Android Operating System

Article · May 2018

CITATIONS
3

READS
22,659

1 author:

Nimesh Kasun Ekanayake
University of Westminster
14 PUBLICATIONS **12** CITATIONS

SEE PROFILE

Android Operating System

Nimesh K. Ekanayake
Department of Information Technology,
General Sir John Kotelawala Defence University, Rathmalana, Sri Lanka
34-it-001@kdu.ac.lk

Abstract

The purpose of this article is to study the structure of the Android Operating System and view how the operating system functions inside with its continuously adding new functions. Based on the Linux kernel, Google developed an operating system for mobiles. They called it Android. The Android operating system is primarily designed for smartphone devices which implement a touch screen input interface. Not only that, it has also been developed for many devices such as smart watches (Android Wear), tablet computers and cars (Android Auto). With a new version of Android OS is being released every few months, the development of next Android takes place quickly. Android release many numerous updates that improve the operating system. With each release comes with adding new features and fixing bugs in older releases.

Keywords: process management, handling threads, cpu scheduling, memory management, file system management, references

Contents

1. Introduction

Based on the Linux kernel, Google developed an operating system for mobiles. They called it *Android*. The Android operating system is primarily designed for smartphone devices which implement a touch screen input interface. Not only that, it has also been developed for many devices such as smart watches (Android Wear), tablet computers and cars (Android Auto).

Android is known for its OS touch inputs that correspond to real-world actions such as swiping, tapping, pinching and reverse pinching.

Among many mobile operating systems, Android is the most popular operating system, which is competing with IOS for Apple devices and Windows Phones. A developer survey conducted in 2017 found that 64.8% of mobile developers use Android as their preferred platform.[1]

Since the most of Android devices are designed with a combination of open source software and proprietary software developed by Google, Android source code is also released by Google under the open source licenses. Because of open source nature of Android has enabled many to develop and distribute developer own modified version of the OS through the Android Open Source Project (ASOP). *CyanogenMod* is the most famous and widely used community firmware which is developed by Steve Kondik, aka Cyanogen, in 2009.[2]

Unlike any other operating systems, Android is developed using JAVA and it is run on virtual machines. The Android operating system features the Dalvik Runtime Machine and Android Runtime (ART) in newer versions which executes its own bytecode.[3] Dalvik is the core component and all Android user applications and application framework are written in JAVA and executed in Dalvik Runtime Machine.

With a new version of Android OS is being released every few months, the development of next Android takes place quickly. Android release many numerous updates that improve the operating system. With each release comes with adding new features and fixing bugs in older releases. Every major release is released under the name of a dessert or sugary treat. The evaluation of Android OS began in 2008, named Cup Cake and the versions to follow are Donut, Éclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean, KitKat, Lollipop, Marshmallow, Nougat and the latest, Oreo.[4] And the future version of Android will be Android P.[5]

2. Process Management

In the Android operating system, apps which are running in the background are prioritized. But apps have much freedom to run in the background than other mobile operating systems such as iOS.

2.1. Processes

A process is an instance of a program which is being executed. A process on Android operating systems can be in one of five states at any given time, from most priority to least priority.

2.1.1. Foreground Process

A foreground process is one that's needed for what the user is presently doing. numerous application parts will cause its containing method to be thought of foreground in several ways in which. A process is in the foreground if any of the subsequent conditions hold:

- It is running an Activity at the top of the screen that the user is interacting with (its *onResume()* method has been called).
- It has a BroadcastReceiver that is currently running (its *BroadcastReceiver.onReceive()* method is executing).
- It has a Service that is currently executing code in one of its callbacks (*Service.onCreate()*, *Service.onStart()*, or *Service.onDestroy()*).

There will solely ever be some such processes within the system, and these can solely be killed as a final resort if memory is very low that not even these processes will still run. Generally, at this time, the device has reached a memory paging state. Therefore, the paging action is needed to keep the program responsive.

