Effective Handling of Low Memory Scenarios in Android using Logs

M.Tech Dissertation 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

June 20, 2013

Declaration

Rajesh Prodduturi
(113050076)

Place: IIT Bombay, Mumbai
Date: 23th June, 2013

Acknowledgments

I would like to thank my guide, Prof. Deepak B Phatak for his tremendous support and help. I would also like to thank Sir Nagesh, for his consistent encouragement and right directions, with which we are able to do this project work. I would also like to thank Pavan, Subbu for their inputs.

Abstract

Android memory management unit plays vital role in great success of android among other operating systems. Memory is a very limited resource especially in embedded systems like phones and tablets. Android contains modified Linux kernel for handling low memory scenarios and other issues. Out Of Memory (OOM) killer in linux kills some of the processes in low memory scenarios. We found out problems with OOM killer.Activity Manager Service(AMS), Low Memory Killer(LMK) in android kills some of the applications in low memory scenarios along with OOM killer. Once an application gets killed, if it needs to be accessed again, it should go through the memory loading cycle once again. In general, the memory loading cycle takes time in the order of 3 - 5 secs which is high. So, while selecting the applications to be killed in low memory scenarios, the applications that are being accessed frequently should be given very low priority. So, if we can figure out most frequently and recently used applications and protect them from getting killed. We can reduce number of applications gets killed in a period of time. We developed AppsLogger application, which predicts the user interesting applications using log history. We modified traditional AMS source code in android 4.2, such that AMS does not kill user interesting applications received from AppsLogger. Based on our experimental evaluation, we have seen our Log-Expo and Log-Linear approaches killed as low as 60%,72% lesser number of applications, while reducing the average response time by 24%,27% respectively, when compared to the traditional approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Embedded Operating System

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.
    - Binder Driver: It facilitates IPC(Inter process communication). Since data can be shared by multiple applications through the use of shared memory. Binder

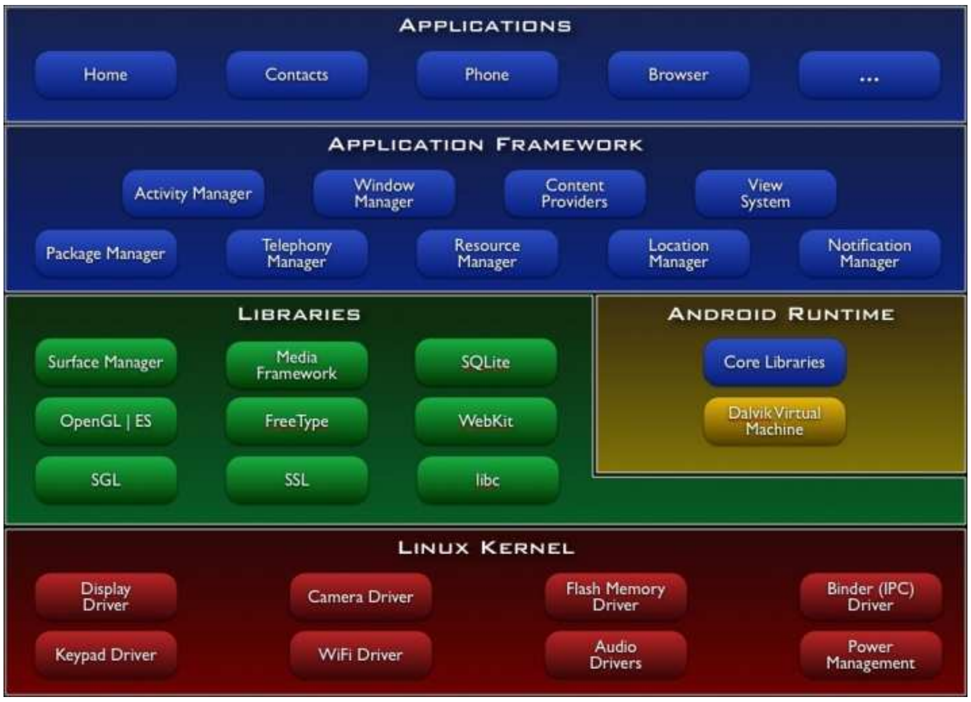


Figure 1.1: Android architecture
[1]

also takes care of synchronization among processes. IPC is being optimized, as less data has to be transferred

