Processes and threads overview

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). If an application component starts and there already exists a process for that application (because another component from the application exists), then the component is started within that process and uses the same thread of execution. However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.

This document discusses how processes and threads work in an Android application.

Processes

By default, all components of the same application run in the same process and most applications should not change this. However, if you find that you need to control which process a certain component belongs to, you can do so in the manifest file.

The manifest entry for each type of component element—**`<activity>`** (https://developer.android.com/guide/topics/manifest/activity-element.html), **`<service>`** (https://developer.android.com/guide/topics/manifest/service-element.html), **`<receiver>`** (https://developer.android.com/guide/topics/manifest/receiver-element.html), and **`<provider>`** (https://developer.android.com/guide/topics/manifest/provider-element.html)—supports an **`android:process`** attribute that can specify a process in which that component should run. You can set this attribute so that each component runs in its own process or so that some components share a process while others do not. You can also set **`android:process`** so that components of different applications run in the same process—provided that the applications share the same Linux user ID and are signed with the same certificates.

The **`<application>`** (https://developer.android.com/guide/topics/manifest/application-element.html) element also supports an **`android:process`** attribute, to set a default value that applies to all components.

Android might decide to shut down a process at some point, when memory is low and required by other processes that are more immediately serving the user. Application

components running in the process that's killed are consequently destroyed. A process is started again for those components when there's again work for them to do.

When deciding which processes to kill, the Android system weighs their relative importance to the user. For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to a process hosting visible activities. The decision whether to terminate a process, therefore, depends on the state of the components running in that process.

The details of the process lifecycle and its relationship to application states are discussed in Processes and Application Lifecycle (https://developer.android.com/guide/topics/processes/process-lifecycle.html).

Threads

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events. It is also almost always the thread in which your application interacts with components from the Android UI toolkit (components from the **`android.widget`** (https://developer.android.com/reference/android/widget/package-summary.html) and **`android.view`** (https://developer.android.com/reference/android/view/package-summary.html) packages). As such, the main thread is also sometimes called the UI thread. However, under special circumstances, an app's main thread might not be its UI thread; for more information, see Thread annotations (https://developer.android.com/studio/write/annotations.html#thread-annotations).

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 from that thread. Consequently, methods that respond to system callbacks (such as **`onKeyDown()`** (https://developer.android.com/reference/android/view/View.html#onKeyDown(int, android.view.KeyEvent)) to report user actions or a lifecycle callback method) always run in the UI thread of the process.

For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue. The UI thread dequeues the request and notifies the widget that it should redraw itself.

When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly. Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI. When the thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to hang. Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "application not responding (http://developer.android.com/guide/practices/responsiveness.html)" (ANR) dialog. The user might then decide to quit your application and uninstall it if they are unhappy.

Additionally, the Android UI toolkit is *not* thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Thus, there are simply two rules to Android's single thread model:

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

Worker threads

Because of the single threaded 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).

However, note that you cannot update the UI from any thread other than the UI thread or the "main" thread.

To fix this problem, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:

- `Activity.runOnUiThread(Runnable)` (https://developer.android.com/reference/android/app/Activity.html#runOnUiThread(java.lang.Runnable))
- `View.post(Runnable)` (https://developer.android.com/reference/android/view/View.html#post(java.lang.Runnable))
- `View.postDelayed(Runnable, long)` (https://developer.android.com/reference/android/view/View.html#postDelayed(java.lang.Runnable, long))

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
```

```
            // a potentially  time consuming task
            final Bitmap bitmap =
                    processBitMap("image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

This implementation is thread-safe: the background operation is done from a separate thread while the `ImageView` (https://developer.android.com/reference/android/widget/ImageView.html) is always manipulated from the UI thread.

However, as the complexity of the operation grows, this kind of code can get complicated and difficult to maintain. To handle more complex interactions with a worker thread, you might consider using a `Handler` (https://developer.android.com/reference/android/os/Handler.html) in your worker thread, to process messages delivered from the UI thread. Perhaps the best solution, is to extend the `AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) class, which simplifies the execution of worker thread tasks that need to interact with the UI.