2.1.2. Visible Process

A visible process is doing work that the user is presently alert to, thus killing it might have an apparent negative impact on the user experience. A process is considered visible under the following conditions:

- It is running an Activity that is visible to the user on-screen but not in the foreground (its *onPause()* method has been called). This may occur, for example, if the foreground Activity is displayed as a dialog that allows the previous Activity to be seen behind it.
- It has a Service that is running as a foreground service, through *Service.startForeground()* (which is asking the system to treat the service as something the user is aware of, or essentially visible to them).
- It is hosting a service that the system is using for a feature that the user is aware, such as a live wallpaper, input method service, etc.

The number of those processes running within the system is a smaller amount delimited than foreground processes, however still comparatively controlled. These processes are thought-about extraordinarily vital and cannot be killed unless doing so is needed to stay all foreground processes running.

2.1.3. Service Process

A service process is one holding a Service that has been started with the *startService()* technique. although these processes aren't directly visible to the user, they're typically doing things that the user cares concerning (such as background network information transfer or download), therefore the system can often hold such processes operating except there's not sufficient memory to retain all foreground and visual processes.

Services that are running for an extended time (like a half-hour or more) may even be downgraded in importance to let their process to release to the cached LRU list represented next. This helps avoid things wherever extremely long-running services with memory leaks or entirely distinct matters waste such lots RAM that they forestall the system by creating productive use of cached processes.

2.1.4. Cashed Process

A cached process is one that's not presently required, therefore the system is unengaged to kill it as desired once memory is required elsewhere. in an ordinarily behaving system, these are the sole processes concerned in memory management: a well running system can have multiple cached processes continuously accessible (for the additional efficient shift between applications) and often kill the oldest ones as required. solely in terribly vital (and undesirable) things will the system get to a degree wherever all cached processes are killed, and it should begin killing service processes.

These processes usually hold one or additional Activity instances that aren't presently visible to the user (the *onStop()* method has been called and returned). Provided they implement their Activity life-cycle properly, once the system kills such processes it'll not impact the user's experience when returning to that app: it will restore the previously saved state once the associated activity is recreated in an exceedingly new process.

These processes are unbroken in a pseudo-LRU list, wherever the last process on the list is that the initial killed to reclaim memory. the precise policy of ordering on this list is an implementation detail of the platform, however, typically it'll try and keep additional helpful processes (one hosting the user's home application, the last activity they saw, etc.) before different kinds of processes. different policies for killing processes might also be applied: hard limits on the number of processes allowed, limits on the quantity of your time a process will keep frequently cached, etc.

2.1.5. Empty Process

This empty process would not be including any app data hereafter. They are going to keep all around for the caching aims to make speed launch hereafter. but it will be killed by the system if needed.

Generally, only background and empty processes are used to killed by the system. so, the user experience would keep not influencing. Android needed kills apps. Because the memory usage goes high level. but usually Android does not try to kill the apps

Processes could be including the multiple threads. Like Linux based systems. Most Android applications execute the thread to differ from the UI from i I/O operations and long-running calculations methods.

2.2. Android Automatically Manages Processes

We don't need a task killer on Android because it is capable of automatically managing processes.

Android will start killing the least important processes first when it needs more system resources. If the device is running low on memory, Android starts to kill empty and background processes. Android wisely uses device's RAM for caching apps and other data. Because there is no point in leaving device's RAM empty.[6]

2.3. Android Apps Can Start in Response to Events

In response to events, Android apps can be started. *For example, a developer can code an app to automatically start at startup. And run a service in the background.* Apps can start as a response to different other events. Such as when taking a picture, when data connection changes, and so on. This feature allows apps to perform actions as a response to events without continuously running in the background.[6]

2.4. Managing Processes

There is no need to manage processes manually. But there a few methods to manage processes manually. We can use the multitasking menu on versions of Android 4.0 or later to do some basic process management. To access it we can double-tap or long press the Home button. Or tap the dedicated multitasking button on Nexus devices.

Apps showed in the menu are most likely in *background process* state. We can terminate them by swiping an app to left or right. Then it will be removed from device's memory.

If it is a misbehaving app, we can kill that app through the settings screen. For this go to the **Settings**, tap **Apps**, tap on the misbehaving app and use the *Force stop* button.[6]

3. Handling Threads

The threads are also an important part of the Android OS. Each process has a separate thread by default. When an application is launched, the system creates a thread of execution for the app and is known as the main thread. This thread is very important because it oversees dispatching events including drawing events. The main thread is also called the UI thread. The system does not create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched for that thread.