- 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

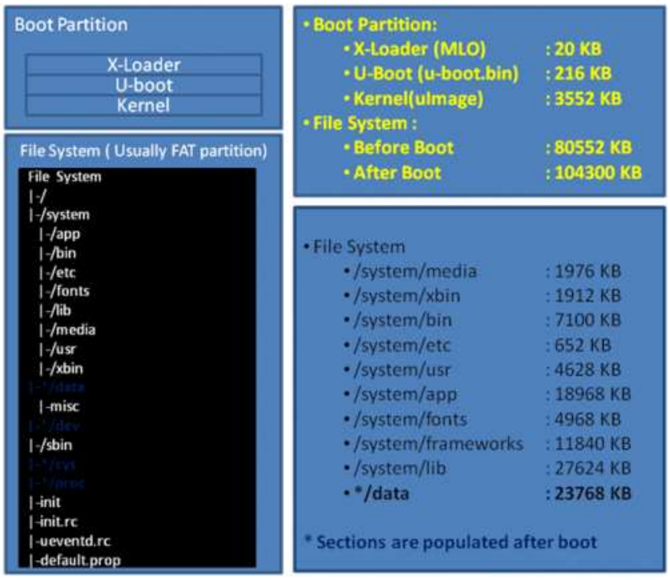
Table 1.1: YFFS File Structure

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 how much memory is needed for each component. 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.



```
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**

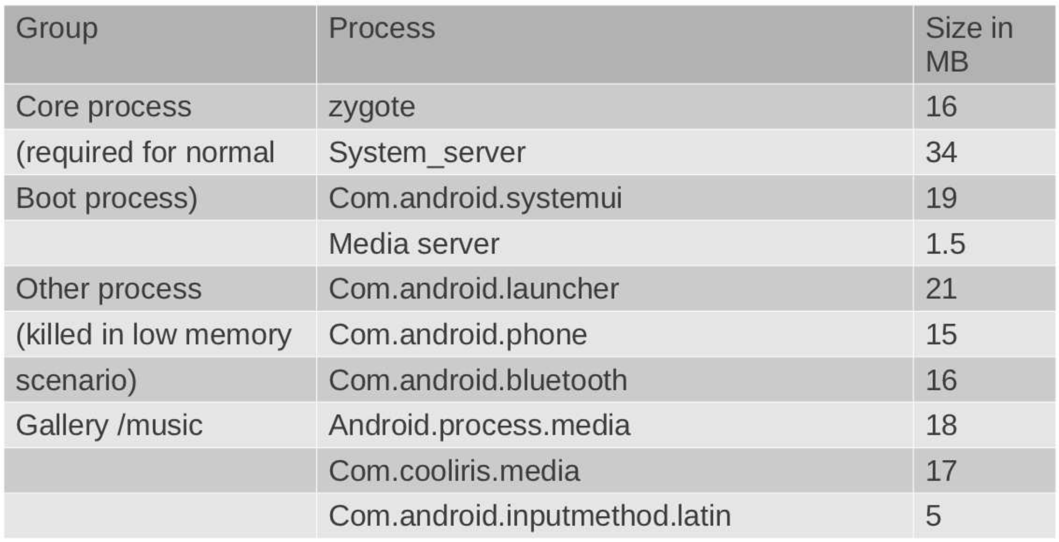
- File System
  - /system/media : 1976 KB
  - /system/xbin : 1912 KB
  - /system/bin : 7100 KB
  - /system/etc : 652 KB
  - /system/usr : 4628 KB
  - /system/app : 18968 KB
  - /system/fonts : 4968 KB
  - /system/frameworks : 11840 KB
  - /system/lib : 27624 KB
  - ***/data** : **23768 KB**

* Sections are populated after boot

Figure 1.2: Android memory footprint
[18]

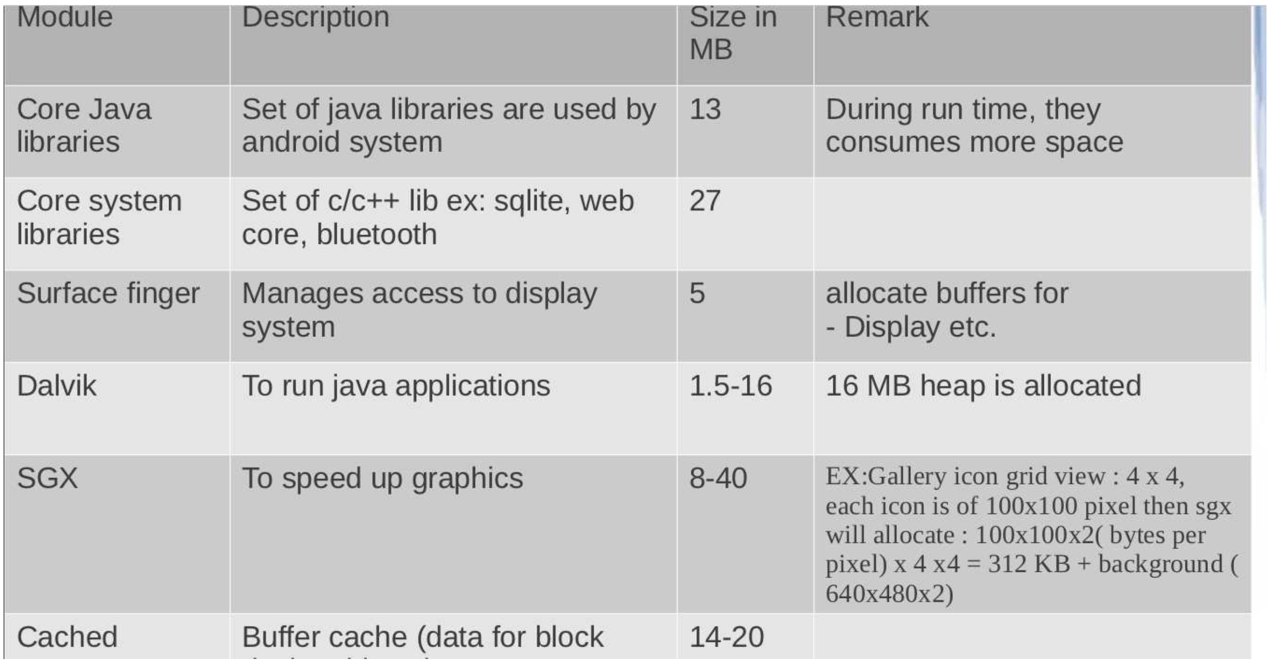
Figure 1.3 depicts the memory footprint of major processes like zygote, system_server etc. It also shows different kinds of processes which are killable in low memory situations. More details on killable/unkillable processes are explained in chapter 04. Minimum android boot configuration from figure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

Figure 1.4 depicts the memory footprint of libraries, other packages in android layers(frame work, libraries) . Dalvic module consumes 1.5MB 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.



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]



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]

1.2 Organization of the Report

Chapter 01 described two embedded operating systems android. 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. Chapter 04 covers process management in android, it describes how android process management is different from linux process management. Chapter 05 describes problems, effects of low memory scenarios and different set problems in AMS. Chapter 06 describes how AppsLogger and Improved AMS handles low memory problems using logs. Chapter 07 describes experimental results of traditional, Log-expo, Log-linear approaches in low memory scenarios.

Chapter 2

Linux Memory Management

Memory is a very limited resource especially in embedded systems like phones, tablets etc. So a good memory management system is needed. 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) translates address from one format to other format. Fig 2.1 represents how the memory translation takes place from one format to other format.

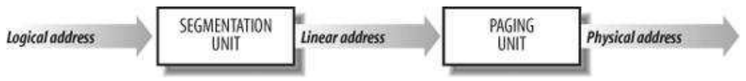


Figure 2.1: Memory addressing in Linux
[5]

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 is 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].

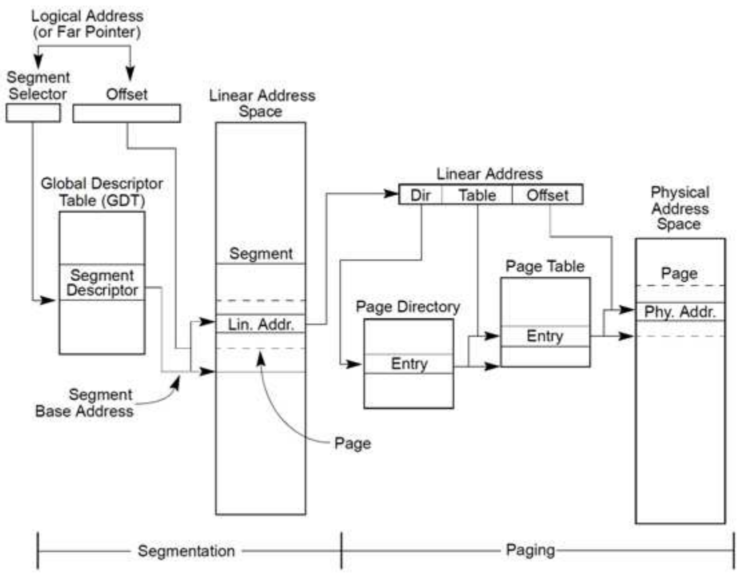


Figure 2.2: Segmentation on paging in Linux
[5]

Every process in Linux have 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 has 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.

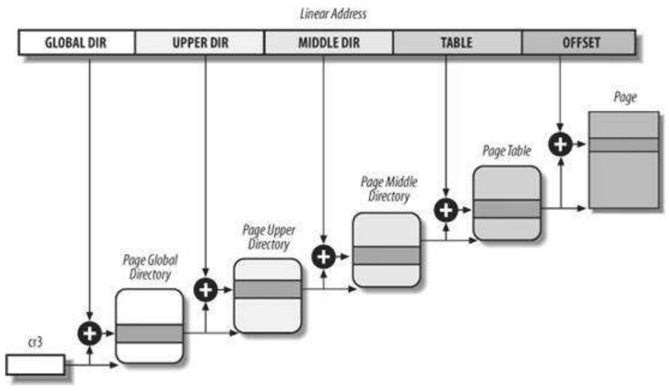


Figure 2.3: Page table in Linux
[5]

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.
   - 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 focuses on physical RAM management. All the physical RAM is classified into frames. Memory management unit dynamically allocates frames to a 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.

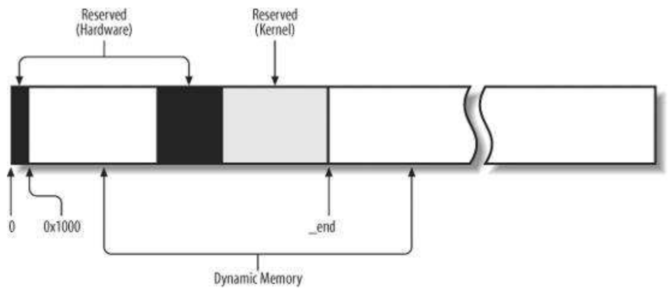


Figure 2.4: Dynamic Memory in RAM
[5]

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. It 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 ($\leq$ 16MB), 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 ZONE_NORMAL. Fig 2.5 shows the classification of RAM into zones.

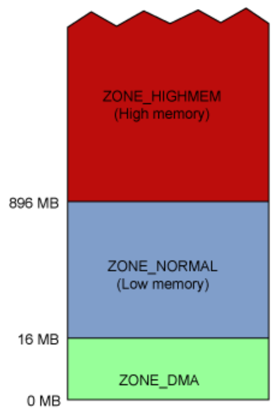


Figure 2.5: Ram classification in Linux

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 is 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].

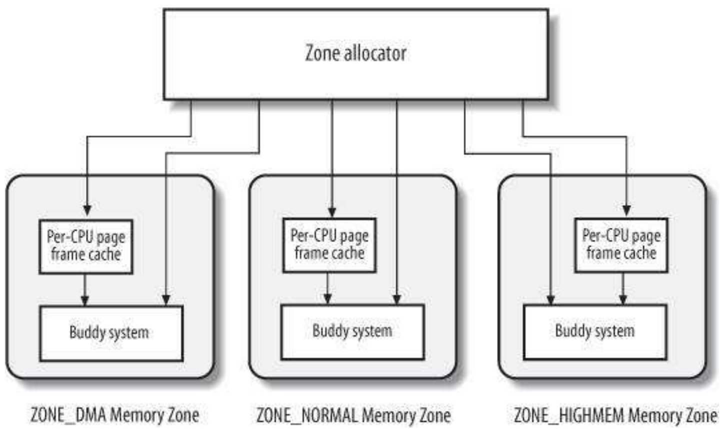


Figure 2.6: Zone allocator in Linux
[5]

2.3 Memory Allocation and Deallocation

Memory management unit handles allocation and deallocation of memory requests in a short time. It also consider that 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 is designed to avoid fragmentation with in frames. Slab allocator is mainly used for allocating data for kernel data structures[10]. Fig 2.7 shows the overall memory allocation and deallocation process. It also clearly depicts the memory allocation for kernel mode and user mode process.

1. **When new memory gets allocated**

   - Process stack grows
   - Process creates (attaches) to shared memory segment(shmat())
   - Process expands heap (malloc())

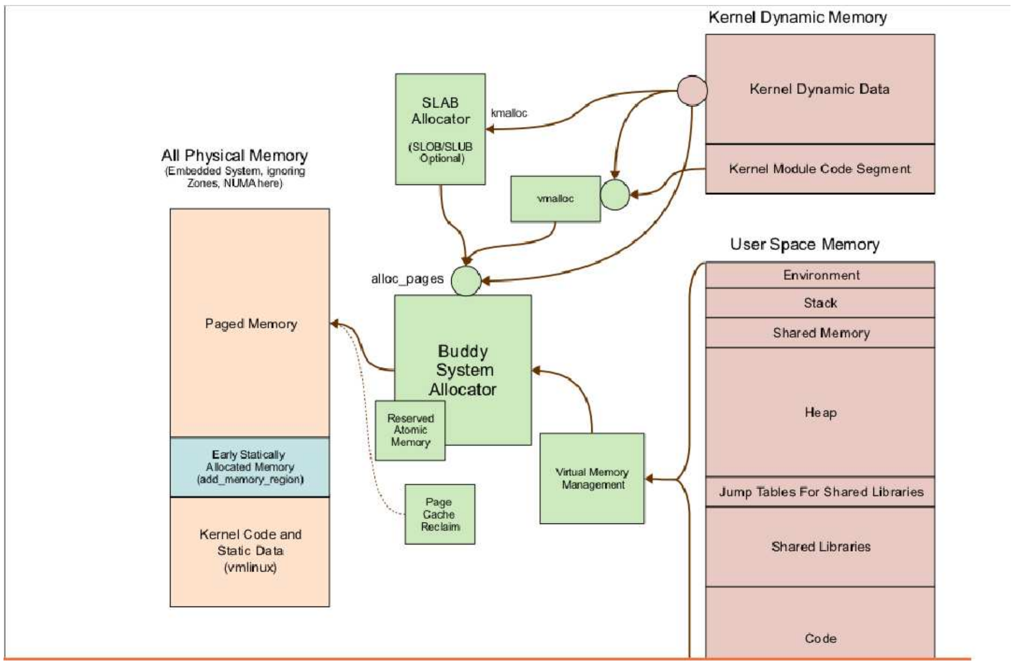


Figure 2.7: Memory Allocation & deallocation in Linux
[10]

- New process gets created (fork())
- Map a file to memory (mmap())

2. **Allocation at different levels**

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

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

   (c) Allocation at user level by 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 memory in the sizes of pages/frames. It also provides less external fragmentation. It maintains different free lists with different sizes in each zone. Example of Buddy algorithm is 4,8,16,32,64 are different groups 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.

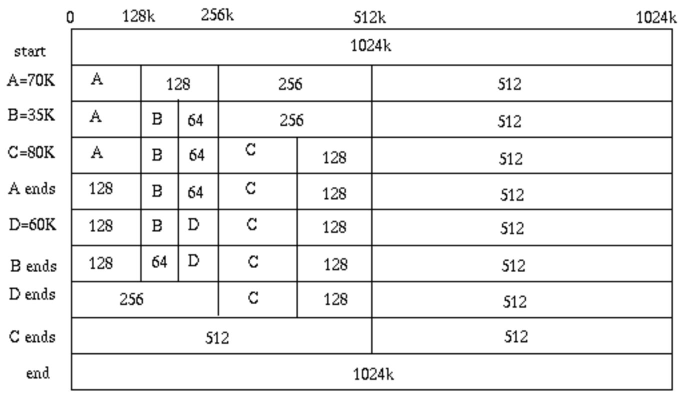


Figure 2.8: Buddy memory allocation and deallocation
[9]

Fig 2.8 explains Buddy algorithm with an example. Initially Buddy algorithm treats entire memory as single unit. Whenever a memory request is received, 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 until a small buddy which is just greater than memory request and power of 2 is allocated. And whenever a memory deallocation request is received, it simply deallocates that buddy, and tries 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 is 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 similar type of objects. Fig 2.9 depicts the clear idea of slab allocator. It shows the relation among caches, slabs and objects.

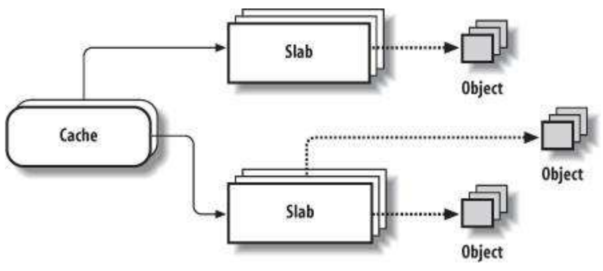


Figure 2.9: Slab allocation in Linux
[5]

1. **Slab Data Structures**
   - Cache descriptor depicts the type of object that could be cached. Slab maintains a list for all the caches.
   - Slab descriptor contains pointers to actual memory address of the object. Every slab contains some state like full, partial, and 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

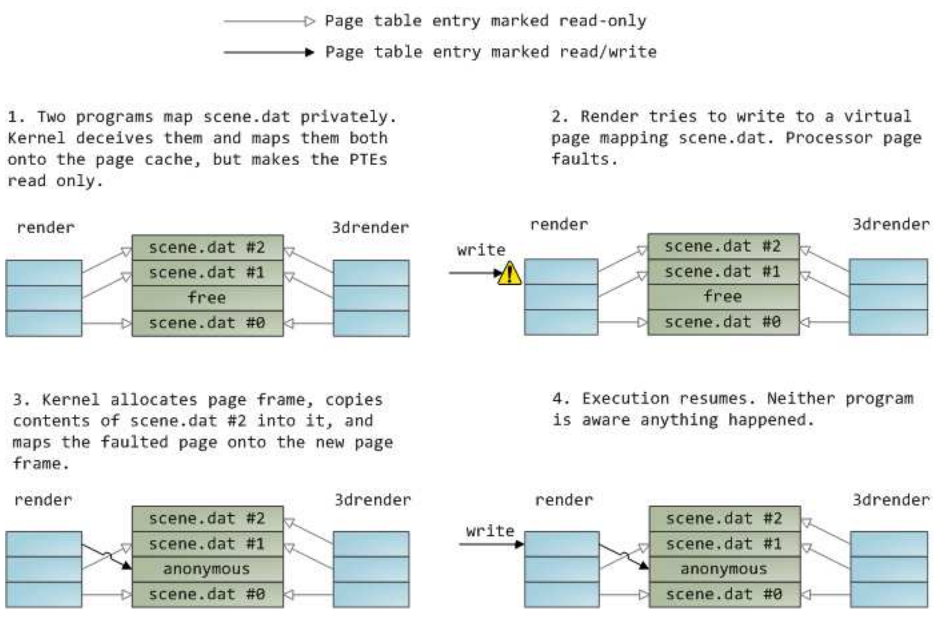
Whenever a process requests kernel to load a new file, page cache reads the file from disk. Page cache requests to the memory allocator for new pages, after the allocation of pages succeed it copies contents of file into pages. Finally using mmap() it maps corresponding physical address into process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



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.
render
3drender
scene.dat #2
scene.dat #1
free
scene.dat #0
2. Render tries to write to a virtual page mapping scene.dat. Processor page faults.
write
3. Kernel allocates page frame, copies contents of scene.dat #2 into it, and maps the faulted page onto the new page frame.
anonymous
4. Execution resumes. Neither program is aware anything happened.

Figure 2.10: page cache in Linux
[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.

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() 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.

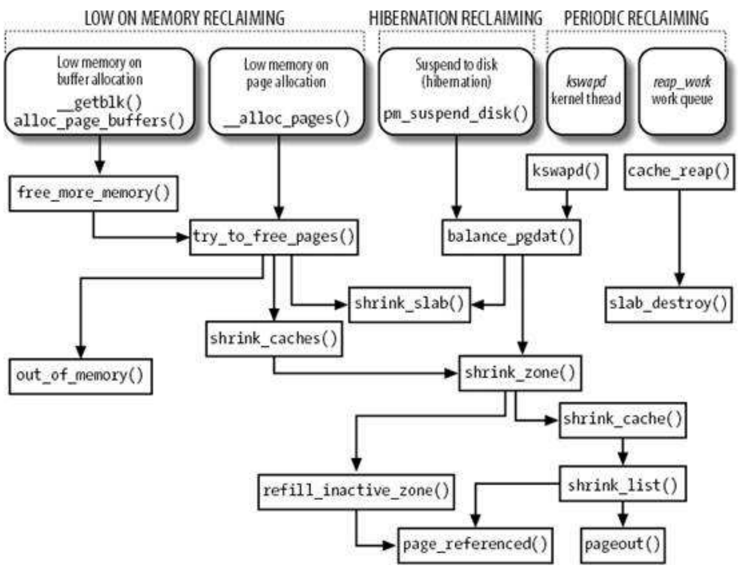


Figure 2.11: Frame Reclaiming system calls in Linux
[5]

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.

Page Frame Reclaiming Algorithm(PFRA) in linux always tries to avoid system to enter into low memory scenarios. But in some cases when it fail to reclaim sufficient memory then it calls out of memory killer to kill some of the processes. More details about working procedure and what kind of processes gets killed in low memory scenarios will be discussed in chapter 03.

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 between 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".

Android does not have swap space, because of limitations of flash memory. All android using devices like phones and tablets have very limited main memory, so there is a more probability for occurring low memory scenarios. In low memory scenarios we should not kill users interesting applications. This chapter covers analysis on Out Of Memory (OOM) Killer in Linux, and low memory killer(LMK) in android, to check whether they are giving priority to do not kill user interested applications are not.

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. 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().
- **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() 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. If the process is killable then it calculates score for this process. And finally identifies bad process as which have highest score.

