Android Developers

Overview of Android Memory Management

The Android Runtime (ART) and Dalvik virtual machine use paging (http://en.wikipedia.org/wiki/Paging) and memory-mapping (http://en.wikipedia.org/wiki/Memory-mapped_files) (mmapping) to manage memory. This means that any memory an app modifies—whether by allocating new objects or touching mmapped pages—remains resident in RAM and cannot be paged out. The only way to release memory from an app is to release object references that the app holds, making the memory available to the garbage collector. That is with one exception: any files mmapped in without modification, such as code, can be paged out of RAM if the system wants to use that memory elsewhere.

This page explains how Android manages app processes and memory allocation. For more information about how to manage memory more efficiently in your app, see Manage Your App's Memory (https://developer.android.com/training/articles/memory.html).

In this document

See Also

Manage Your App's Memory
Investigating Your RAM Usage

Garbage collection

A managed memory environment, like the ART or Dalvik virtual machine, keeps track of each memory allocation. Once it determines that a piece of memory is no longer being used by the program, it frees it back to the heap, without any intervention from the programmer. The mechanism for reclaiming unused memory within a managed memory environment is known as *garbage collection*. Garbage collection has two goals: find data objects in a program that cannot be accessed in the future; and reclaim the resources used by those objects.

Android’s memory heap is a generational one, meaning that there are different buckets of allocations that it tracks, based on the expected life and size of an object being allocated. For example, recently allocated objects belong in the *Young generation*. When an object stays active long enough, it can be promoted to an older generation, followed by a permanent generation.

Each heap generation has its own dedicated upper limit on the amount of memory that objects there can occupy. Any time a generation starts to fill up, the system executes a garbage collection event in an attempt to free up memory. The duration of the garbage collection depends on which generation of objects it's collecting and how many active objects are in each generation.

Even though garbage collection can be quite fast, it can still affect your app's performance. You don't generally control when a garbage collection event occurs from within your code. The system has a running set of criteria for determining when to perform garbage collection. When the criteria are satisfied, the system stops executing the process and begins garbage collection. If garbage collection occurs in the middle of an intensive processing loop like an animation or during music playback, it can increase processing time. This increase can potentially push code execution in your app past the recommended 16ms threshold for efficient and smooth frame rendering.

Additionally, your code flow may perform kinds of work that force garbage collection events to occur more often or make them last longer-than-normal. For example, if you allocate multiple objects in the innermost part of a for-loop during each frame of an alpha blending animation, you might pollute your memory heap with a lot of objects. In that circumstance, the garbage collector executes multiple garbage collection events and can degrade the performance of your app.

For more general information about garbage collection, see Garbage collection (https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) .

Sharing Memory

In order to fit everything it needs in RAM, Android tries to share RAM pages across processes. It can do so in the following ways:

- Each app process is forked from an existing process called Zygote. The Zygote process starts when the system boots and loads common framework code and resources (such as activity themes). To start a new app process, the system forks the Zygote process then loads and runs the app's code in the new process. This approach allows most of the RAM pages allocated for framework code and resources to be shared across all app processes.

and also allows it to be paged out when needed. Example static data include: Dalvik code (by placing it in a pre-linked `.odex` file for direct mmapping), app resources (by designing the resource table to be a structure that can be mmapped and by aligning the zip entries of the APK), and traditional project elements like native code in `.so` files.

- In many places, Android shares the same dynamic RAM across processes using explicitly allocated shared memory regions (either with ashmem or gralloc). For example, window surfaces use shared memory between the app and screen compositor, and cursor buffers use shared memory between the content provider and client.

Due to the extensive use of shared memory, determining how much memory your app is using requires care. Techniques to properly determine your app's memory use are discussed in Investigating Your RAM Usage (https://developer.android.com/studio/profile/investigate-ram.html).

Allocating and Reclaiming App Memory

The Dalvik heap is constrained to a single virtual memory range for each app process. This defines the logical heap size, which can grow as it needs to but only up to a limit that the system defines for each app.

The logical size of the heap is not the same as the amount of physical memory used by the heap. When inspecting your app's heap, Android computes a value called the Proportional Set Size (PSS), which accounts for both dirty and clean pages that are shared with other processes—but only in an amount that's proportional to how many apps share that RAM. This (PSS) total is what the system considers to be your physical memory footprint. For more information about PSS, see the Investigating Your RAM Usage (https://developer.android.com/studio/profile/investigate-ram.html) guide.

The Dalvik heap does not compact the logical size of the heap, meaning that Android does not defragment the heap to close up space. Android can only shrink the logical heap size when there is unused space at the end of the heap. However, the system can still reduce physical memory used by the heap. After garbage collection, Dalvik walks the heap and finds unused pages, then returns those pages to the kernel using madvise. So, paired allocations and deallocations of large chunks should result in reclaiming all (or nearly all) the physical memory used. However, reclaiming memory from small allocations can be much less efficient because the page used for a small allocation may still be shared with something else that has not yet been freed.

Restricting App Memory

To maintain a functional multi-tasking environment, Android sets a hard limit on the heap size for each app. The exact heap size limit varies between devices based on how much RAM the device has available overall. If your app has reached the heap capacity and tries to allocate more memory, it can receive an `OutOfMemoryError` (https://developer.android.com/reference/java/lang/OutOfMemoryError.html).

In some cases, you might want to query the system to determine exactly how much heap space you have available on the current device—for example, to determine how much data is safe to keep in a cache. You can query the system for this figure by calling `getMemoryClass()` (https://developer.android.com/reference/android/app/ActivityManager.html#getMemoryClass()). This method returns an integer indicating the number of megabytes available for your app's heap.

Switching apps

When users switch between apps, Android keeps apps that are not foreground—that is, not visible to the user or running a foreground service like music playback— in a least-recently used (LRU) cache. For example, when a user first launches an app, a process is created for it; but when the user leaves the app, that process does *not* quit. The system keeps the process cached. If the user later returns to the app, the system reuses the process, thereby making the app switching faster.

If your app has a cached process and it retains memory that it currently does not need, then your app—even while the user is not using it— affects the system's overall performance. As the system runs low on memory, it kills processes in the LRU cache beginning with the process least recently used. The system also accounts for processes that hold onto the most memory and can terminate them to free up RAM.

> **Note:** When the system begins killing processes in the LRU cache, it primarily works bottom-up. The system also considers which processes consume more memory and thus provide the system more memory gain if killed. The less memory you consume while in the LRU list overall, the better your chances are to remain in the list and be able to quickly resume.

decides which ones can be killed, see the Processes and Threads (https://developer.android.com/guide/components/processes-and-threads.html) guide.

Follow @AndroidDev on Twitter

Follow Android Developers on Google+

Check out Android Developers on YouTube