Now there are situations when an application performs intensive work in response to the user interaction, the single thread model i.e. the UI thread performs poorly and we do not get the desired output, also the UI thread is not thread-safe, so we use another type of thread called as worker threads who work along with the main UI thread. However, there are some limitations on the worker threads as:

- The worker threads can't block the UI thread.[7]
- The worker threads only run in the background and can't update the application UI. the application UI can be updated by only the main thread.[7]

The threads in android are generally implemented with the help of pthread libraries. The pthread (POSIX thread) libraries are natively implemented in C/C++.

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, then a new name is generated for it.

3.1. Worker Threads

Because of the single thread model represented above, it is important to the responsiveness of your application's UI that you simply don't block the UI thread. If you've got operations to perform that don't seem to be fast, you must ensure to do them in separate threads ("background" or "worker" threads).

Because of the single thread model described above, it's vital to the responsiveness of your application's UI that you do not block the UI thread. If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).

For example, below is some code for a click listener that downloads an image from a separate thread and displays it in an ImageView:

```
public void onClick(View v) {
   new Thread(new Runnable() {
      public void run() {
         Bitmap b =
loadImageFromNetwork("http://example.com/imag
e.png");
         mImageView.setImageBitmap(b);
      }
   }).start();
}
```

At first, this looks to work fine, because of it creates a replacement thread to handle the network operation. However, it violates the second rule of the single-threaded model: don't access the android UI toolkit from outside the UI thread—this sample modifies the ImageView from the worker thread rather than the UI thread. this will lead to undefined and unexpected behavior, which may be tough and long to trace down.

To fix this drawback, Android offers many ways in which to access the UI thread from alternative threads. Here may be a list of strategies that may help:

- *Activity.runOnUiThread(Runnable)*
- *View.post(Runnable)*
- *View.postDelayed(Runnable, long)*

For example, you can fix the above code by using the *View.post(Runnable)* method:

```
public void onClick(View v) {
   new Thread(new Runnable() {
      public void run() {
         final Bitmap bitmap =
loadImageFromNetwork("http://example.com/imag
e.png");
         mImageView.post(new Runnable() {
            public void run() {
```

```
                mImageView.setImageBitmap(bitmap)
;
            }
        });
    }
}).start();
}
```

Now, this implementation is thread-safe. Because the network operation is performed from a separate thread. And it's happening while the ImageView is handled from the UI thread.

However, because the complexness of the operation grows, this sort of code will get sophisticated and tough to keep up. To handle additional complicated interactions with a worker thread, you would possibly think about using a Handler in your worker thread, to process messages delivered from the UI thread. maybe the most effective answer, though, is to increase the AsyncTask class, that simplifies the execution of worker thread tasks that require interacting with the UI.

Using AsyncTask

AsyncTask permits you to perform asynchronous work on your user interface. It performs the interference operations in a worker thread and then publishes the results on the UI thread, while not requiring you to handle threads and/or handlers yourself.[8]

To use it, it is required to subclass AsyncTask and implement the *doInBackground()* callback method, that operates in a pool of background threads. To update your UI, you must implement *onPostExecute()*, that delivers the result from *doInBackground()* and runs within the UI thread, therefore you'll be able to safely update your UI. you'll be able to then run the task by calling *execute()* from the UI thread.