```
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;
  }
```

- **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.

3.2 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).

3.2.1 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 3.1 except process nine all are 1GB processes. All the processes are started at same time in seconds(varies in micro seconds). Figure 3.1 shows the memory footprint of all the processes before OOM killer was called. From figure 3.1 we can simply say process nine is the bad process because it occupied large memory as compared with other process. This same scenario is conducted 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 occupied more memory as compared with other processes, so OOM killer gives priority to long running processes and less memory occupied process.

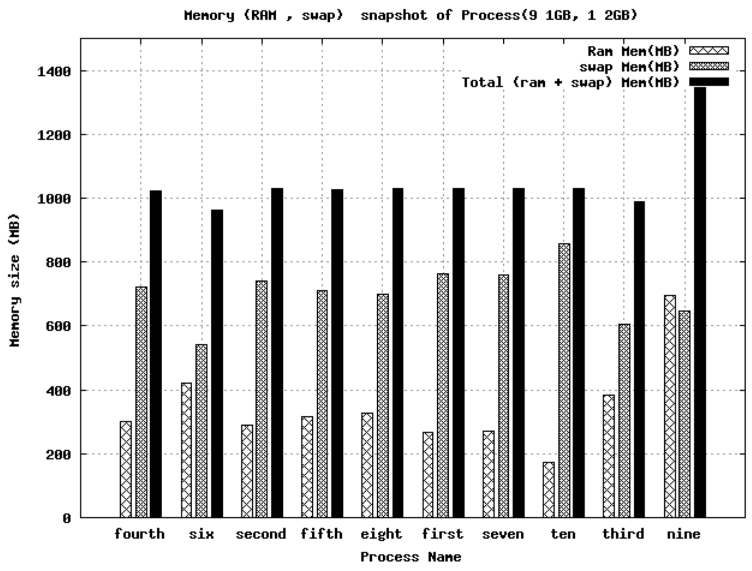


Figure 3.1: Processes(Nine 1GB, One 2GB) footprint in low memory scenario

3.2.2 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 3.2 except process nine all are 1GB processes. All the processes are started at same time in seconds(varies in micro seconds). Figure 3.2 shows the memory footprint of all the processes before OOM killer was called. From figure 3.2 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.

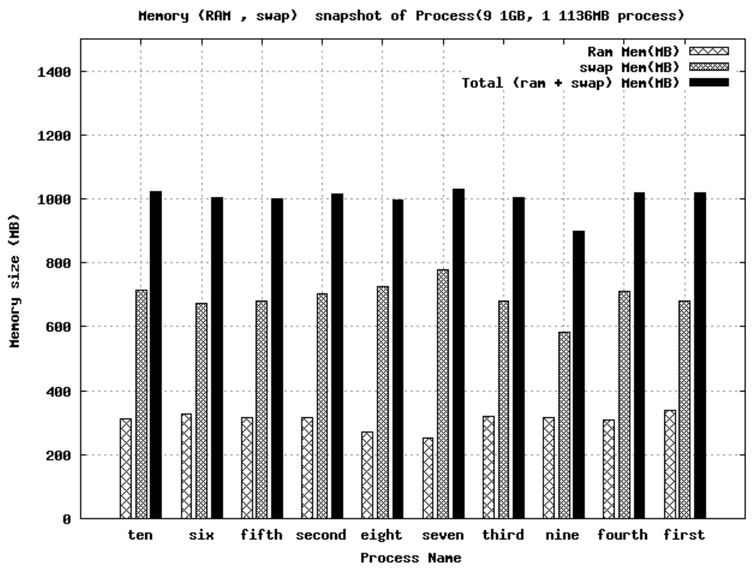


Figure 3.2: Processes(Nine 1GB, One 1136MB) footprint in low memory scenario

According to our 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 3.2 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.

3.2.3 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 3.3 shows the memory footprint of all the processes before OOM killer was called. From figure 3.3 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.

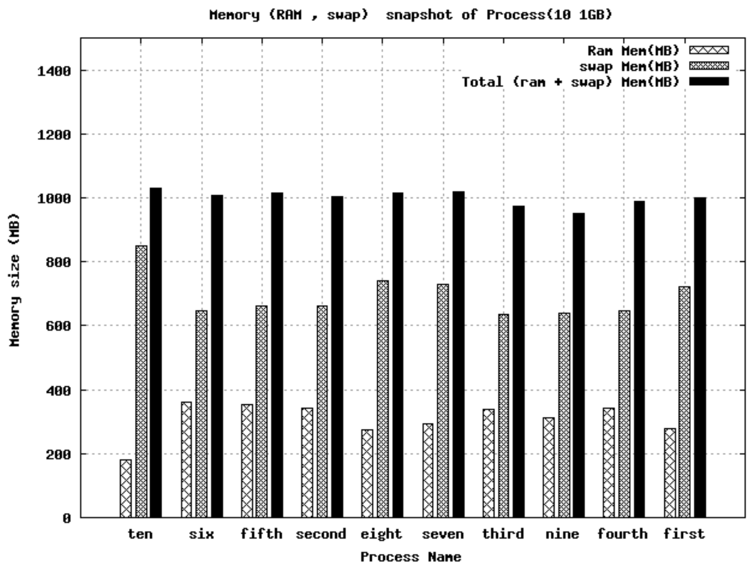


Figure 3.3: Processes(Ten 1GB) footprint in low memory scenario

One good observation from figure 3.3 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.

3.2.4 Problems in OOM Killer

1. **Problem 01**
   From section 3.2.1 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 achieve better user experience.

   Solution for the above problem is to 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. If we can maintain information about frequently accessed application, then we can implement this heuristic in OOM killer.

2. **Problem 02**
   From section 3.2.2 no fairness in the order of killing processes. Because from section 3.2.2 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.

   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. Whenever OOM killer decides to kill a processes, it checks whether this processes gets killed many times or not. If true then it searches for next bad process.

3. **Problem 03**
   From section 3.2.3, we can identify OOM killer giving equal priority to all memories (RAM + SWAP + PTS) rather than giving wighted priority to all memories. From section 3.2.3 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.

3.3 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 does not have swap space, because of limitations of flash memory. Android main memory contains so many empty(terminated) process, so order of killing should be different unlike OOM killer.

3.3.1 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.

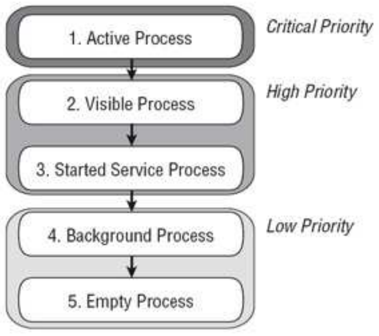


Figure 3.4: Process groups in android
[14]

More details on how a process can be moved from one group to other group, would be explained in chapter 04. Figure 3.4 depicts order of importance of groups, order of priority decreases from foreground(active) to empty.

3.3.2 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 15. 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].

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

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 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.

3.4 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.

3.4.1 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 3.2: 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.

3.4.2 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 3.3: Pros/Cons of High minfree thresholds 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. 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. 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.

Finally above problems need to be controlled to handle low memory scenarios. If we port any linux distributions into any phones are tablets, we will face all problems discussed in section 3.2.So we need to optimize OOM killer source code to handle problems discussed in section 3.2. In android before low memory killer(LMK) kills applications, activity manager service(AMS) kills some of the applications. In later chapters 5,6 we will study about AMS, approaches to reduce number of applications gets killed in a period of time.

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].

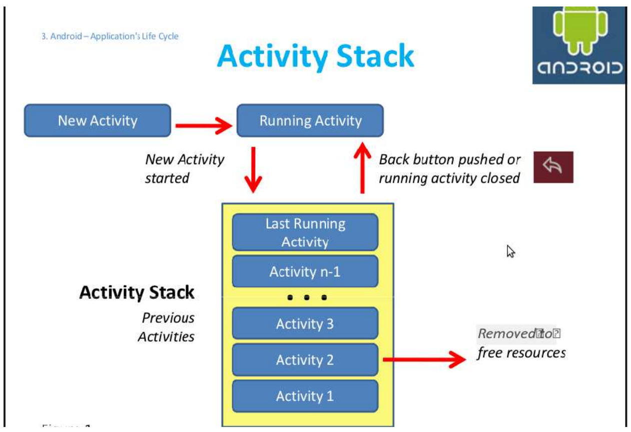


Figure 4.1: Activity Stack in android [16]

Figure 4.1 represents the pictorial view of activity stack of android. Whenever a new activity starts it is placed on top of stack, 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. **onCreate()** 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()** method called whenever user switches to 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.

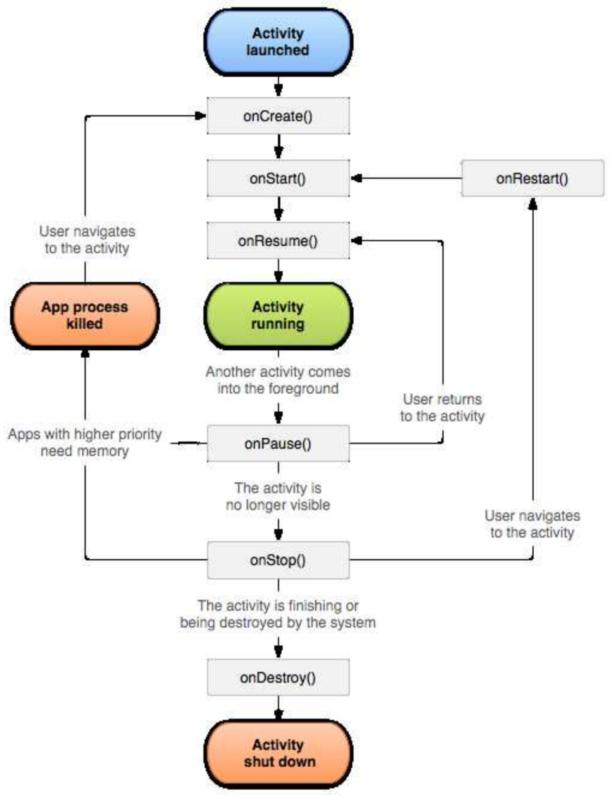

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to 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]

Activity class provides multiple methods to implement an activity. An application class can extends activity class to override methods like oncreate(), 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.

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.

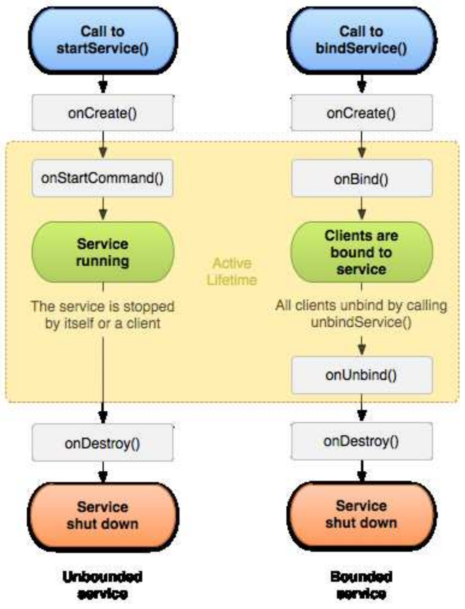


Figure 4.3: Services life cycle in android
[15]

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.

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.

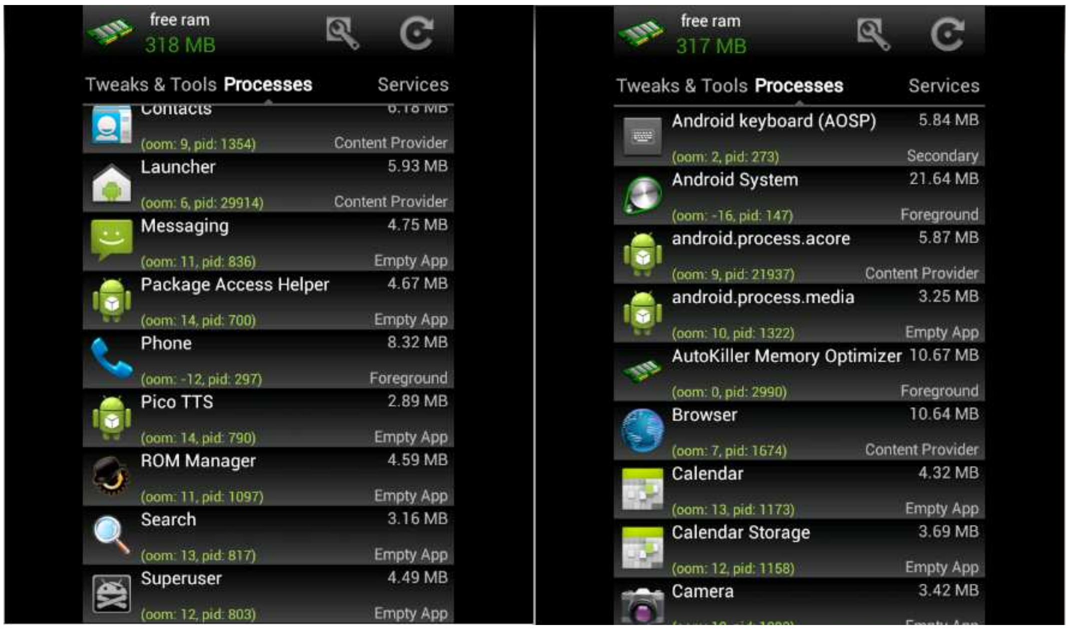
- When a service is started using startService().
- Eg: playing music in background, downloading file from Internet.

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 empty 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
```

