Android Developers

Threading Performance

Making adept use of threads on Android can help you boost your app's performance. This page discusses several aspects of working with threads: working with the UI, or main, thread; the relationship between app lifecycle and thread priority; and, methods that the platform provides to help manage thread complexity. In each of these areas, this page describes potential pitfalls and strategies for avoiding them.

In this document

Main Thread

When the user launches your app, Android creates a new Linux process (https://developer.android.com/guide/components/fundamentals.html) along with an execution thread. This **main thread,** also known as the UI thread, is responsible for everything that happens onscreen. Understanding how it works can help you design your app to use the main thread for the best possible performance.

Internals

The main thread has a very simple design: Its only job is to take and execute blocks of work from a thread-safe work queue until its app is terminated. The framework generates some of these blocks of work from a variety of places. These places include callbacks associated with lifecycle information, user events such as input, or events coming from other apps and processes. In addition, app can explicitly enqueue blocks on their own, without using the framework.

Nearly any block of code your app executes (https://www.youtube.com/watch?v=qk5F6Bxqhr4&index=1&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE) is tied to an event callback, such as input, layout inflation, or draw. When something triggers an event, the thread where the event happened pushes the event out of itself, and into the main thread's message queue. The main thread can then service the event.

While an animation or screen update is occurring, the system tries to execute a block of work (which is

responsible for drawing the screen) every 16ms or so, in order to render smoothly at 60 frames per second (https://www.youtube.com/watch?v=CaMTIgxCSqU&index=62&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE). For the system to reach this goal, the UI/View hierarchy must update on the main thread. However, when the main thread's messaging queue contains tasks that are either too numerous or too long for the main thread to complete the update fast enough, the app should move this work to a worker thread. If the main thread cannot finish executing blocks of work within 16ms, the user may observe hitching, lagging, or a lack of UI responsiveness to input. If the main thread blocks for approximately five seconds, the system displays the *Application Not Responding* (https://developer.android.com/training/articles/perf-anr.html) (ANR) dialog, allowing the user to close the app directly.

Moving numerous or long tasks from the main thread, so that they don't interfere with smooth rendering and fast responsiveness to user input, is the biggest reason for you to adopt threading in your app.

Threading and UI Object References

By design, Android View objects are not thread-safe (https://www.youtube.com/watch?v=tBHPmQQNiS8&index=3&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE). An app is expected to create, use, and destroy UI objects, all on the main thread. If you try to modify or even reference a UI object in a thread other than the main thread, the result can be exceptions, silent failures, crashes, and other undefined misbehavior.

Issues with references fall into two distinct categories: explicit references and implicit references.

Explicit references

Many tasks on non-main threads have the end goal of updating UI objects. However, if one of these threads accesses an object in the view hierarchy, application instability can result: If a worker thread changes the properties of that object at the same time that any other thread is referencing the object, the results are undefined.

For example, consider an app that holds a direct reference to a UI object on a worker thread. The object on the worker thread may contain a reference to a `View` (https://developer.android.com/reference/android/view/View.html); but before the work completes, the `View` (https://developer.android.com/reference/android/view/View.html) is removed from the view hierarchy. When these two actions happen simultaneously, the reference keeps the `View` (https://developer.android.com/reference/android/view/View.html) object in memory and sets properties on it. However, the user never sees this object, and the app deletes the object once the reference to it is gone.

In another example, `View` (https://developer.android.com/reference/android/view/View.html) objects contain references to the activity that owns them. If that activity is destroyed, but there remains a threaded block of work that references it—directly or indirectly—the garbage collector will not collect the activity until that block of work finishes executing.

This scenario can cause a problem in situations where threaded work may be in flight while some activity lifecycle event, such as a screen rotation, occurs. The system wouldn't be able to perform garbage collection until the in-flight work completes. As a result, there may be two `Activity` (https://developer.android.com/reference/android/app/Activity.html) objects in memory until garbage collection can take place.

With scenarios like these, we suggest that your app not include explicit references to UI objects in threaded work tasks. Avoiding such references helps you avoid these types of memory leaks, while also steering clear of threading contention.