For example, you can implement the previous example using AsyncTask this way:

```
public void onClick(View v) {
    new
DownloadImageTask().execute("http://example.co
m/image.png");
}

private class DownloadImageTask extends
AsyncTask<String, Void, Bitmap> {
    /** The system calls this to perform work in a
worker thread and
     * delivers it the parameters given to
AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls)
{
        return loadImageFromNetwork(urls[0]);
    }

    /** The system calls this to perform work in the
UI thread and delivers
     * the result from doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

Now the UI is safe, and the code is less complicated, because of it separates the work into the part that must be done on a worker thread and the part that should be done on the UI thread.

You should read the AsyncTask reference for a full understanding of how to use this class, however here is a fast summary of how it works:

- You can specify the sort of the parameters, the progress values, and the final value of the task, using generics.
- The method *doInBackground()* executes automatically on a worker thread.
- *onPreExecute()*, *onPostExecute()*, and *onProgressUpdate()* are all invoked on the UI thread.
- The value returned by *doInBackground()* is sent to *onPostExecute()*.
- You can call *publishProgress()* at any time in *doInBackground()* to execute *onProgressUpdate()* on the UI thread.
- You can cancel the task at any time, from any thread.

3.2. Thread-Safe Methods

In some situations, the methods you implement could be called from over one thread, and so should be written to be thread-safe.

This is primarily true for methods which will be called remotely—such as methods in a bound service. once a call on a method, it is executed in an IBinder. Which originates within the same process, in which the IBinder is running. Then the method is dead in the caller's thread. However, once the call originates in another process, the method is dead in a very thread chosen from a pool of threads that the system maintains within the same process as the IBinder (it's not dead within the UI thread of the process). for example, whereas a service's *onBind()* method would be called from the UI thread of the service's process, methods implemented within the

object that *onBind()* returns (for example, a subclass that implements RPC methods) would be called from threads within the pool. because of a service will have over one client, over one pool thread will interact the same IBinder method at the same time. IBinder methods should be executed to be thread-safe.

Similarly, a content provider will receive data requests that originate in alternative processes. though the *ContentResolver* and *ContentProvider* classes hide the details of how the interposes communication is managed, ContentProvider methods that answer those requests—the methods *query()*, *insert()*, *delete()*, *update()*, and *getType()*—are called from a pool of threads within the content provider's process, not the UI thread for the process. Because of these methods could be called from any number of threads at the same time, they too should be enforced to be thread-safe.

4. CPU Scheduling

4.1. Normal Scheduling

The Android operating system depends on Linux. And it uses the scheduling mechanisms of Linux kernels for deciding scheduling methods. This can be also true for Java code and threads.

The Linux's time-sliced scheduling policy combines static and dynamic priorities. Processes may be given an initial priority from 19 to -20 (very low to terribly high priority). This priority can assure that higher priority processes can get a lot of CPU time when once required. These levels are however dynamic, low-level priority tasks that don't consume their CPU time can find their dynamic priority increased. This dynamic behavior result is an overall higher responsiveness.

In terms of dynamic priorities, it's ensured that lower priority processes can continuously have a lower dynamic priority than processes with real-time priorities.

Android uses two totally different mechanisms once scheduling the Linux kernel to perform process level scheduling.

4.2. Real-time Scheduling

SCHED_FIFO and SCHED_RR are the two real-time scheduling policies, provided by usual Linux kernel. SCHED_FIFO is known as the main real-time strategy. It performs a first-in, first-out scheduling algorithmic program. Once a SCHED_FIFO task starts running, it continues to run till it voluntarily yields the processor, blocks or is acquired by a higher-priority real-time task. it's no time slices. All alternative tasks of lower priority won't be scheduled till it abandons the CPU. Two equal-priority SCHED_FIFO tasks don't acquire one another. SCHED_RR is comparable to SCHED_FIFO, except that such tasks are assigned time slices based on their priority and run till they exhaust their time slice. Tasks identified as non-real-time tasks use the SCHED_NORMAL scheduling policy (older kernels had a policy called SCHED_OTHER).[9]

The default android kernel is organized to allow group scheduling of real-time processes and the file system to manage this is mounted under /dev/cpuctl.

Android operating system practices two entirely different scheduling classes (using Linux cgroups). That is bg_non_interactive and default (foreground). The configuration is that bg_non_interactive is low priority and might most utilize ~5% of the CPU (including all background tasks) and foreground ~95%. Foreground means that either an Activity or a service that's started foreground.

At startup, services are operating in bg_non_interactive except they have been elevated to foreground scheduling group using startForeground (HMI applications are always set to foreground).

4.3. Binder and Priorities

The binder mechanism additionally propagates priorities. that's the binder process called will run with an equivalent priority as the caller.[10]

The Binder framework uses its own language to call facilities and components. This section summarizes the foremost vital terms.

- *Binder* - This term is used ambiguously. The Binder refers to the general Binder design, whereas a Binder refers to a specific implementation of a Binder interface.
- *Binder Object* – It is an instance of a class that implements the Binder interface. A Binder object will implement multiple Binders.
- *Binder Protocol* - The Binder middleware uses a very low-level protocol to communicate with the driver.
- *IBinder* - Interface A Binder interface could be a well-defined set of methods, properties, and events that a Binder will implement. it's typically delineated by AIDL1 language.

- *Binder Token* - A numeric value that unambiguously identifies a Binder.

4.4. JVM Thread and Process Scheduling

A group of UNIX processes running may be seen in an Android system. Some are native processes. However, several processes are going to process that run a Java virtual machine. These processes typically are going to be multi-threaded, all android threads are native pthreads (no green threads). There are two ways in which to alter the priority handling one by Calling *Thread.setPriority* that's a part of the quality Java API and contains a value from *MIN_PRIORITY(1)* to *MAX_PRIORITY(10)*.As all threads are pthreads these priorities are going to be mapped to Linux process priorities (MIN_PRIORITY being 19 and MAX_PRIORITY -8).