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, we 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 Android

Low memory problems in linux, android causes to degrade the system performance, if we do not handle them in a proper manner. Low memory is nothing but, if the available free memory is less than some threshold we call it as low memory scenario. In low memory scenarios Activity manager service(AMS) and Low memory killer(LMK) kills some of the applications in the RAM. For more details about LMK are described in earlier section 3.2. Initially we thought most of the applications are get killed by LMK, and we did analysis on LMK. But later we observed that most of the applications are get killed by AMS before low memory scenarios will occur. In this chapter we will discuss about working functionality of AMS and reasons for low memory scenarios and effects of low memory scenarios.

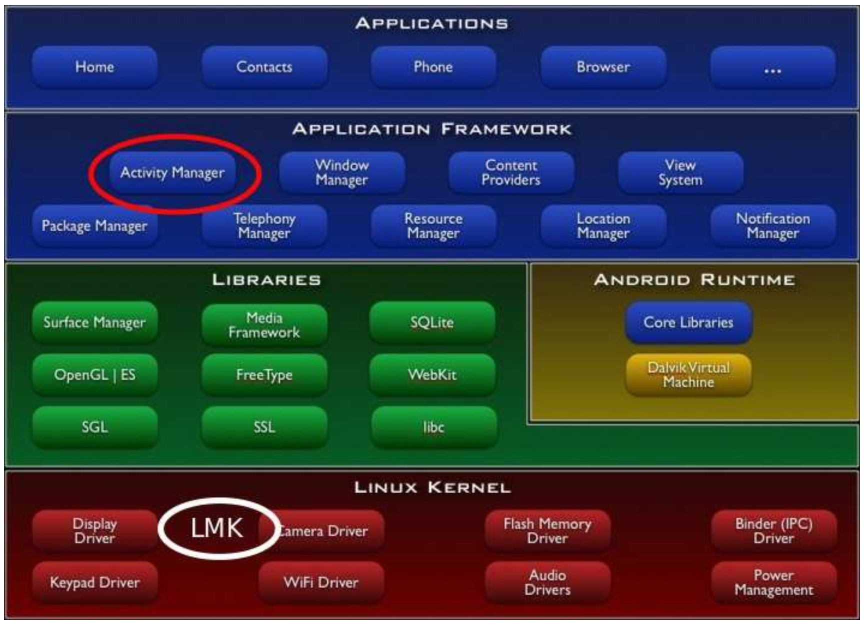


Figure 5.1: AMS & LMK position in Android Architecture

Figure 5.1 depicts the position of AMS and LMK in android architecture[1]. AMS works at framework layer, it's an interface between user and kernel. AMS receives requests from user and handover to below layers. LMK works at kernel layer and performs killing applications in low memory scenarios.

5.1 Activity Manager Service(AMS)

AMS is a service in android, which works at framework layer in android. AMS is responsible for receiving requests from user and handling them. In android during boot time system server process starts all the services like AMS, window manager, etc. Once AMS started by system server, it starts to loading other system applications like UI, home,contacts, keyboard and other third party applications those have RECEIVE_BOOT_COMPLETED permission. Figure 5.2 depicts the booting process and how AMS gets started by system server during boot process[4].

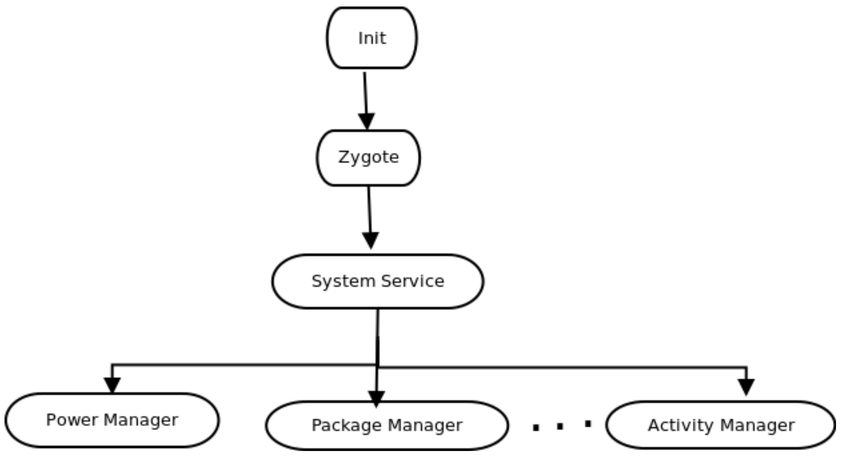


Figure 5.2: Android booting process

5.1.1 AMS Algorithm

AMS is responsible for receiving requests from user and handling them. Whenever user clicks on an application and switches from one application to other application all these requests are handled by AMS. Some of the basic functionalities of AMS are Starting a process, Update OOM_Adj values of applications, Killing a process[29]. Each application contains OOM_Adj (Out of Memory Adjust) value which represents killing priority[17] of an application. OOM_Adj value varies from -17 to 15, lower the value higher the priority. Algorithm 1 gives more details on some of the working functionalities of AMS (Android 4.2).

Algorithm 1 AMS

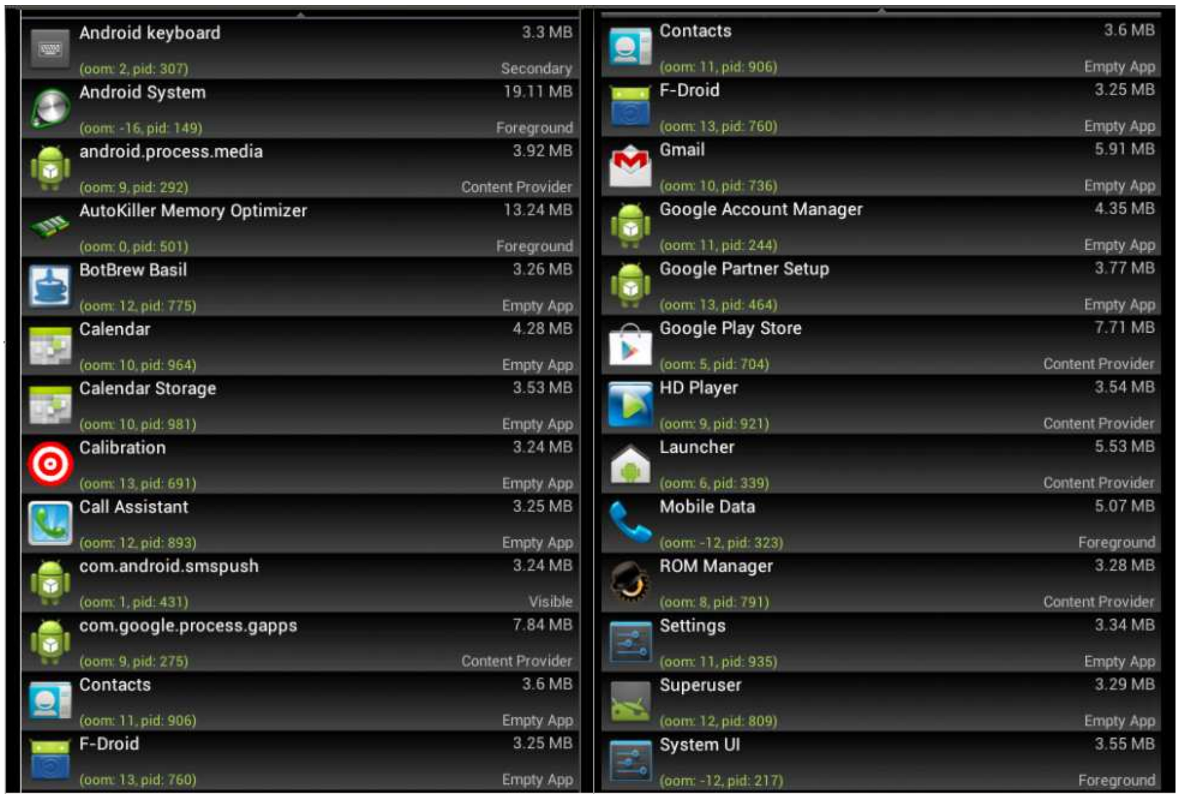
```
List lru; //least recently used applications list
mprocessLimit=24; //maximum processes (Empty + Hidden)
oldProcess=30; //maximum time(minutes) an empty process can stay in memory

// starting an application
function startProcessLocked(appName)
    start(appName);
    lru.add(appName);
    updateOomAdjLocked();
end function

//Updates OOM Adj value for all processes
function updateOomAdjLocked
    mEmpty=(mprocessLimit*2)/3; //maximum allowed empty processes
    mHidden=mprocessLimit-mempty; //maximum allowed hidden processes
    for all i: in lru list do
        if cEmpty >mEmpty then //cEmpty means current empty processes
            //too many background processes
            killProcess(lru[i]);
        end if
        if cHidden>mHidden then //cHidden means current hidden processes
            //too many background processes
            killProcess(lru[i]);
        end if
        if lru[i].lastAccessed>oldProcess then
            //old processes(if this process did not accessed from last 30 minutes)
            killProcess(lru[i]);
        end if
    end for
end function
```

5.2 Causes for Low memory Scenarios

There are many causes, those leads to low memory scenarios, some of the reasons are like keeping too many empty application in memory, keeping user infrequently accessing applications in memory and loading waste system applications into memory. This section covers different types reasons that causes to low memory scenarios in detail.

