Save the date! Android Dev Summit (/dev-summit) **is coming to Mountain View, CA on November 7-8, 2018.**

Processes and Application Lifecycle

In most cases, every Android application runs in its own Linux process. This process is created for the application when some of its code needs to be run, and will remain running until it is no longer needed *and* the system needs to reclaim its memory for use by other applications.

An unusual and fundamental feature of Android is that an application process's lifetime is *not* directly controlled by the application itself. Instead, it is determined by the system through a combination of the parts of the application that the system knows are running, how important these things are to the user, and how much overall memory is available in the system.

It is important that application developers understand how different application components (in particular `Activity` (https://developer.android.com/reference/android/app/Activity.html), `Service` (https://developer.android.com/reference/android/app/Service.html), and `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html)) impact the lifetime of the application's process. **Not using these components correctly can result in the system killing the application's process while it is doing important work.**

A common example of a process life-cycle bug is a `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) that starts a thread when it receives an Intent in its `BroadcastReceiver.onReceive()` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context, android.content.Intent)) method, and then returns from the function. Once it returns, the system considers the BroadcastReceiver to be no longer active, and thus, its hosting process no longer needed (unless other application components are active in it). So, the system may kill the process at any time to reclaim memory, and in doing so, it terminates the spawned thread running in the process. The solution to this problem is typically to schedule a `JobService` (https://developer.android.com/reference/android/app/job/JobService.html) from the BroadcastReceiver, so the system knows that there is still active work being done in the process.

To determine which processes should be killed when low on memory, Android places each process into an "importance hierarchy" based on the components running in them and the state of those components. These process types are (in order of importance):

1. A **foreground process** is one that is required for what the user is currently doing. Various application components can cause its containing process to be considered foreground in different ways. A process is considered to be in the foreground if any of the following conditions hold:

    - It is running an `Activity` (https://developer.android.com/reference/android/app/Activity.html) at the top of the screen that the user is interacting with (its `onResume()` (https://developer.android.com/reference/android/app/Activity.html#onResume()) method has been called).
    - It has a `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) that is currently running (its `BroadcastReceiver.onReceive()` (https://developer.android.com/reference/android/content/BroadcastReceiver.html#onReceive(android.content.Context, android.content.Intent)) method is executing).
    - It has a `Service` (https://developer.android.com/reference/android/app/Service.html) that is currently executing code in one of its callbacks (`Service.onCreate()` (https://developer.android.com/reference/android/app/Service.html#onCreate()), `Service.onStart()` (https://developer.android.com/reference/android/app/Service.html#onStart(android.content.Intent, int)), or `Service.onDestroy()` (https://developer.android.com/reference/android/app/Service.html#onDestroy())).

    There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive.

2. A **visible process** is doing work that the user is currently aware of, so killing it would have a noticeable negative impact on the user experience. A process is considered visible in the following conditions:

    - It is running an `Activity` (https://developer.android.com/reference/android/app/Activity.html) that is visible to the user on-screen but not in the foreground (its `onPause()` (https://developer.android.com/reference/android/app/Activity.html#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` (https://developer.android.com/reference/android/app/Service.html) that is running as a foreground service, through `Service.startForeground()` (https://developer.android.com/reference/android/app/Service.html#startForeground(int, android.app.Notification))
  (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 particular feature that the user is aware, such as a live wallpaper, input method service, etc.

The number of these processes running in the system is less bounded than foreground processes, but still relatively controlled. These processes are considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.

3. A **service process** is one holding a `Service` (https://developer.android.com/reference/android/app/Service.html) that has been started with the `startService()` (https://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent))
   method. Though these processes are not directly visible to the user, they are generally doing things that the user cares about (such as background network data upload or download), so the system will always keep such processes running unless there is not enough memory to retain all foreground and visible processes.

   Services that have been running for a long time (such as 30 minutes or more) may be demoted in importance to allow their process to drop to the cached LRU list described next. This helps avoid situations where very long running services with memory leaks or other problems consume so much RAM that they prevent the system from making effective use of cached processes.

4. A **cached process** is one that is not currently needed, so the system is free to kill it as desired when memory is needed elsewhere. In a normally behaving system, these are the only processes involved in memory management: a well running system will have multiple cached processes always available (for more efficient switching between applications) and regularly kill the oldest ones as needed. Only in very critical (and undesireable) situations will the system get to a point where all cached processes are killed and it must start killing service processes.

   These processes often hold one or more `Activity` (https://developer.android.com/reference/android/app/Activity.html) instances that are not currently visible to the user (the `onStop()`

(https://developer.android.com/reference/android/app/Activity.html#onStop()) method has been called and returned). Provided they implement their Activity life-cycle correctly (see `Activity` (https://developer.android.com/reference/android/app/Activity.html) for more details), when the system kills such processes it will not impact the user's experience when returning to that app: it can restore the previously saved state when the associated activity is recreated in a new process.

These processes are kept in a pseudo-LRU list, where the last process on the list is the first killed to reclaim memory. The exact policy of ordering on this list is an implementation detail of the platform, but generally it will try to keep more useful processes (one hosting the user's home application, the last activity they saw, etc) before other types of processes. Other policies for killing processes may also be applied: hard limits on the number of processes allowed, limits on the amount of time a process can stay continually cached, etc.

When deciding how to classify a process, the system will base its decision on the most important level found among all the components currently active in the process. See the `Activity` (https://developer.android.com/reference/android/app/Activity.html), `Service` (https://developer.android.com/reference/android/app/Service.html), and `BroadcastReceiver` (https://developer.android.com/reference/android/content/BroadcastReceiver.html) documentation for more detail on how each of these components contribute to the overall life-cycle of a process. The documentation for each of these classes describes in more detail how they impact the overall life-cycle of their application.

A process's priority may also be increased based on other dependencies a process has to it. For example, if process A has bound to a `Service` (https://developer.android.com/reference/android/app/Service.html) with the `Context.BIND_AUTO_CREATE` (https://developer.android.com/reference/android/content/Context.html#BIND_AUTO_CREATE) flag or is using a `ContentProvider` (https://developer.android.com/reference/android/content/ContentProvider.html) in process B, then process B's classification will always be at least as important as process A's.

Content and code samples on this page are subject to the licenses described in the Content License (/license). *Java is a registered trademark of Oracle and/or its affiliates.*

Last updated April 17, 2018.

Twitter
Follow @AndroidDev on Twitter

Google+
Follow Android Developers on Google+

YouTube
Check out Android Developers on YouTube