```
-Combined dalvik/vm/Thread.c and -
frameworks/base/include/utils/threads.h

Thread.priority ,   Java name    , Android property
name            , Unix priority

1               MIN_PRIORITY
ANDROID_PRIORITY_LOWEST,            19

2
ANDROID_PRIORITY_BACKGROUND + 6
16

3
ANDROID_PRIORITY_BACKGROUND + 3
13

4
ANDROID_PRIORITY_BACKGROUND
10

5               NORM_PRIORITY
ANDROID_PRIORITY_NORMAL             0

6
ANDROID_PRIORITY_NORMAL - 2         -2

7
ANDROID_PRIORITY_NORMAL - 4         -4

8
ANDROID_PRIORITY_URGENT_DISPLAY + 3
-5

9
ANDROID_PRIORITY_URGENT_DISPLAY + 2
-6

10              MAX_PRIORITY
ANDROID_PRIORITY_URGENT_DISPLAY
-8
```

Calling android.os.Process.setThreadPriority() is the second way to set priorities. This enables to set the priority to higher priorities. For that it should be declared in your AndroidManifest and call process.setThreadPriority(Process.myTid(), Process.THREAD_PRIORITY_URGENT_DISPLAY).

```
frameworks/base/include/utils/threads.h
   ANDROID_PRIORITY_LOWEST        = 19,


   /* use for background tasks */
   ANDROID_PRIORITY_BACKGROUND    =
10,


   /* most threads run at normal priority */
   ANDROID_PRIORITY_NORMAL        =  0,


   /* threads currently running a UI that the user is
interacting with */
   ANDROID_PRIORITY_FOREGROUND    = -
2,


   /* the main UI thread has a slightly more
favorable priority */
   ANDROID_PRIORITY_DISPLAY       = -4,
   /* ui service treads might want to run at a urgent
display (uncommon) */
   ANDROID_PRIORITY_URGENT_DISPLAY
=  -8,


   /* all normal audio threads */
   ANDROID_PRIORITY_AUDIO         = -16,


   /* service audio threads (uncommon) */
   ANDROID_PRIORITY_URGENT_AUDIO  =
-19,
```

Nimesh K. Ekanayake
34-it-001@kdu.ac.lk
May 20, 2018

```
   /* should never be used in practice. regular
process might not
    * be allowed to use this level */
   ANDROID_PRIORITY_HIGHEST       = -20,