5.2.1 Empty Applications

Generally in linux whenever an user terminates from an application, all the details belongs to that application are removed from main memory. But in android, it will not remove the contents of application from main memory. Because if the user access the same application in future, we does not need to load this application from secondary memory, visit section 4.2.5 for more details about empty applications.



```
Android keyboard                      3.3 MB
(oom: 2, pid: 307)                 Secondary
Android System                      19.11 MB
(oom: -16, pid: 149)              Foreground
android.process.media                3.92 MB
(oom: 9, pid: 292)          Content Provider
AutoKiller Memory Optimizer         13.24 MB
(oom: 0, pid: 501)                Foreground
BotBrew Basil                        3.26 MB
(oom: 12, pid: 775)                Empty App
Calendar                             4.28 MB
(oom: 10, pid: 964)                Empty App
Calendar Storage                     3.53 MB
(oom: 10, pid: 981)                Empty App
Calibration                          3.24 MB
(oom: 13, pid: 691)                Empty App
Call Assistant                       3.25 MB
(oom: 12, pid: 893)                Empty App
com.android.smspush                  3.24 MB
(oom: 1, pid: 431)                   Visible
com.google.process.gapps             7.84 MB
(oom: 9, pid: 275)          Content Provider
Contacts                              3.6 MB
(oom: 11, pid: 906)                Empty App
F-Droid                              3.25 MB
(oom: 13, pid: 760)                Empty App
```

```
Contacts                              3.6 MB
(oom: 11, pid: 906)                Empty App
F-Droid                              3.25 MB
(oom: 13, pid: 760)                Empty App
Gmail                                5.91 MB
(oom: 10, pid: 736)                Empty App
Google Account Manager               4.35 MB
(oom: 11, pid: 244)                Empty App
Google Partner Setup                 3.77 MB
(oom: 13, pid: 464)                Empty App
Google Play Store                    7.71 MB
(oom: 5, pid: 704)          Content Provider
HD Player                            3.54 MB
(oom: 9, pid: 921)          Content Provider
Launcher                             5.53 MB
(oom: 6, pid: 339)          Content Provider
Mobile Data                          5.07 MB
(oom: -12, pid: 323)              Foreground
ROM Manager                          3.28 MB
(oom: 8, pid: 791)          Content Provider
Settings                             3.34 MB
(oom: 11, pid: 935)                Empty App
Superuser                            3.29 MB
(oom: 12, pid: 809)                Empty App
System UI                            3.55 MB
(oom: -12, pid: 217)              Foreground
```

Figure 5.3: List of Empty applications inside memory

The figure 5.3 depicts the list of applications reside in main memory and their states. From figure 5.3, we can observe some facts like main memory is full with so many empty application. So, there are many advantages and disadvantages with empty applications. Figures 5.3,5.4,5.5 are taken from Aakash tablet by using autoKiller application.

Some of the Advantages of Empty Applications:

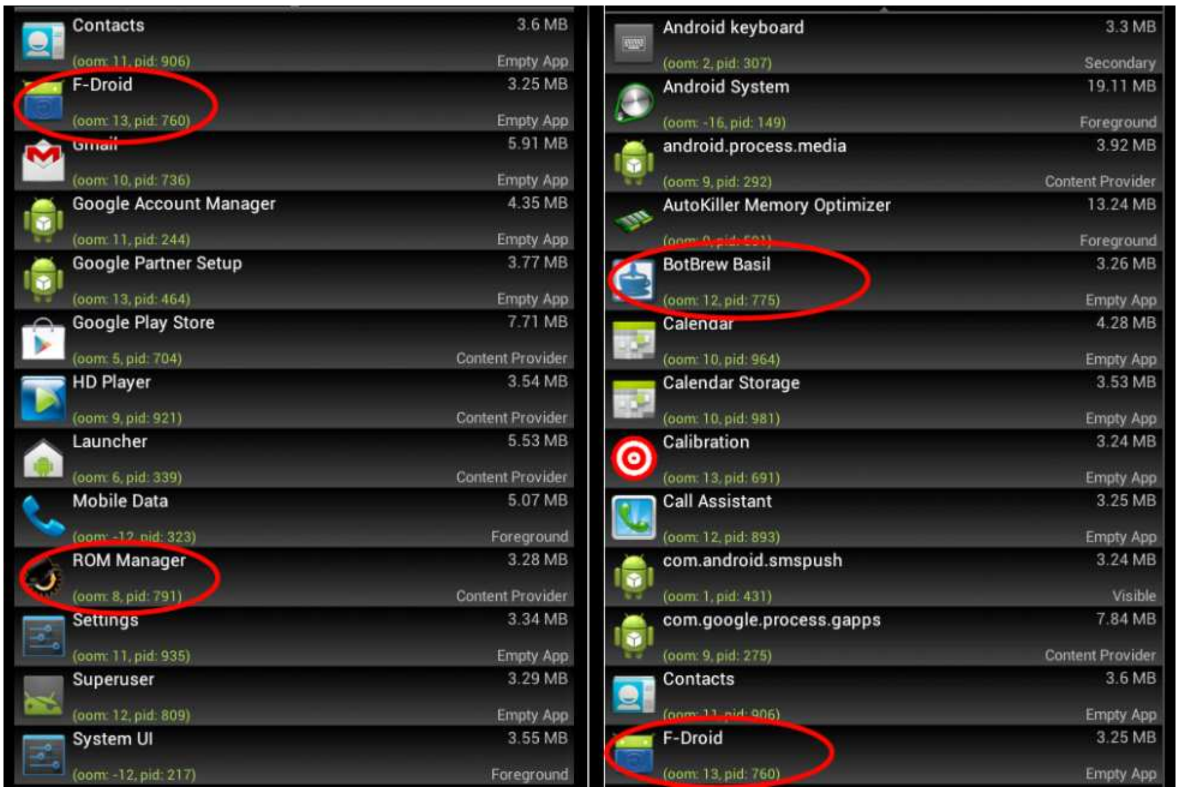
- Reduces the loading time of applications available in the main memory. So finally reduces response time.
- For loading an application from secondary memory will consume more power, so if we avoid to many loads then we can some save power consumption.

Some of the Disadvantages of Empty Applications:

- Increases the response time of applications which are not there in main memory.
- Too much of empty applications leads to frequent low memory scenarios.

5.2.2 RECEIVE_BOOT_COMPLETED

Android API provides a permission like RECEIVE_BOOT_COMPLETED to developers. This permission provides facility to developer to start an application immediately after boot completion[3]. If any applications have this permission will be started by AMS immediately on receiving RECEIVE_BOOT_COMPLETED broadcast message.
We have installed 35 number of applications on our emulator, out of those 13 number of applications have RECEIVE_BOOT_COMPLETED permission. So irrespective of whether user is using those applications are not they are loaded into main memory after booting.



```
Contacts                              3.6 MB
(oom: 11, pid: 906)                Empty App
F-Droid                              3.25 MB
(oom: 13, pid: 760)                Empty App
Gmail                                5.91 MB
(oom: 10, pid: 736)                Empty App
Google Account Manager               4.35 MB
(oom: 11, pid: 244)                Empty App
Google Partner Setup                 3.77 MB
(oom: 13, pid: 464)                Empty App
Google Play Store                    7.71 MB
(oom: 5, pid: 704)          Content Provider
HD Player                            3.54 MB
(oom: 9, pid: 921)          Content Provider
Launcher                             5.53 MB
(oom: 6, pid: 339)          Content Provider
Mobile Data                          5.07 MB
(oom: -12, pid: 323)              Foreground
ROM Manager                          3.28 MB
(oom: 8, pid: 791)          Content Provider
Settings                             3.34 MB
(oom: 11, pid: 935)                Empty App
Superuser                            3.29 MB
(oom: 12, pid: 809)                Empty App
System UI                            3.55 MB
(oom: -12, pid: 217)              Foreground
```

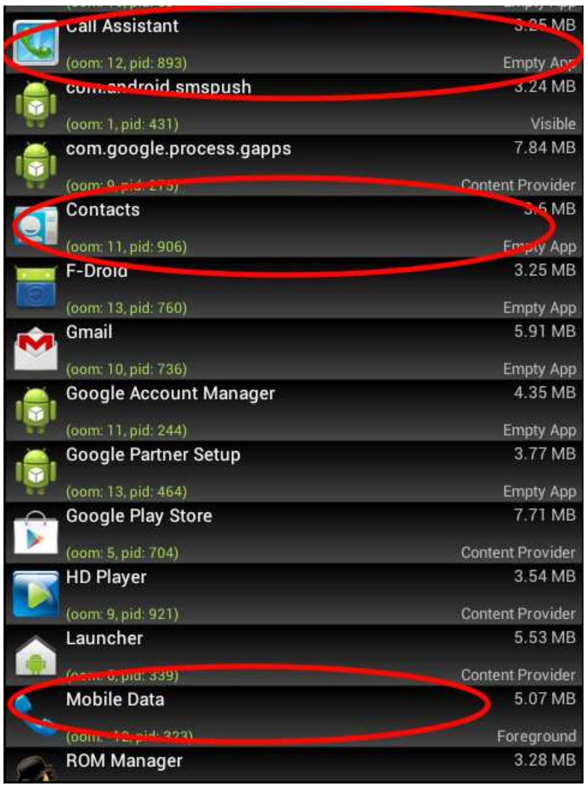
```
Android keyboard                      3.3 MB
(oom: 2, pid: 307)                 Secondary
Android System                      19.11 MB
(oom: -16, pid: 149)              Foreground
android.process.media                3.92 MB
(oom: 9, pid: 292)          Content Provider
AutoKiller Memory Optimizer         13.24 MB
(oom: 0, pid: 591)                Foreground
BotBrew Basil                        3.26 MB
(oom: 12, pid: 775)                Empty App
Calendar                             4.28 MB
(oom: 10, pid: 964)                Empty App
Calendar Storage                     3.53 MB
(oom: 10, pid: 981)                Empty App
Calibration                          3.24 MB
(oom: 13, pid: 691)                Empty App
Call Assistant                       3.25 MB
(oom: 12, pid: 893)                Empty App
com.android.smspush                  3.24 MB
(oom: 1, pid: 431)                   Visible
com.google.process.gapps             7.84 MB
(oom: 9, pid: 275)          Content Provider
Contacts                              3.6 MB
(oom: 11, pid: 906)                Empty App
F-Droid                              3.25 MB
(oom: 13, pid: 760)                Empty App
```

Figure 5.4: List of applications have RECEIVE_BOOT_COMPLETED permission

From figure 5.4 depicts some list of applications (rounded in red circle) those are started after booting. If we can identify which applications are more frequently used by user than we can kill other unimportant applications in low memory scenarios.

5.2.3 Loading System Apps

Android contains two kinds of applications one is system applications and second one is third party applications. In general system applications are installed in /system/app folder third party applications are installed in /data/app folder. System application have more privileges than third party application to access internal resources. AMS loads some of system applications into main memory during boot time. Some of the applications are phone, contacts, email , messages, and email etc. AMS never consider whether user is using Internet or not, irrespective of that it simply loads email, browser etc applications into memory.



Call Assistant	3.25 MB
(oom: 12, pid: 893)	Empty App
com.android.smspush	3.24 MB
(oom: 1, pid: 431)	Visible
com.google.process.gapps	7.84 MB
(oom: 9, pid: 275)	Content Provider
Contacts	3.6 MB
(oom: 11, pid: 906)	Empty App
F-Droid	3.25 MB
(oom: 13, pid: 760)	Empty App
Gmail	5.91 MB
(oom: 10, pid: 736)	Empty App
Google Account Manager	4.35 MB
(oom: 11, pid: 244)	Empty App
Google Partner Setup	3.77 MB
(oom: 13, pid: 464)	Empty App
Google Play Store	7.71 MB
(oom: 5, pid: 704)	Content Provider
HD Player	3.54 MB
(oom: 9, pid: 921)	Content Provider
Launcher	5.53 MB
(oom: 6, pid: 339)	Content Provider
Mobile Data	5.07 MB
(oom: 12, pid: 323)	Foreground
ROM Manager	3.28 MB

Figure 5.5: List of System apps loaded inside memory

Figure 5.5 shows list of phone related applications reside in main memory even though Aakash tablet do not have any phone facility. AMS loads all phone, contacts, messages applications into main memory, irrespective of existence of phone hardware on device. Aakash tablet does not have any phone facility, but phone application always reside in main memory. Android contains two kinds of processes persistent and non-persistent.

