory if we want to write some thing, then first we have to erase that block. Every block in flash memory can not be erased more than some threshold. Because of this problem android does not have any swap space.

But a group of people they are implemented swap space inside main memory using some compression/decompression methods. It swaps in pages through compression and swaps out pages through decompression. Compressed cache project separates some part of main memory for swap space. There is one tool(cynogenmod) which provides a chance to user to choose size of swap space in the main memory. It also provides the option to enable/disable compressed cache.

5.3 Analysis of Low Memory Killer

Low memory killer plays vital role in android for effectively handling memory pressure situations. This section talks about different kinds of problems in low memory killer. The speediness of phone, responsiveness of applications and degree of multi programing in android completely depends upon minfree thresholds as referred in table 3.1.

5.3.1 Low Minfree Thresholds

Let us take minfree values from table 3.1, (8MB, 12MB, 16MB, 24MB, 28MB, 32MB) in 512MB RAM system. So low memory killer do not kill any empty applications until free memory is less than 32MB, as a result there are more number of empty applications present in main memory. It increases degree of multi programing as a result, i.e more number of applications are available in main memory. It improves the responsiveness of applications, if user access those applications frequently.

Advantages	*Disadvantages*
Improves degree of multi programing	Lagging time increases
Improves responsive time for frequently accessed applications	More number of page faults

Table 5.3: Pros/Cons of low minfree thresholds in low memory killer

Problems with low minfree thresholds are increase in lagging time for launching big applications like 3D games, database applications, for playing HD videos and image processing applications. Because all these applications occupies large amount of main memory, so before launching these applications kernel should free memory through killing some applications thats why for launching big applications more lagging time is needed. Minfree thresholds also increases number of page faults for big applications, because free memory in system is very less.

5.3.2 High Minfree Thresholds

Let us take minfree values from table 3.1, (28MB, 32MB, 64MB, 156MB, 228MB, 256MB) in 512MB RAM system. So low memory killer do not kill any empty applications until free memory is less than 256MB, as a result there are less number of empty applications present in main memory. It decreases degree of multi programing as a result, i.e less number of applications are available in main memory. It reduces the responsiveness of applications, because of the aggressive nature of minfree thresholds most of the frequently accessed applications also gets killed. So whenever a user wants to access an applications, which is not available in main memory, then kernel need to launch it from flash memory it consumes more power and reduces responsiveness of applications. But one advantage with these minfree thresholds is lagging time for launching big applications is very small.

Advantages	*Disadvantages*
Lagging time is less	Decreases degree of multi programing
Less number of page faults	Increases responsive time for frequently accessed applications

Table 5.4: Pros/Cons of High minfree thresholds in low memory killer

5.3.3 Problems Identified in Low Memory Killer

Low memory killer should maintain moderate thresholds to solve problems discussed in above section. So there is a necessity for tunning minfree parameters of low memory killer to improve system performance.

1. **Problem 04**
   Most of the android users complained in android forums that the default minfree values in their system, do not giving good performance[20]. If they change the minfree values suitable to their mobile configuration, they are improving their mobiles speediness, responsiveness of applications. And same set of minfree parameters are not suitable for all kinds of users. Users(Engineers) who are using huge memory consume applications need one set of minfree parameters. And the users(school children) those use less memory consume applications need another set of minfree parameters. So there is a great necessity for setting default values for a system based upon type of user and hardware configuration.

   Solution for the above problem is develop a benchmark which run different kinds of applications with different combinations of minfree value. Finally figure out a suitable minfree values based upon the hardware configuration. Instead of checking with multiple combinations of minfree values, we can prune some of the high minfree, low minfree values. Beacuse from sections 5.3.1, 5.3.2 we can identify disadvantages in high, low minfree values.

2. **Problem 05**
   Static minfree values may not be suitable in all the situations, If we tune the minfree thresholds, based upon the user frequently accessed applications in a period of time, we can improve performance of system and user experience. For example if the user is using big applications like database, 3D games, other engineering tools which consume large main memory. In this situations if we tune minfree parameters (28MB, 32MB, 64MB, 96MB, 128MB, 156MB in a 512MB RAM system) with large values. Here a large amount of free memory is available, so lagging time for big applications is very less and less number of page faults will takes place. Another example if the user is using small applications like call, SMS, email, etc. In this situation if we tune minfree parameters (8MB, 12MB, 16MB, 24MB, 28MB, 32MB in a 512MB RAM system) with small values. Here kernel keeps so many number of empty applications inside memory, all the frequently accessed applications are available in memory. So responsiveness time for accessing small applications is very less.