In all cases, your app should only update UI objects on the main thread. This means that you should craft a negotiation policy that allows multiple threads to communicate work back to the main thread, which tasks the topmost activity or fragment with the work of updating the actual UI object.

Implicit references

A common code-design flaw with threaded objects can be seen in the snippet of code below:

```
public class MainActivity extends Activity {
  // ….
  public class MyAsyncTask extends AsyncTask<Void, Void, String>   {
    @Override protected String doInBackground(Void... params) {...}
    @Override protected void onPostExecute(String result) {...}
  }
}
```

The flaw in this snippet is that the code declares the threading object `MyAsyncTask` as a non-static inner class of some activity. This declaration creates an implicit reference to the enclosing `Activity` (https://developer.android.com/reference/android/app/Activity.html) instance. As a result, the object contains a reference to the activity until the threaded work completes, causing a delay in the destruction of the referenced activity. This delay, in turn, puts more pressure on memory.

A direct solution to this problem would be to define your overloaded class instances either as static classes, or in their own files, thus removing the implicit reference.

Another solution is to declare the `AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) object as a static nested class. Doing so eliminates the implicit reference problem

because of the way a static nested class differs from an inner class: An instance of an inner class requires an instance of the outer class to be instantiated, and has direct access to the methods and fields of its enclosing instance. By contrast, a static nested class does not require a reference to an instance of enclosing class, so it contains no references to the outer class members.

```
public class MainActivity extends Activity {
  // ….…
  static public class MyAsyncTask extends AsyncTask<Void, Void, String>   {
    @Override protected String doInBackground(Void... params) {...}
    @Override protected void onPostExecute(String result) {...}
  }
}
```

Threading and App and Activity Lifecycles

The app lifecycle can affect how threading works in your application. You may need to decide that a thread should, or should not, persist after an activity is destroyed. You should also be aware of the relationship between thread prioritization and whether an activity is running in the foreground or background.

Persisting threads

Threads persist past the lifetime of the activities that spawn them. Threads continue to execute, uninterrupted, regardless of the creation or destruction of activities. In some cases, this persistence is desirable.

Consider a case in which an activity spawns a set of threaded work blocks, and is then destroyed before a worker thread can execute the blocks. What should the app do with the blocks that are in flight?

If the blocks were going to update a UI that no longer exists, there's no reason for the work to continue. For example, if the work is to load user information from a database, and then update views, the thread is no longer necessary.

By contrast, the work packets may have some benefit not entirely related to the UI. In this case, you should persist the thread. For example, the packets may be waiting to download an image, cache it to disk, and update the associated `View` (https://developer.android.com/reference/android/view/View.html) object. Although the object no longer exists, the acts of downloading and caching the image may still be helpful, in case the user returns to the destroyed activity.

Managing lifecycle responses manually for all threading objects can become extremely complex. If you don't manage them correctly, your app can suffer from memory contention and performance issues. Loaders (https://developer.android.com/guide/components/loaders.html) are one solution to this problem. A loader facilitates asynchronous loading of data, while also persisting information through configuration changes. For more information, see `AsyncTaskLoader` (https://developer.android.com/reference/android/content/AsyncTaskLoader.html).

Thread priority

As described in Processes and the Application Lifecycle (https://developer.android.com/guide/topics/processes/process-lifecycle.html), the priority that your app's threads receive depends partly on where the app is in the app lifecycle. As you create and manage threads in your application, it's important to set their priority so that the right threads get the right priorities at the right times. If set too high, your thread may interrupt the UI thread and RenderThread, causing your app to drop frames. If set too low, you can make your async tasks (such as image loading) slower than they need to be.

Every time you create a thread, you should call `setThreadPriority()` (https://developer.android.com/reference/android/os/Process.html#setThreadPriority(int, int)). The system's thread scheduler gives preference to threads with high priorities, balancing those priorities with the need to eventually get all the work done. Generally, threads in the foreground group get about 95% (https://www.youtube.com/watch?v=NwFXVsM15Co&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE&index=9) of the total execution time from the device, while the background group gets roughly 5%.