Processes which have persistent property true they always reside in main memory. Phone application have persistent property, so it is always reside in main memory.

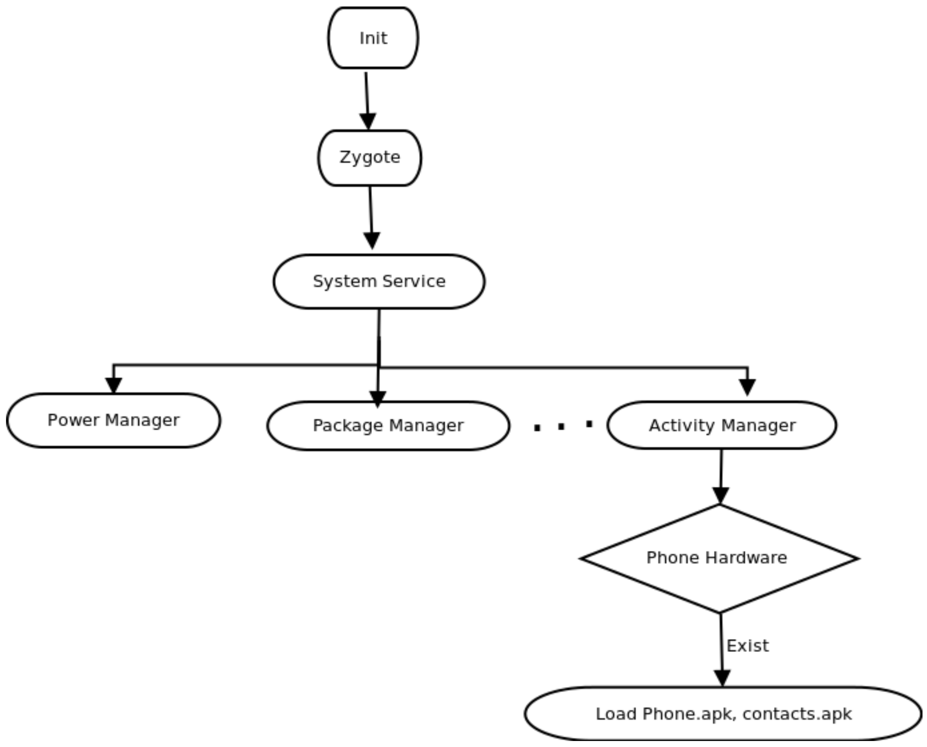


Figure 5.6: Modified AMS for loading phone application

So if we modify AMS like check phone hardware supporting facility on device and if exist then only load into main memory. Figure 5.6 depicts pictorial representation modified AMS for loading phone related applications into main memory.

5.3 Effects of Low Memory Scenarios

In low memory scenarios AMS and LMK kill some of the applications. If user wants to access the killed application in future we need to load such application from secondary storage. Loading an application from secondary memory takes more time, so for every kill operations leads to one more load operation, if user wants to access that killed application.

Figure 5.7 represents cost of load operation. So to avoid too many load operations, we have to kill applications very carefully. If we kill applications, which are infrequently used by user so we can reduce number of load operations. load is a costly operation, so we have to minimize as many as load operations. If we can reduce number of user frequently

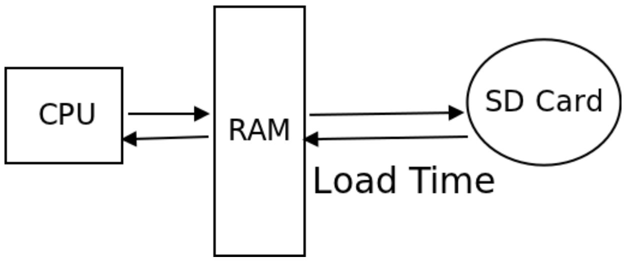


Figure 5.7: Effects of Low memory scenarios

accessing applications get killed, then we can minimize number of load operations. Finally we can reduce response time of accessing frequently accessed applications.

5.4 Problems in AMS

AMS kills more application before low memory scenarios will occur. Most of the applications are get killed by AMS rather than LMK. Section 5.1.1 gives more details on working procedure of AMS. AMS does not consider user infrequently accessing applications for killing and it does not consider free memory size while killing.

5.4.1 Does not considers free memory size

AMS kills applications in low memory scenarios based on LRU list. While killing applications it does not consider the free memory available in main memory. From figure 5.8 we can observe that nearly 400,250 applications were get killed even though free memory is between 250-300MB, 300-350MB respectively. AMS kills applications before low memory scenarios will occur, because it does not allow number of empty applications more than some threshold. So we need consider free memory available in main memory and to do not kill applications aggressively. Finally we can reduce number of applications gets killed in a period of time.

5.4.2 Does not considers infrequently accessed applications for killing

AMS does not consider an application is interested to user or not before killing that application. Because it simply kills application based on LRU and it does not consider the frequency of that application. If AMS kills user interested applications then we need to reload such application. So if we can predict the user future accessing applications based on frequency and recency. Finally, we can protect such kinds of applications get killed in low memory scenarios, then we can reduce number of applications get killed and number of load operations.

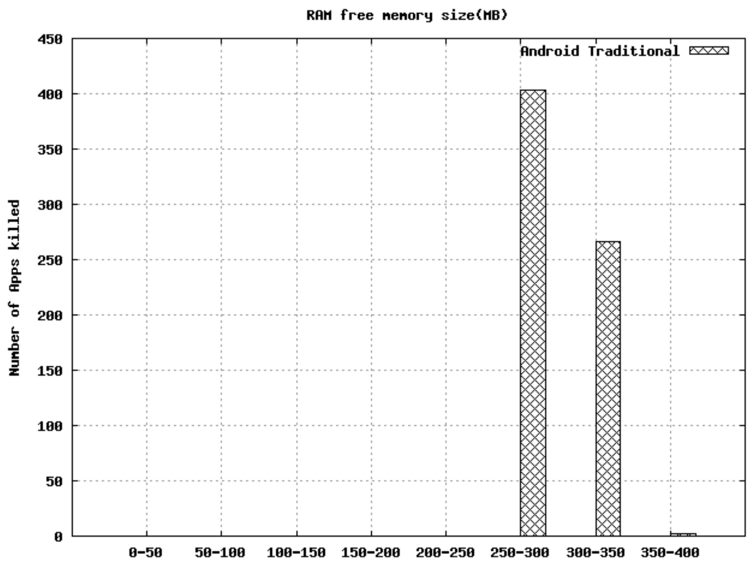


Figure 5.8: Free memory size, when application get killed

$AppName$	$Today$	$Day2$	$Day3$	$Day4$	$Day5$	$Day6$	$Day7$
aakash.lab	54	159	18	1	248	50	67
android.email	0	0	0	0	0	0	0
aakashdemo	2	2	0	0	2	3	12
fdroid	0	0	0	0	1	0	0
android.su	0	0	0	0	0	0	0
mindicator	55	125	21	8	244	43	64
autokiller	152	130	179	4	415	50	82

Table 5.1: Frequency of applications accessed in a period of one week

Table 5.1 has taken from android emulator, which depicts the frequency of some applications accessed by user in a period of one week. From table 5.1, we can observe some of the applications were infrequently used by user. if we can kill such kind of infrequently accessed application in low memory scenarios, we can save user interesting application.

So, if we can predict the user interesting applications based frequency and recency of applications, then we can protect those applications in low memory scenarios. Such that we can reduce number of applications gets killed in a period of time. Finally we can minimize number of load operations. AMS kills applications before low memory scenarios will occur and it kills applications based on LRU. If we can consider both recency and frequency while killing applications, we can reduce number of applications get killed.

Chapter 6

Handling Low Memory Scenarios in Android using Logs

In general different users are interested in different applications. So interested application to user varies from one person to another. But if we can track user interacting applications using logs, we can simply predict user important or interesting applications. This chapter gives details on design and implementation of AppsLogger application and modified AMS algorithm. AppsLogger is an application, which tracks the user interacting applications and predicts the list of user interesting applications. Now modified AMS before killing any application, it checks whether this application is exist in the list of user interested applications or not. If exists then does not kills otherwise it kills.

6.1 Developed Android Apps

Before developing AppsLogger application, we developed other applications with simple features. Finally we are using all these features in our final application called AppsLogger.

6.1.1 Database Application

This application contains features like creating database, insert/retrieve data from database[21].

6.1.2 Content Provider Application

Android API provides two features like Content Providers, Content Resolver[22][23]. By using Content Providers, Content Resolver we can access database of other application. Content Provider provides an URI to access database. URI helps for unique identification of database. Content Resolver access the database using the URI provided by Content Provider.

6.1.3 Process Info Application

This application contains features like getting list of running applications, running tasks, running services[25]. It also provides features like memory details of running applications, list of permissions of installed applications, Check hardware facility for making Phone calls exist or not, if hardware facility not exist for making calls then it removes phone.apk application.

6.2 Design and Implementation of AppsLogger

AppsLogger application tracks the applications interacted by the user and predicts the user future accessing applications. It contains components like Logs Parser, Score Calculator, and Apps Manager each have their individual roles. Figure 6.1 depicts the communication among components of AppsLogger and modified AMS. AppsLogger contains main task activity, which is responsible for co-ordinating all other components.

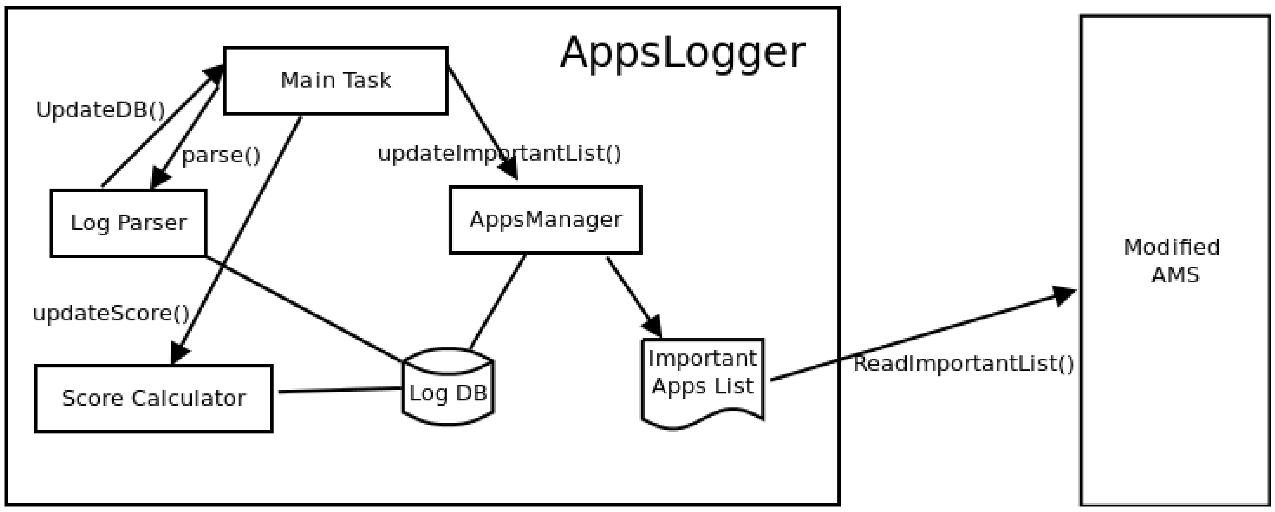


Figure 6.1: Components of AppsLogger

6.2.1 Logs Parser

Logs Parser component responsible for tracking user log records and parsing[26]. After parsing it stores applications details in a database. For each application it stores recently accessed time, frequency of that application gets accessed in a period of one week day wise, and memory size occupied by that application. For more details on structure of frequency count of applications look at section 5.4.2. Algorithm 2 gives more details on functionality of logs parser component. Whenever number of applications accessed by user is greater than 10, Log parser intimates to main task to recompute scores and interested application list.

Algorithm 2 Logs_parser

```
// parsing log buffer periodically
function LOG_BUFFER_PARSER
    while (True) do
        if (log buffer full) then
            applications_Accessed=log_Record_Parser();
            if (applications_Accessed > threshold) then
                updateDB() //publish_progressToMainTask;
            end if
        end if
        sleep(time); // sleep some time
    end while
end function

// parsing log records
function LOG_RECORD_PARSER()
    while (data in log buffer) do
        //parses log records and collects application name and accessed time
        parseLine();
    end while
    //updates the corresponding applications recently accessed time and frequency
    updateDetails();
return numOfApplicationsAccessed
end function
```

6.2.2 Score Calculator

Score calculator component calculates score for each application stored in Database[21]. Score represents importance of application, higher the score highly important. Main task sends request to score calculator, immediately after receiving updateDB()[22][23] call from Logs Parser. We followed two approaches to calculate score one is linear approximation and second one is exponential function.