   ANDROID_PRIORITY_DEFAULT       =
ANDROID_PRIORITY_NORMAL,
   ANDROID_PRIORITY_MORE_FAVORABLE
= -1,
   ANDROID_PRIORITY_LESS_FAVORABLE
= +1,
```

*The GC runs (per process) with all the threads suspended.

5. Memory Management

The Android Runtime (ART) and Dalvik virtual machine use paging and memory-mapping (mmapping) to manage memory.[11] This implies that any memory an app modifies—whether by allocating new objects or touching mmapped pages—remains resident in RAM and can't be paged out. The sole way to unleash memory from an app is to unleash object references that the app holds. And making the memory accessible to the garbage collector.[12] that's with one exception: any files mmapped in without modification, like code, may be paged out of RAM if the system desires to use that memory elsewhere.

5.1. Garbage collection

A managed memory atmosphere, just like the ART or Dalvik virtual machine, keeps track of every memory allocation. Once it determines that a piece of memory isn't any longer getting used by the program, it frees it back to the heap, with none intervention from the programmer. The mechanism for reclaiming unused memory inside a managed memory atmosphere is understood as garbage collection. garbage collection has 2 goals:

1. Notice data objects in a program that can't be accessed in the future.[11]
2. Reclaim the resources employed by those objects.[11]

Android's memory heap is a generational one, which means that there are totally different buckets of allocations that it tracks, based on the expected life and size of an object being allocated. as an example, recently allocated objects belong in the Young generation. once an object stays active long enough, it will be promoted to an older generation, followed by a permanent generation.

Each heap generation has its own dedicated higher limit on the quantity of memory that objects there can occupy. Any time a generation starts to refill, the system executes a garbage collection event to release memory. The period of the garbage collection depends on which generation of objects it's collecting and the way several active objects are in every generation.

Even though garbage collection may be quite quick, it can still influence your app's performance. You don't usually control once a garbage collection event happens from inside your code. The system contains a running set of criteria for deciding when to perform garbage collection. once the factors are satisfied, the system stops executing the process and begins garbage collection. If garbage collection happens within the middle of an intensive processing loop like an animation or throughout music playback, it will increase processing time. This increase will probably push code execution in your app past the suggested 16ms threshold for efficient and smooth frame rendering.

Additionally, your code flow could perform types of work that force garbage collection events to occur a lot of usually or make them last longer-than-normal. as an example, if you assign multiple objects within the innermost part of a for-loop throughout every frame of an alpha mixing animation, you would possibly pollute your memory heap with plenty of objects. in this circumstance, the garbage collector executes multiple garbage collection events and might degrade the performance of your app.

5.2. Share memory

To suit everything, it wants in RAM, android tries to share RAM pages across processes. It will do so within the following ways:

- Each app process is forked from an existing process referred to as zygote. The zygote process starts once the system boots and loads common framework code and resources (such as activity themes). to begin a replacement app method, the system forks the cell method then masses and runs the app's code within the new method. This approach permits most of the RAM pages allocated for framework code and resources to be shared across all app processes.
- Most static data are mmapped into a process. this method permits data to be shared between processes, and additionally

permits it to be paged out once required. Example static data include: Dalvik code (by inserting it in a pre-linked *.odex* file for direct mmapping), app resources (by planning the resource table to be a structure that may be mmapped and by positioning the zip entries of the APK), and traditional project components like native code in *.so* files.

- In several places, android shares a similar dynamic RAM across processes using expressly allocated shared memory regions (either with ashmem or gralloc). as an example, window surfaces use shared memory between the app and screen compositor, and cursor buffers use shared memory between the content supplier and consumer.

5.3. Allocate and reclaim app memory

The Dalvik heap is constrained to one virtual memory range for every app process. This defines the logical heap size, which might grow because it must however solely up to a limit that the system defines for every app.

The logical size of the heap isn't a similar because of the quantity of physical memory utilized by the heap. once inspecting your app's heap, android computes a value referred to as the Proportional Set Size (PSS), which accounts for both dirty and clean pages that are shared with alternative processes—but solely in a quantity that is proportional to what percentage apps share that RAM. This (PSS) total is what the system considers to be your physical memory footprint.