The system also assigns each thread its own priority value, using the `Process` (https://developer.android.com/reference/android/os/Process.html) class.

By default, the system sets a thread's priority to the same priority and group memberships as the spawning thread. However, your application can explicitly adjust thread priority by using `setThreadPriority()` (https://developer.android.com/reference/android/os/Process.html#setThreadPriority(int, int)).

The `Process` (https://developer.android.com/reference/android/os/Process.html) class helps reduce complexity in assigning priority values by providing a set of constants that your app can use to set thread priorities. For example, `THREAD_PRIORITY_DEFAULT` (https://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_DEFAULT) represents the default value for a thread. Your app should set the thread's priority to `THREAD_PRIORITY_BACKGROUND` (https://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_BACKGROUND) for threads that are executing less-urgent work.

Your app can use the `THREAD_PRIORITY_LESS_FAVORABLE` (https://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_LESS_FAVORABLE) and `THREAD_PRIORITY_MORE_FAVORABLE`

(https://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_MORE_FAVORABLE) constants as incrementers to set relative priorities. For a list of thread priorities, see the THREAD_PRIORITY (https://developer.android.com/reference/android/os/Process.html#THREAD_PRIORITY_AUDIO) constants in the Process (https://developer.android.com/reference/android/os/Process.html) class.

For more information on managing threads, see the reference documentation about the Thread (https://developer.android.com/reference/java/lang/Thread.html) and Process (https://developer.android.com/reference/android/os/Process.html) classes.

Helper Classes for Threading

The framework provides the same Java classes and primitives to facilitate threading, such as the Thread (https://developer.android.com/reference/java/lang/Thread.html), Runnable (https://developer.android.com/reference/java/lang/Runnable.html) , and Executors (https://developer.android.com/reference/java/util/concurrent/Executors.html) classes. In order to help reduce the cognitive load associated with of developing threaded applications for Android, the framework provides a set of helpers which can aide in development, such as AsyncTaskLoader (https://developer.android.com/reference/android/content/AsyncTaskLoader.html) and AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html). Each helper class has a specific set of performance nuances that make them unique for a specific subset of threading problems. Using the wrong class for the wrong situation can lead to performance issues.

The AsyncTask class

The AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) class is a simple, useful primitive for apps that need to quickly move work from the main thread onto worker threads. For example, an input event might trigger the need to update the UI with a loaded bitmap. An AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) object can offload the bitmap loading and decoding to an alternate thread; once that processing is complete, the AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) object can manage receiving the work back on the main thread to update the UI.

When using AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html), there are a few important performance aspects to keep in mind. First, by default, an app pushes all of the AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) objects it creates into a single thread. Therefore, they execute in serial fashion, and—as with the main thread—an especially long work packet can block the queue. For this reason, we suggest that you only use AsyncTask (https://developer.android.com

/reference/android/os/AsyncTask.html) to handle work items shorter than 5ms in duration.

AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) objects are also the most common offenders for implicit-reference issues. AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) objects present risks related to explicit references, as well, but these are sometimes easier to work around. For example, an AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) may require a reference to a UI object in order to update the UI object properly once AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) executes its callbacks on the main thread. In such a situation, you can use a WeakReference (https://developer.android.com/reference/java/lang/ref/WeakReference.html) to store a reference to the required UI object, and access the object once the AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) is operating on the main thread. To be clear, holding a WeakReference (https://developer.android.com/reference/java/lang/ref/WeakReference.html) to an object does not make the object thread-safe; the WeakReference (https://developer.android.com/reference/java/lang/ref/WeakReference.html) merely provides a method to handle issues with explicit references and garbage collection.

The HandlerThread class

While an AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html) is useful, it may not always be the right solution (https://www.youtube.com/watch?v=adPLIAnx9og&index=5&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE) to your threading problem. Instead, you may need a more traditional approach to executing a block of work on a longer running thread, and some ability to manage that workflow manually.