Algorithm 3 Score_Calculator

```
1: //Retrieve logs database
2: applications= getLogsDatabaseRecords();
3: //Calculates score using linear or exponential formula
4: for all i: in applications do
5:     score=linear() (or) exponential();
6:     //Update score for corresponding application
7:     updateScoreAppWise(application[i]);
8: end for
```

AppsLogger database contains details for each application like recency (how recently used that application), frequency day wise for one week(Today to last 6 days(day2 - day7)) and it also stores how much memory it occupies inside main memory[28]. All these attributes are input to the score calculator and it calculates score for each application using below formulas. Higher the score higher probability for accessing that application in future.

Linear Approximation

Assigning proper weights to the input attributes has to be done in a proper format. Weights plays vital role in selecting user interested application list. So first of all we need to identify, which attributes we need to give more priority. In this specific problem from the set of input attributes, recency and frequency of today, are the important attributes. The reason for giving more priority to frequency of today is there is a more probability for accessing those applications. Later we analyzed this problem with Firefox frecency(frequency + recency) algorithm[27]. Firefox frecency algorithm calculates score for each URI visited by user based upon recency and frequency of last 90 days history. Firefox frecency algorithm also classifies attributes in the order of priority and assigns weights to attributes based upon priority. It assigns weights based on linear formula like w(x1)-w(x2)=20, i.e for first priority attribute it assigns 100 and for remaining ones it decreases weight by 20 respectively. In our problem we are making sum weights equals to 100.

First time when we conducted experiments,we given 55% weightage to frequency of today,day2 and recency. we are using linear formula as w(x1)-w(x2)=5. Starting weight for linear formula is 30 and sum all weights are equal to 100.
score= (recency * w2)/1000 + ((today + day2)*w1) + ((day3+ day4)*w3) +

((day5+day6 +day7)*w5) - (avgMemSize * w4)

In second time, we given 50% weightage to frequency of today and recency. In linear approximation weights are given as w1=25,w2=25,w3=25,w4=15,w5=10.
score= (recency * w2)/1000 + (today*w1) + ((day2 + day3+ day4)*w3) + ((day5+day6 +day7)*w5) - (avgMemSize * w4)

For assigning weights we tried with multiple combinations, most of the times, frequently accessed applications list is almost same. But order of applications in the list is different. Experimental result in chapter-07 are taken from second time.

Exponential Function

Exponential decay function provides values in decreasing order, if we can arrange our input attributes in an order of decreasing priority, then we can arrange weights respectively. In exponential score calculator formula we given more weightage to recency rather than frequency.
$w(x) = e^{-x}$;
score= (recency * w1)/1000 + ((today + day2)*w2) + ((day3+ day4)*w3) + ((day5+day6 +day7)*w5) - (avgMemSize * w4)

6.2.3 Apps Manager

Apps Manager component responsible for preparing user interested and uninterested applications list using applications log database[23]. It considers user interesting applications as top 10 which have highest score. It considers very least uninteresting applications as top 10 which have lowest score. It would be better if you we can increase or decrease number of interested applications based on available free memory size. Main task sends request to Apps Manager, immediately after receiving updateDB() call from Logs Parser. Algorithm 4 explains more on functionality of Apps manager component.

6.3 Improved AMS

Most of the scenarios AMS kills applications before low memory scenarios occur, look at section 5.4.1 for more details. AMS kills applications based upon lru and AMS does not consider user interesting applications, while killing. So we need to improve AMS in these two issues, such that we can reduce number of applications gets killed.

6.3.1 AMS-v01

AppsLogger application predicts user interesting applications based on user log history. AppsLogger periodically prepares user interesting applications and shares that list with

Algorithm 4 Apps_Manager

```
 1: List **unkillableApps**;
 2: List **killableApps**;
 3:
 4: // Preparing user interested applications
 5: function INTERESTED_APPLICATIONS_LIST
 6:     // Retrieve applications in increasing order group by score from logs database
 7:     while (i : application in logs database < threshold) do
 8:         unkillableApps.add(application[i]);
 9:     end while
10: end function
11:
12: // Preparing user infrequently accessed applications
13: function UNINTERESTED_APPLICATIONS_LIST
14:     // Retrieve applications in decreasing order group by score from logs database
15:     while (i : application in logs database < threshold) do
16:         killableApps.add(application[i]);
17:     end while
18: end function
```

AMS. Now we are improving source code of AMS (Android 4.2) as do not kill user interested applications until very low critical memory scenarios. Before killing any application AMS-v01 checks, whether this application is in interested applications list or not. If it is present in memory and free memory greater than some threshold(30MB) do not kill. If that application does not present in interested applications list, then kill irrespective of free memory size. For killing applications not in interested applications list, AMS-v01 does not consider free memory size. Algorithm 5 describes the improved functionality of AMS.

6.3.2 AMS-v02

In this AMS version, we are considering free main memory size while killing an application. From section 5.4.1 we can clearly say that in traditional AMS most of the applications are gets killed, even though free memory is between 250-300MB out of 512MB. Even though free memory is greater than 3/5 of main memory, still AMS kills applications. So if we consider free memory size while killing applications, we can reduce number of applications gets killed. In AMS-v02 we are added one more constraint to AMS-v01, like do not kill any application until free memory is less than some threshold (2/5 of main memory, i.e = 200MB). We need to keep some free memory always for running applications, because they use that free memory for loading files(video, audio, and downloading data from Internet, etc). Algorithm 5 describes the improved functionality of AMS.
AppsLogger tracks user interacting applications list and predicts user interested

Algorithm 5 AMS-v01

```
List lru; //least recently used applications list
mprocessLimit=24; //maximum processes (Empty + Hidden)
oldProcess=30; //maximum time(minutes) an empty process can stay in memory

//List of interested applications received from AppsLogger application
List interestedApps;
// starting an application
function startProcessLocked(appName)
    start(appName);
    lru.add(appName);
    updateOomAdjLocked();
end function

//Updates OOM Adj value for all processes
function updateOomAdjLocked
    mEmpty=(mprocessLimit*2)/3; //maximum allowed empty processes
    mHidden=mprocessLimit-mempty; //maximum allowed hidden processes
    for all i: in lru list do
        //if this process is in interested apps list and
        //free mem > 30MB, then do not kill this application
        if (!interestedApps[lru[i]] & freeMem>=30) then
            if cEmpty >mEmpty then //cEmpty means current empty processes
                //too many background processes
                killProcess(lru[i]);
            end if
            if cHidden>mHidden then //cHidden means current hidden processes
                //too many background processes
                killProcess(lru[i]);
            end if
            if lru[i].lastAccessed>oldProcess then
                //old processes(if this process did not accessed from last 30 minutes)
                killProcess(lru[i]);
            end if
        end if
    end for
end function
```

Algorithm 6 AMS-v02

```
List lru; //least recently used applications list
mprocessLimit=24; //maximum processes(Empty + Hidden)
oldProcess=30; //maximum time(minutes) an empty process can stay in memory

//List of interested applications received from AppsLogger application
List interestedApps;
// starting an application
function startProcessLocked(appName)
    start(appName);
    lru.add(appName);
    updateOomAdjLocked();
end function

//Updates OOM Adj value for all processes
function updateOomAdjLocked
    mEmpty=(mprocessLimit*2)/3; //maximum allowed empty processes
    mHidden=mprocessLimit-mempty; //maximum allowed hidden processes
    for all i: in lru list do
        //Do not kill applications untill free memory less than threshold(200MB)
        if (freeMem < threshold) then
            //if this process is in interested apps list and
            //free mem > 30MB, then do not kill this application
            if (!interestedApps[lru[i]] & freeMem>=30) then
                if cEmpty >mEmpty then //cEmpty means current empty processes
                    //too many background processes
                    killProcess(lru[i]);
                end if
                if cHidden>mHidden then //cHidden means current hidden processes
                    //too many background processes
                    killProcess(lru[i]);
                end if
                if lru[i].lastAccessed>oldProcess then
                    //old processes(if this process did not accessed from last 30 minutes)
                    killProcess(lru[i]);
                end if
            end if
        end if
    end for
end function
```

applications. Improved AMS-v01, AMS-v02 do not kill user interesting applications. Finally these two plays vital role in reducing number of applications get killed in low memory scenarios.

Chapter 7

Experimental Results

This chapter covers different experiments done on low memory scenarios using AppsLogger and AMS. It also includes statistics about number of applications get killed and response time in different approaches.

7.1 Experimental setup

All these experiments conducted on an emulator. We are using Android 4.2, we modified the AMS source code as per requirements in section 6.3.1 and 6.3.2. Initially we tried to implement all the functionalities like do not kill user interested applications using AppsLogger application, we tried three approaches but all were failed. So later we decided to modify AMS source code to provide such functionality.

Emulator configuration:

- OS: Android 4.2
- RAM: 512MB
- Tool: monkey

Monkey is a tool which helps to interact with an application on a device, without user. Monkey takes input as application name and number of events to be interacted and delay between each event. Once monkey receives proper inputs, it starts that application and performs different events on that application without user interaction.

$MethodName$	$AppLogger$	$ScoreFormula$	$AMS-version$
Traditional	No	No	Traditional
Log-Expo	Yes	Exponential	AMS-v01
Log-Linear	Yes	Linear	AMS-v02

Table 7.1: Experimental setup

We conducted experiments in three approaches, Table 7.1 clearly depicts different components used in each approach. In every approach, we have conducted five rounds, each round conducted per 1 hr except fifth round. In fifth round we increased time by 10 minutes.We have installed 30 applications on our emulator. In every round different set of applications are given as input to monkey tool. We are maintaining two lists of frequent and infrequent applications lists. In each round, we are selecting applications from these two lists based upon some ratio and given as input to the Monkey. These are the corresponding ratios from first round to fourth round of frequent, infrequent lists 3:5,1:1,1:1, and 5:3 respectively. In fifth round applications are selected from both the lists randomly. The number of applications accessed in every round are approximately equal, But they are not exactly equal. The reason is Monkey tool randomly performs some of events on given input application, the time taken for doing those events varies based on type of events. Some of the events are like clicking buttons, loading a file, typing some text, and browsing an Url etc.

7.1.1 Traditional

In this approach, we have not done any changes to AMS source code and we are not using log based killing(AppsLogger). Table 7.2 gives details on total response time(Resp.Time) in seconds, average response time (AvgResp.Time) in seconds and number of applications get killed in each round in traditional approach. No.Apps, No.Access in Table 7.2 represents number of distinct applications accessed, total number of times all applications are accessed in each round respectively.

$RoundNo$	$No.Apps$	$No.Access$	$Resp.Time$	$AvgResp.Time$	$No.Kills$
01	28	214	370	1.7296262	87
02	26	216	444	2.0564816	129
03	27	222	486	2.1908107	125
04	25	192	431	2.2463021	104
05	25	300	738	2.4621	207

Table 7.2: Round wise results in Traditional Method

7.1.2 Log-Expo

In this approach, we are using AppsLogger to predict user interested applications using logs and AMS-v01 to do not kill user interested applications as described in section 6.3.1. Here we are using exponential weighted formula for calculating scores to the applications as described in section 6.2.2. Table 7.3 gives details on total response time(Resp.Time) in seconds, average response time (AvgResp.Time) in seconds and number of applications get killed in each round in traditional approach. No.Apps, No.Access in Table 7.3 represents number of distinct applications accessed, total number of times all applications are accessed in each round respectively.

RoundNo	*No.Apps*	*No.Access*	*Resp.Time*	*AvgResp.Time*	*No.Kills*
01	27	212	375	1.7706132	50
02	26	245	373	1.5239592	45
03	24	199	408	2.0539699	56
04	23	228	265	1.1646491	39
05	22	280	449	1.6065357	65

Table 7.3: Round wise results in Log-Expo Method

7.1.3 Log-Linear

In this approach, we are using AppsLogger to predict user interested applications using logs and AMS-v02 to do not kill user interested applications and do not kill other applications also until free memory is less than some threshold as described in section 6.3.2. Here we are using linear weighted formula for calculating scores to the applications as described in section 6.2.2. Table 7.4 gives details on total response time(Resp.Time) in seconds, average response time (AvgResp.Time) in seconds and number of applications get killed in each round in traditional approach. No.Apps, No.Access in Table 7.4 represents number of distinct applications accessed, total number of times all applications are accessed in each round respectively.