The Dalvik heap doesn't compact the logical size of the heap, that means that Android doesn't defragment the heap to shut up space. android will solely shrink the logical heap size once there's unused space at the end of the heap. However, the system can still cut back physical memory used by the heap. after garbage collection, Dalvik walks the heap and finds unused pages and after that it returns those pages to the kernel using madvise. So, paired allocations and deallocations of huge chunks ought to lead to reclaiming all (or nearly all) the physical memory used. However, recovering memory from tiny allocations may be a lot of less effective because of the page used for a small allocation should be shared with something else that has not yet been freed.

5.4. Restrict app memory

To maintain a useful multi-tasking environment, Android sets a tough limit on the heap size for every app. the precise heap size limit varies between devices based on how much RAM the device has accessible overall. If your app has reached the heaped capacity and tries to assign additional memory, it will receive an *OutOfMemoryError*.

In some cases, you may need to query the system to determine specifically how much heap space you have got accessible on this device—for example, to see how much data is safe to stay in a cache. If you wish you can query the system concerning this structure. You have to do it by executing *getMemoryClass()*. This method returns an integer indicating the number of megabytes accessible for your app's heap.

5.5. Switch apps

When users switch among apps, Android keeps apps that are not foreground. That is, not noticeable to the user or running a foreground service like music playback— in a least-recently-used (LRU) cache. as an example, once a user first launches an app, a process is formed for it; however, once the user leaves the app, that process doesn't quit. The system keeps the process cached. If the user later returns to the app, the system reuses the process, thereby creating the app switching quicker.

If your app includes a cached process and it holds the memory that it presently doesn't require, then your app (even whereas the user isn't using it) affects the system's overall performance. because the system runs low on memory, it kills processes within the LRU cache starting with the process least recently used. The system additionally accounts for processes that hold onto the most available memory and might terminate them to free up RAM.

6. File System Management

Android provides many choices for you to save lots of your app data. the answer you select depends on your specific wants, like how much space your data needs, what kind of data you would like to store, and whether the data ought to be personal to your app or accessible to alternative apps and the user.

Various data storage choices offered on Android:

- **Internal file storage:** Store app-private files on the device file system.
- **External file storage:** Store files on the shared external file system. this is often typically for shared user files, like photos.
- **Shared preferences:** Store personal primitive data in key-value pairs.

- **Databases:** Store structured data in a personal database.

Except for some kinds of files on external storage, of these choices are supposed for app-private data—the data isn't naturally accessible to alternative apps. If you would like to share files with alternative apps, you ought to use the *FileProvider* API.

If you would like to reveal your app's data to alternative apps, you'll be able to use a *ContentProvider*. Content suppliers offer you full control of what read/write access is accessible to alternative apps, no matter the storage medium you have chosen for the data (though it's always a database).

6.1. Internal storage

By default, files saved to the internal storage are non-public to your app, and alternative apps will not access them (nor can the user, unless they have root access). This makes internal storage a decent place for internal app data that the user does not ought to directly access. The system provides a non-public directory on the file system for every app wherever you'll be able to organize any files your app desires.

When the user uninstalls your app, the files saved on the internal storage are removed. thanks to this behavior, you must not use internal storage to save anything the user expects to persist independently of your app. as an example, if your app permits users to capture photos, the user would expect that they will access those photos even after they uninstall your app. thus you must instead save those sorts of files to the public external storage.

6.1.1. Internal cache files

If you want to keep some data temporarily, instead of store it persistently, you must use the special cache directory to avoid wasting the data. every app contains a private cache directory specifically for these types of files. once the device is running low on space of internal storage, Android may delete these cache files to recover space. However, you must not rely on the system to clean up these files for you. So, you must continuously maintain the cache files yourself. And keep it within a reasonable limit of space obsessed (like 1MB). once the user uninstalls your app, these files are removed.

6.2. External storage

