Processes and Threads

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.

IN THIS DOCUMENT

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>` (/guide/topics/manifest/activity-element.html), `<service>` (/guide/topics/manifest/service-element.html), `<receiver>` (/guide/topics/manifest/receiver-element.html), and `<provider>` (/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>` (/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 rules used to decide which processes to terminate is discussed below.

Process lifecycle

The Android system tries to maintain an application process for as long as possible, but eventually needs to remove old processes to reclaim memory for new or more important processes. To determine which processes to keep and which to kill, the system places each process into an "importance hierarchy" based on the components running in the process and the state of those components. Processes with the lowest importance are eliminated first, then those with the next lowest importance, and so on, as necessary to recover system resources.

There are five levels in the importance hierarchy. The following list presents the different types of processes in order of importance (the first process is *most important* and is *killed last*):

1. **Foreground process**

   A process that is required for what the user is currently doing. A process is considered to be in the foreground if any of the following conditions are true:

- It hosts an `Activity` that the user is interacting with (the `Activity`'s `onResume()` method has been called).
- It hosts a `Service` that's bound to the activity that the user is interacting with.
- It hosts a `Service` that's running "in the foreground"—the service has called `startForeground()`.
- It hosts a `Service` that's executing one of its lifecycle callbacks (`onCreate()`, `onStart()`, or `onDestroy()`).
- It hosts a `BroadcastReceiver` that's executing its `onReceive()` method.

Generally, only a few foreground processes exist at any given time. They are killed only as a last resort—if memory is so low that they cannot all continue to run. Generally, at that point, the device has reached a memory paging state, so killing some foreground processes is required to keep the user interface responsive.

2. **Visible process**

   A process that doesn't have any foreground components, but still can affect what the user sees on screen. A process is considered to be visible if either of the following conditions are true:

- It hosts an `Activity` that is not in the foreground, but is still visible to the user (its `onPause()` method has been called). This might occur, for example, if the foreground activity started a dialog, which allows the previous activity to be seen behind it.
- It hosts a `Service` that's bound to a visible (or foreground) activity.

   A visible process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.

3. **Service process**

   A process that is running a service that has been started with the `startService()` (/reference/android/content/Context.html#startService(android.content.Intent)) method and does not fall into either of the two higher categories. Although service processes are not directly tied to anything the user sees, they are generally doing things that the user cares about (such as playing music in the background or downloading data on the network), so the system keeps them running unless there's not enough memory to retain them along with all foreground and visible processes.

4. **Background process**

   A process holding an activity that's not currently visible to the user (the activity's `onStop()` (/reference/android/app/Activity.html#onStop()) method has been called). These processes have no direct impact on the user experience, and the system can kill them at any time to reclaim memory for a foreground, visible, or service process. Usually there are many background processes running, so they are kept in an LRU (least recently used) list to ensure that the process with the activity that was most recently seen by the user is the last to be killed. If an activity implements its lifecycle methods correctly, and saves its current state, killing its process will not have a visible effect on the user experience, because when the user navigates back to the activity, the activity restores all of its visible state. See the Activities (/guide/components/activities.html#SavingActivityState) document for information about saving and restoring state.

5. **Empty process**

   A process that doesn't hold any active application components. The only reason to keep this kind of process alive is for caching purposes, to improve startup time the next time a component needs to run in it. The system often kills these processes in order to balance overall system resources between process caches and the underlying kernel caches.

Android ranks a process at the highest level it can, based upon the importance of the components currently active in the process. For example, if a process hosts a service and a visible activity, the process is ranked as a visible process, not a service process.

In addition, a process's ranking might be increased because other processes are dependent on it—a process that is serving another process can never be ranked lower than the process it is serving. For example, if a content provider in process A is serving a client in process B, or if a service in process A is bound to a component in process B, process A is always considered at least as important as process B.

Because a process running a service is ranked higher than a process with background activities, an activity that initiates a long-running operation might do well to start a service (/guide/components/services.html) for that operation, rather than simply create a worker thread—particularly if the operation will likely outlast the activity. For example, an activity that's uploading a picture to a web site should start a service to perform the upload so that the upload can continue in the background even if the user leaves the activity. Using a service guarantees that the operation will have at least "service process" priority, regardless of what happens to the activity. This is the same reason that broadcast receivers should employ services rather than simply put time-consuming operations in a thread.