RoundNo	*No.Apps*	*No.Access*	*Resp.Time*	*AvgResp.Time*	*No.Kills*
01	27	234	267	1.1451709	35
02	26	216	284	1.3183334	19
03	24	172	285	1.6593605	38
04	22	201	342	1.7058706	32
05	21	252	483	1.9188095	58

Table 7.4: Round wise results in Log-Linear Method

7.2 Number of applications killed

Log based killing reduces number of applications gets killed in a period of time. From figure 7.1, we can clearly observe number of applications gets killed in three approaches. In traditional approach nearly 652 applications gets killed when 1144 applications were accessed, in Log-Expo approach nearly nearly 255 applications gets killed when 1164 applications were accessed, in Log-Linear approach nearly nearly 182 applications gets killed when 1075 applications were accessed. The number of applications accessed in every approach are approximately equal, But they are not exactly equal. The reason is Monkey tool randomly performs some of events on given input application, the time taken for doing those events varies based upon type of events like click buttons, loading a file, typing some text, etc. So from figure 7.1, both Log-Expo and Log-Linear methods reduced number of applications gets killed in a period of time.

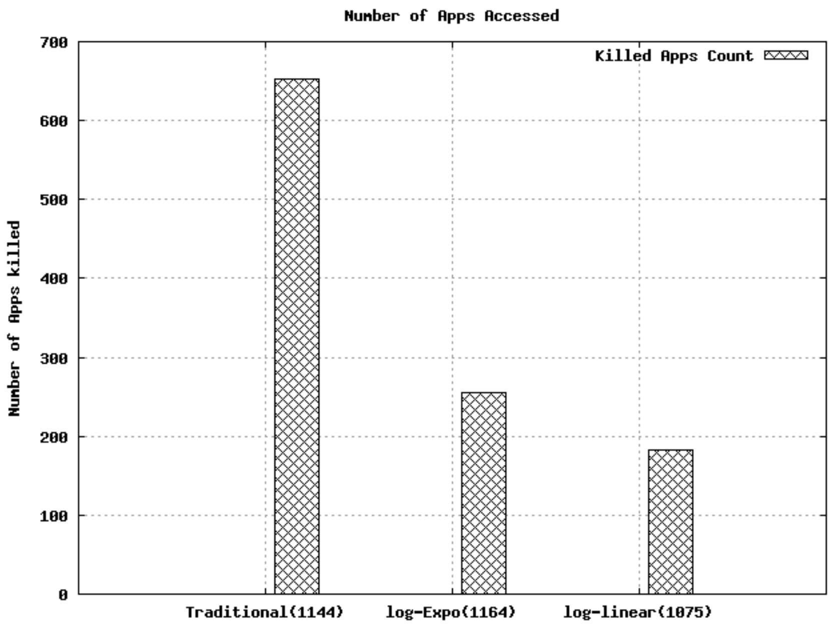


Figure 7.1: Number of Applications killed

7.2.1 Free Memory size vs Number of applications killed

In traditional approach most of the applications are gets killed, even though free memory is between 300-350MB. In log-expo approach we are not killing user interested application, instead of them we are killing infrequently used application. So always user interested applications are staying in memory till very critical low memory scenarios and interested applications are occupying more memory. Thats why most of the applications are gets killed when free memory is between 200-250MB, utilization of memory is increased in this approach. In log-linear approach we are not allowing applications to be get killed till free memory is less than 200MB, thats why most of the applications are get killed when free memory is between 150-200MB.

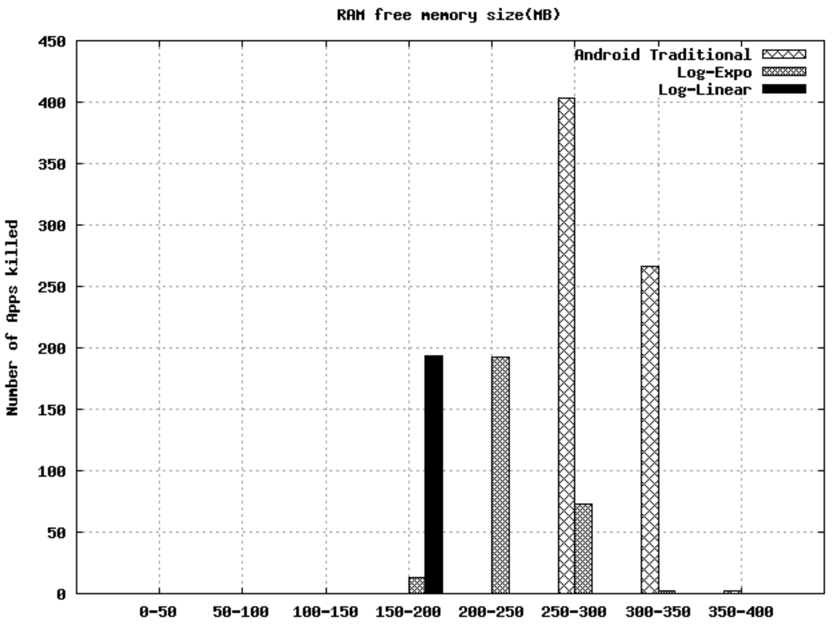


Figure 7.2: Free memory size when application get killed

7.2.2 Frequently accessed applications kill Count

Log-Expo and Log-Linear approaches are succeeded in reducing number of applications get killed. From table 7.5 we can clearly observe number of times each application get killed in three approaches. In both approaches frequently accessed applications are never get killed, because in our approaches we are not killing frequently accessed applications until very low memory scenarios(free memory < 30MB).

AppName	Traditional		Log-Expo		Log-Linear	
	No.Access	No.Kills	No.Access	No.Kills	No.Access	No.Kills
piano	57	42	63	0	54	0
gallery3d	62	35	61	0	50	0
adobe.reader	95	28	104	0	93	0
mindicator	59	36	42	0	28	0
music	41	33	39	0	39	0
proxymity	87	41	92	0	86	0
autokiller	81	27	91	0	78	0

Table 7.5: Frequently accessed applications kill count

7.2.3 Moderately accessed applications kill Count

In log-expo and log-linear approaches moderately or infrequently(as compared with frequent applications) accessed applications are get killed almost equal or little bit higher than traditional approach. From table 7.6 we can clearly observe number of times get killed in three approaches. So we can observe that moderately accessed applications get killed equal or more number of times.

AppName	Traditional		Log-Expo		Log-Linear	
	No.Access	No.Kills	No.Access	No.Kills	No.Access	No.Kills
skype	16	18	14	13	12	7
calendar	19	26	15	12	20	16
android.dev	6	6	2	1	4	3
proxydroid	21	25	22	17	18	10
animation	17	18	18	8	19	4
flashplayer	19	17	22	14	19	10

Table 7.6: Moderately accessed applications kill count

7.2.4 Apps wise Avg Response Time

In log-expo and log-linear approaches, we succeeded in reducing response time of user frequently accessed applications. From table 7.7, we can clearly observe reduced response time per single access of corresponding application in all approaches. For some of the applications like paino, mindicator, and autoKiller, we are able to reduce more than 50% of response time as compared with traditional approach.
In Log-Expo and Log-Linear approaches are causes to increase time of response time of user moderately accessed applications. From table 7.8, we can clearly observe increase in response time for all applications as compared with traditional approach.

7.3 Response Time

Response time measured as how much time taking for launching an application after user request. By protecting killing of frequently accessed applications, we are reducing the response time of frequently accessing applications. Because if we kill user frequently interacting applications, then we need to reload them in future. From table 7.9 we can observe total response time(seconds) in each round and No.access refer to number of applications are accessed in that round.
From table 7.10 we can clearly observe reduce in average response time(seconds) per accessing any kind of application. In all approaches we need to compare response times

AppName	Avg Response Time in milli sec		
	Traditional	Log-Expo	Log-Linear
piano	2217	894	699
autokiller	3331	1678	1898
adobe.reader	1728	1175	1324
gallery3d	2888	1496	826
mindicator	2106	855	752
music	3486	3272	4551
proxymity	812	141	97

Table 7.7: Frequently accessed applications Avg Response Time

AppName	Avg Response Time in milli sec		
	Traditional	Log-Expo	Log-Linear
playstroe	2336	3256	2766
skype.raider	3082	2867	3045
calendar	5611	7344	5200
android.dev	2316	3405	1872
proxydroid	3504	2711	2297
animation	1980	1416	678
flashplayer	1501	1406	1574

Table 7.8: Moderately accessed applications Avg Response Time

Round	Traditional		Log-Expo		Log-Linear	
	No.Access	Resp.Time	No.Access	Resp.Time	No.Access	Resp.Time
01	214	370	212	375	234	267
02	216	444	245	373	216	284
03	222	486	199	408	172	285
04	192	431	228	265	201	342
05	300	738	280	449	252	483

Table 7.9: Total Response Time

in round wise. Because in each round, we are giving similar set of applications as input to Monkey. In all the rounds avg response time taken in Log-Expo and Log-Linear approaches are having less response time as compared with traditional method.

Round No	Avg Response Time in sec		
	Traditional	Log-Expo	Log-Linear
01	1.7296262	1.7706132	1.1451709
02	2.0564816	1.5239592	1.3183334
03	2.1908107	2.0539699	1.6593605
04	2.2463021	1.1646491	1.7058706
05	2.4621	1.6065357	1.9188095

Table 7.10: Avg Response Time

Finally we can observe from Figure 7.1, we are able to reduce number of applications gets killed in Log-Expo, Log-Linear approaches as compared with traditional approach. Similarly from Table 7.10, Log-Expo, Log-Linear approaches are succeeded to reduce avg response time of accessing any application as compared with traditional approach.

Chapter 8

Conclusion

AMS kills applications based upon LRU and it does not consider free memory size, while killing applications, We introduced log based analysis techniques - Log-Expo and Log-Linear - to figure out most frequently and recently used applications and protect them from getting killed. Based on our experimental evaluation, we have seen our Log-Expo and Log-Linear approaches killed as low as 60% ,72% lesser number of applications, while reducing the average response time by 24%,27% respectively, when compared to the traditional approach. Finally, log based analysis for killing applications in low memory scenarios, is able to reduce number of applications get killed and average application access response time.

8.1 Future Work

We developed AppsLogger application to track user log history and to predict user interesting applications. Instead of developing an application, we can add this functionality to AMS. So we do not need to always keep this application inside memory. It is also CPU intensive application, because it has to continuously track user log history. Instead of continuously monitoring log history, we can modify AMS to record user history. AMS records user events, whenever user access any application. Finally, problems discussed in section 3.2 on OOM killer also be need to implement to achieve better fairness in killing.

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] RECEIVE_BOOT_COMPLETED, StackOverflow, Available at: *http://stackoverflow.com/questions/5051687/broadcastreceiver-not-receiving-boot-completed*

[4] Android Blog, XDIN, Available at: *http://www.androidenea.com/2009/06/android-boot-process-from-power-on.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*

[21] SQL Demo, Marakana Available at: *http://marakana.com/forums/android/examples/55.html*

[22] Writing your own ContentProvider, Think Android Available at: *http://thinkandroid.wordpress.com/2010/01/13/writing-your-own-contentprovider/*

[23] ContentProvider User Demo, Marakana Available at: *http://marakana.com/forums/android/examples/64.html*

[24] Kill Processes, StackOverflow Available at: *http://stackoverflow.com/questions/10528818/android-kill-processes-and-close-applications*

[25] Running Process, StackOverflow Available at: *http://stackoverflow.com/questions/8982143/any-way-to-check-if-a-running-process-is-a-system-process-in-android?rq=1*

[26] Constantly Monitor Logcat Available at: *http://stackoverflow.com/questions/4661234/how-to-constanly-monitor-logcat-file?rq=1*

[27] Firefox Recency Algorithm, Mozilla Developer Network Available at: *https://developer.mozilla.org/en/docs/The_Places_frecency_algorithm*

[28] Qiang Yang and Haining Henry Zhang, "Web-Log Mining for Predictive Web Caching", *IEEE Transactions on Knowledge and Data Engineering*, VOL. 15, NO. 4, July/August 2003

[29] Downloading the Source, Android Available at: *http://source.android.com/source/downloading.html*