Every android device supports a shared "external storage" space that you just will use to save lots of files. This space is termed external because it is not a bound to be accessible—it may be a storage space that users can mount to a computer as an external storage device, and it'd even be physically removable (such as an SD card). Files saved to the external storage are world-readable and may be changed by the user once they enable USB mass storage to transfer files on a pc.

So, before you commit to access a file in external storage in your app, you must check for the availability of the external storage directories in addition as the files you're attempting to access.

Most often, you must use external storage for user data that ought to be accessible to different apps and saved even if the user uninstalls your app, like captured photos or downloaded files. The system provides standard public directories for these sorts of files, that the user has one location for all their photos, ringtones, music, and such.

You can additionally save files to the external storage in an app-specific directory that the system deletes once the user uninstalls your app. This can be a helpful alternative to internal storage if you wish more space, however the files here are not bound to be accessible because of the user would possibly take away the storage SD card. and, the files are still world readable; they are simply saved to a location that is not shared with different apps.

6.3. Shared preferences

If you do not ought to store lots of data and it does not need structure, you ought to use *SharedPreferences*. The *SharedPreferences* apis permit you to read and write persistent key-value pairs of primitive data types: Booleans, floats, ints, longs, and strings.

The key-value pairs are written to XML files that persist across user sessions, even though your app is killed. you'll manually specify a name for the file or use per-activity files to avoid wasting your data.

The API name "shared preferences" may be a bit misleading because of the API isn't strictly for saving "user preferences," like what ringtone a user has chosen. you'll use SharedPreferences to avoid wasting any quite easy data, like the user's high score. However, if you are doing need to save lots of user preferences for your app, then you ought to scan a way to *create a settings UI*, which uses *PreferenceActivity* to create a settings screen and automatically persist the user's settings.

6.4. Databases

Android provides full support for SQLite databases. Any database you produce is accessible only by your

app. However, rather than using SQLite APIs directly, we suggest that you just produce and interact along with your databases with the room persistence library.

The Room library provides an object-mapping abstraction layer that enables fluent database access while harnessing the total power of SQLite.

Although you'll still save data directly with SQLite, the SQLite APIs are low-level and need a good deal of time and effort to use. For example:

- There is no compile-time verification of raw SQL queries.
- As your schema changes, you need to update the affected SQL queries manually. This process may be time overwhelming and error-prone.
- You need to write down numerous boilerplate code to convert between SQL queries and Java data objects.

The Room persistence library takes care of those considerations for you while providing an abstraction layer over SQLite.

6.4.1. Database debugging

The android SDK includes a sqlite3 database tool that permits you to browse table contents, run SQL commands, and perform alternative helpful functions on SQLite databases.

References

[1] "Editorial: Mobile and the 2017 Developer Survey Results," *SitePoint*, 29-Mar-2017. .

[2] rii Degeler and T. E. at S. • 10 min read, "Open Source Android ROMs You Can Use For Your Device," *Stanfy | San Francisco Mobile App Developers*. [Online]. Available: https://stanfy.com/blog/open-source-android-roms-you-can-use-for-your-device/. [Accessed: 29-Apr-2018].

[3] "ART vs Dalvik - introducing the new Android runtime in KitKat." [Online]. Available: https://infinum.co/the-capsized-eight/art-vs-dalvik-introducing-the-new-android-runtime-in-kit-kat. [Accessed: 29-Apr-2018].

[4] "Android - History," *Android*. [Online]. Available: https://www.android.com/history/. [Accessed: 29-Apr-2018].

[5] "Android P: Top 6 things you need to know!," *Android Central*, 07-Mar-2018. [Online]. Available: https://www.androidcentral.com/android-p. [Accessed: 29-Apr-2018].

[6] C. Hoffman, "How Android Manages Processes." [Online]. Available: https://www.howtogeek.com/161225/htg-explains-how-android-manages-processes/. [Accessed: 29-Apr-2018].

[7] "Managing Threads and Custom Services | CodePath Android Cliffnotes." [Online]. Available: https://guides.codepath.com/android/managing-threads-and-custom-services. [Accessed: 30-Apr-2018].

[8] "Using AsyncTask in Android App Development," *Hiring | Upwork*, 20-Jan-2017. [Online]. Available: https://www.upwork.com/hiring/mobile/why-you-should-use-asynctask-in-android-development/. [Accessed: 30-Apr-2018].

[9] "Real-Time group scheduling." [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt. [Accessed: 01-May-2018].

[10] "OpenBinder." [Online]. Available: http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html. [Accessed: 01-May-2018].

[11] "Android Memory Management - in detail for all those questions regarding task killing," *XDA Developers*. [Online]. Available: https://forum.xda-developers.com/showthread.php?t=904023. [Accessed: 07-May-2018].

[12] "Memory Management in Android," *Welcome to Mobile World !!!*, 05-Jul-2010. .

Nimesh K. Ekanayake
34-it-001@kdu.ac.lk
May 20, 2018