Using AsyncTask

`AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) allows you to perform asynchronous work on your user interface. It performs the blocking operations in a worker thread and then publishes the results on the UI thread, without requiring you to handle threads and/or handlers yourself.

To use it, you must subclass `AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) and implement the `doInBackground()` (https://developer.android.com/reference/android/os/AsyncTask.html#doInBackground(Params...)) callback method, which runs in a pool of background threads. To update your UI, you should implement `onPostExecute()` (https://developer.android.com/reference/android/os/AsyncTask.html#onPostExecute(Result)), which delivers the result from `doInBackground()` (https://developer.android.com/reference/android/os/AsyncTask.html#doInBackground(Params...)) and runs in the UI thread, so you can safely update your UI. You can then run the task by calling `execute()`

(https://developer.android.com/reference/android/os/AsyncTask.html#execute(Params...)) from the UI thread.

You should read the `AsyncTask` (https://developer.android.com/reference/android/os/AsyncTask.html) reference for a full understanding on how to use this class.

Thread-safe methods

In some situations, the methods you implement might be called from more than one thread, and therefore must be written to be thread-safe.

This is primarily true for methods that can be called remotely—such as methods in a bound service (https://developer.android.com/guide/components/bound-services.html). When a call on a method implemented in an `IBinder` (https://developer.android.com/reference/android/os/IBinder.html) originates in the same process in which the `IBinder` (https://developer.android.com/reference/android/os/IBinder.html) is running, the method is executed in the caller's thread. However, when the call originates in another process, the method is executed in a thread chosen from a pool of threads that the system maintains in the same process as the `IBinder` (https://developer.android.com/reference/android/os/IBinder.html) (it's not executed in the UI thread of the process). For example, whereas a service's `onBind()` (https://developer.android.com/reference/android/app/Service.html#onBind(android.content.Intent)) method would be called from the UI thread of the service's process, methods implemented in the object that `onBind()` (https://developer.android.com/reference/android/app/Service.html#onBind(android.content.Intent)) returns (for example, a subclass that implements RPC methods) would be called from threads in the pool. Because a service can have more than one client, more than one pool thread can engage the same `IBinder` (https://developer.android.com/reference/android/os/IBinder.html) method at the same time. `IBinder` (https://developer.android.com/reference/android/os/IBinder.html) methods must, therefore, be implemented to be thread-safe.

Similarly, a content provider can receive data requests that originate in other processes. Although the `ContentResolver` (https://developer.android.com/reference/android/content/ContentResolver.html) and `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html) classes hide the details of how the interprocess communication is managed, `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html) methods that respond to those requests—the methods `query()`

(https://developer.android.com/reference/android/content/ContentProvider.html#query(android.net.Uri, java.lang.String[], android.os.Bundle, android.os.CancellationSignal))
, `insert()`
(https://developer.android.com/reference/android/content/ContentProvider.html#insert(android.net.Uri, android.content.ContentValues))
, `delete()`
(https://developer.android.com/reference/android/content/ContentProvider.html#delete(android.net.Uri, java.lang.String, java.lang.String[]))
, `update()`
(https://developer.android.com/reference/android/content/ContentProvider.html#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[]))
, and `getType()`
(https://developer.android.com/reference/android/content/ContentProvider.html#getType(android.net.Uri))
—are called from a pool of threads in the content provider's process, not the UI thread for the process. Because these methods might be called from any number of threads at the same time, they too must be implemented to be thread-safe.

Interprocess communication

Android offers a mechanism for interprocess communication (IPC) using remote procedure calls (RPCs), in which a method is called by an activity or other application component, but executed remotely (in another process), with any result returned back to the caller. This entails decomposing a method call and its data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, then reassembling and reenacting the call there. Return values are then transmitted in the opposite direction. Android provides all the code to perform these IPC transactions, so you can focus on defining and implementing the RPC programming interface.

To perform IPC, your application must bind to a service, using `bindService()`
(https://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent, android.content.ServiceConnection, int))
. For more information, see the Services
(https://developer.android.com/guide/components/services.html) developer guide.

Last updated April 23, 2018.

Twitter
Follow @AndroidDev on Twitter

Google+
Follow Android Developers on Google+

YouTube
Check out Android Developers on YouTube