   Based upon the user previous log history, we can solve above problem. If we design an application which periodically checks the frequently accessed applications and tunes the minfree values based upon type history applications.

3. **Problem 06**
   Compressed cache project provides swap space in android devices. More details on compressed cache are already discussed in section 5.2. Android users have the facility to enable or disable the swap space. But minfree parameters in low memory killer are always same in all modes (swap enabled, swap disabled). And also user have a facility to choose how much amount of RAM can be used as swap space. So if we tune the minfree parameters based upon current settings of swap space, then we can achieve better performance.

   To handle this problem, whenever user enables/disables swap space in general user need to be restart the system. So during the booting time low memory killer need to check whether swap space is enabled or not, based upon state of swap space it should select suitable minfree values. To solve above problem, we need to modify low memory killer such that to select the suitable minfree parameters based upon state of swap space.

Finally above problems need to be controlled to handle low memory scenarios. So we need to optimize OOM killer source code to handle problems discussed in section 5.1.6. Later we need to develop a tool which handles above problems like figure out suitable default minfree parameters, dynamically tunning minfree parameters based upon user frequently accessed applications and swap space. The objective of this tool is effective memory management, improve speediness of system and low power consumption.

Chapter 6

Conclusion

Based upon study of android memory management, there is necessity for modifying OOM killer source code for better user experience. If we modify OOM killer source code to give more priority to frequently accessed applications, fairness in order of killing the applications and choosing the process which occupies more main memory instead of giving priority to total memory gives better user experience. And there is need for dynamically tunning minfree parameters of low memory killer to handle low memory situations effectively. Finally if we modify OOM, low memory killer to solve problems described in chapter 05, we can achive better user experience and better overall performance.

Bibliography

[1] Frank Maker and Yu-Hsuan Chan,"A Survey on Android vs Linux" .*Department of Electrical and Computer Engineering, University of California, Davis*

[2] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. "A Flash-Memory Based File System".*Advanced Research laboratory, Hitachi, Ltd.*

[3] Kimmo Nikkanen,Turku Polytechnic Telecommunications. " UCLINUX AS AN EMBEDDED SOLUTION" ,.*Bachelor's Thesis* in 2003.

[4] [uClinux-dev] [Newbie] why use CPUs without MMU? Available at: *http://www.uclinux.org/pub/uClinux/archive/5762.html*

[5] Daniel P. Bovet, Marco Cesati . "Understanding the Linux Kernel, 3rd Edition" *Publisher: O'Reilly* , Pub Date: November 2005 , ISBN: 0-596-00565-2 , Pages:942.

[6] Robert Love , "Linux Kernel Development, 3rd Edition", *Publisher Addition-wesely* , ISBN-13: 978-0-672-32946-3 , Copyright 2010 Pearson Education,Inc.

[7] Abhishek Nayani,Mel Gorman & Rodrigo S. de Castro . "Memory Management in Linux". *Linux-2.4.19,Version 0.4*, 25 May 02.

[8] Gustavo Duarte,"Page Cache, the Affair Between Memory and Files ". Available at: *http://duartes.org/gustavo/blog/category/internals/*

[9] Buddy memory allocator. Available at: *http://acm.uva.es/p/v8/827.html*

[10] Xi Wang, Broadcom Corporation. "Controlling Memory Footprint at All Layers: Linux Kernel, Applications, Libraries, and Toolchain"

[11] "Process address space". Available at: *http://kernel.org/doc/gorman/html/understand/understand*

[12] OOM killer, linux memory management wiki. Available at: *http://linux-mm.org/OOM Killer*

[13] Android Kernel Features, elinux wiki. Available at: *http://elinux.org/Android Kernel Features*

[14] Memory Management in Android , welcome to mobile world. Available at: *http://mobworld.wordpress.com/2010/07/05/memory-management-in-android/*

[15] Processes and Threads, Android developers wiki. Available at: *http://developer.android.com/guide/components/processes-and-threads.html*

[16] Victor Matos, Cleveland State University. *"Android Applications Life Cycle"*. class notes part-03.

[17] TOMOYO Linux Cross Reference, Linux/mm/oom_kill.c. Available at: *http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/mm/oom_kill.c*

[18] Android Memory Analysis,Texus instruments wiki. Available at: *http://processors.wiki.ti.com/index.php/Android_Memory*

[19] Android Source Code, GrepCode wiki. Available at: *http://repository.grepcode.com/java/ext/com/google/android/android/4.1.1_r1/*

[20] How to configure Android's internal taskkiller, xdadevelopers wiki. Available at: *http://forum.xda-developers.com/showthread.php?t=622666*