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 the thread in which your application interacts with components from the Android UI toolkit (components from the `android.widget` (/reference/android/widget/package-summary.html) and `android.view` (/reference/android/view/package-summary.html) packages). As such, the main thread is also sometimes 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 from that thread. Consequently, methods that respond to system callbacks (such as `onKeyDown()` (/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 Andoid 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 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` (/reference/android/widget/ImageView.html):

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

At first, this seems to work fine, because it creates a new thread to handle the network operation. However, it violates the second rule of the single-threaded model: *do not access the Android UI toolkit from outside the UI thread*—this sample modifies the `ImageView` (/reference/android/widget/ImageView.html) from the worker thread instead of the UI thread. This can result in undefined and unexpected behavior, which can be difficult and time-consuming to track down.

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)`
- `View.post(Runnable)`
- `View.postDelayed(Runnable, long)`

For example, you can fix the above code by using the `View.post(Runnable)` (/reference/android/view/View.html#post(java.lang.Runnable)) method:

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

Now this implementation is thread-safe: the network operation is done from a separate thread while the `ImageView` (/reference/android/widget/ImageView.html) is 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` (/reference/android/os/Handler.html) in your worker thread, to process messages delivered from the UI thread. Perhaps the best solution, though, is to extend the `AsyncTask` (/reference/android/os/AsyncTask.html) class, which simplifies the execution of worker thread tasks that need to interact with the UI.

Using AsyncTask

`AsyncTask` (/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` (/reference/android/os/AsyncTask.html) and implement the `doInBackground()` (/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()` (/reference/android/os/AsyncTask.html#onPostExecute(Result)), which delivers the result from `doInBackground()` (/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()` (/reference/android/os/AsyncTask.html#execute(Params...)) from the UI thread.

For example, you can implement the previous example using `AsyncTask` (/reference/android/os/AsyncTask.html) this way:

```
public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/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 simpler, because it separates the work into the part that should be done on a worker thread and the part that should be done on the UI thread.

You should read the `AsyncTask` (/reference/android/os/AsyncTask.html) reference for a full understanding on how to use this class, but here is a quick overview of how it works:

- You can specify the type 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 anytime in `doInBackground()` to execute `onProgressUpdate()` on the UI thread
- You can cancel the task at any time, from any thread

> **Caution:** Another problem you might encounter when using a worker thread is unexpected restarts in your activity due to a runtime configuration change (/guide/topics/resources/runtime-changes.html) (such as when the user changes the screen orientation), which may destroy your worker thread. To see how you can persist your task during one of these restarts and how to properly cancel the task when the activity is destroyed, see the source code for the Shelves (http://code.google.com/p/shelves/) sample application.

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 (/guide/components/bound-services.html). When a call on a method implemented in an `IBinder (/reference/android/os/IBinder.html)` originates in the same process in which the `IBinder (/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 (/reference/android/os/IBinder.html)` (it's not executed in the UI thread of the process). For example, whereas a service's `onBind() (/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() (/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 (/reference/android/os/IBinder.html)` method at the same time. `IBinder (/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 (/reference/android/content/ContentResolver.html)` and `ContentProvider (/reference/android/content/ContentProvider.html)` classes hide the details of how the interprocess communication is managed, `ContentProvider (/reference/android/content/ContentProvider.html)` methods that respond to those requests—the methods `query() (/reference/android/content/ContentProvider.html#query(android.net.Uri, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String))`, `insert() (/reference/android/content/ContentProvider.html#insert(android.net.Uri,` android.content.ContentValues)), delete() (/reference/android/content/ContentProvider.html#delete(android.net.Uri, java.lang.String, java.lang.String[])), update() (/reference/android/content/ContentProvider.html#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[])), and getType() (/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() (/reference/android/content/Context.html#bindService(android.content.Intent, android.content.ServiceConnection, int)). For more information, see the Services (/guide/components/services.html) developer guide.