Consider a common challenge with getting preview frames from your Camera (https://developer.android.com/reference/android/hardware/Camera.html) object. When you register for Camera preview frames, you receive them in the onPreviewFrame() (https://developer.android.com/reference/android/hardware/Camera.PreviewCallback.html#onPreviewFrame(byte[], android.hardware.Camera)) callback, which is invoked on the event thread it was called from. If this callback were invoked on the UI thread, the task of dealing with the huge pixel arrays would be interfering with rendering and event processing work. The same problem applies to AsyncTask (https://developer.android.com/reference/android/os/AsyncTask.html), which also executes jobs serially and is susceptible to blocking.

This is a situation where a handler thread would be appropriate: A handler thread is effectively a long-running thread that grabs work from a queue, and operates on it. In this example, when your app delegates the Camera.open() (https://developer.android.com/reference/android/hardware/Camera.html#open()) command to a block of work on the handler thread, the associated onPreviewFrame() (https://developer.android.com/reference/android/hardware/Camera.PreviewCallback.html#onPreviewFrame(byte[], android.hardware.Camera))

callback lands on the handler thread, rather than the UI or `AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) threads. So, if you're going to be doing long-running work on the pixels, this may be a better solution for you.

When your app creates a thread using `HandlerThread` (https://developer.android.com/reference/android/os/HandlerThread.html), don't forget to set the thread's priority based on the type of work it's doing (https://www.youtube.com/watch?v=NwFXVsM15Co&index=9&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE). Remember, CPUs can only handle a small number of threads in parallel. Setting the priority helps the system know the right ways to schedule this work when all other threads are fighting for attention.

The ThreadPoolExecutor class

There are certain types of work that can be reduced to highly parallel, distributed tasks. One such task, for example, is calculating a filter for each 8x8 block of an 8 megapixel image. With the sheer volume of work packets this creates, `AsyncTask` and `HandlerThread` aren't appropriate classes (https://www.youtube.com/watch?v=uCmHoEY1iTM&index=6&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE). The single-threaded nature of `AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) would turn all the threadpooled work into a linear system. Using the `HandlerThread` (https://developer.android.com/reference/android/os/HandlerThread.html) class, on the other hand, would require the programmer to manually manage load balancing between a group of threads.

`ThreadPoolExecutor` (https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html) is a helper class to make this process easier. This class manages the creation of a group of threads, sets their priorities, and manages how work is distributed among those threads. As workload increases or decreases, the class spins up or destroys more threads to adjust to the workload.

This class also helps your app spawn an optimum number of threads. When it constructs a `ThreadPoolExecutor` (https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html) object, the app sets a minimum and maximum number of threads. As the workload given to the `ThreadPoolExecutor` (https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html) increases, the class will take the initialized minimum and maximum thread counts into account, and consider the amount of pending work there is to do. Based on these factors, `ThreadPoolExecutor` (https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html) decides on how many threads should be alive at any given time.

How many threads should you create?

Although from a software level, your code has the ability to create hundreds of threads, doing so can create performance issues. Your app shares limited CPU resources with background services, the renderer, audio

engine, networking, and more. CPUs really only have the ability to handle a small number of threads in parallel; everything above that runs into priority and scheduling issue (https://www.youtube.com/watch?v=NwFXVsM15Co&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE&index=9). As such, it's important to only create as many threads as your workload needs.

Practically speaking, there's a number of variables responsible for this, but picking a value (like 4, for starters), and testing it with Systrace (https://developer.android.com/studio/profile/systrace-commandline.html) is as solid a strategy as any other. You can use trial-and-error to discover the minimum number of threads you can use without running into problems.

Another consideration in deciding on how many threads to have is that threads aren't free: they take up memory. Each thread costs a minimum of 64k of memory. This adds up quickly across the many apps installed on a device, especially in situations where the call stacks grow significantly.

Many system processes and third-party libraries often spin up their own threadpools. If your app can reuse an existing threadpool, this reuse may help performance by reducing contention for memory and processing resources.