
Google™
10

Writing Zippy Android Apps

Brad Fitzpatrick
May 20th, 2010

Live Wave: http://bit.ly/aU9bpD

Outline

- Me & why I care
- What is an ANR? Why do you see them?
- Quantifying responsiveness: “jank”
- Android SDK features to use to avoid jankiness & ANRs
- Numbers to know
- War stories from optimizing Froyo (2.2)
- New performance instrumentation available in Froyo
- Q&A and/or tell me what you want

- *Note:* not a talk on the Dalvik JIT (which is cool when CPU-bound, but often not the problem)

About Me

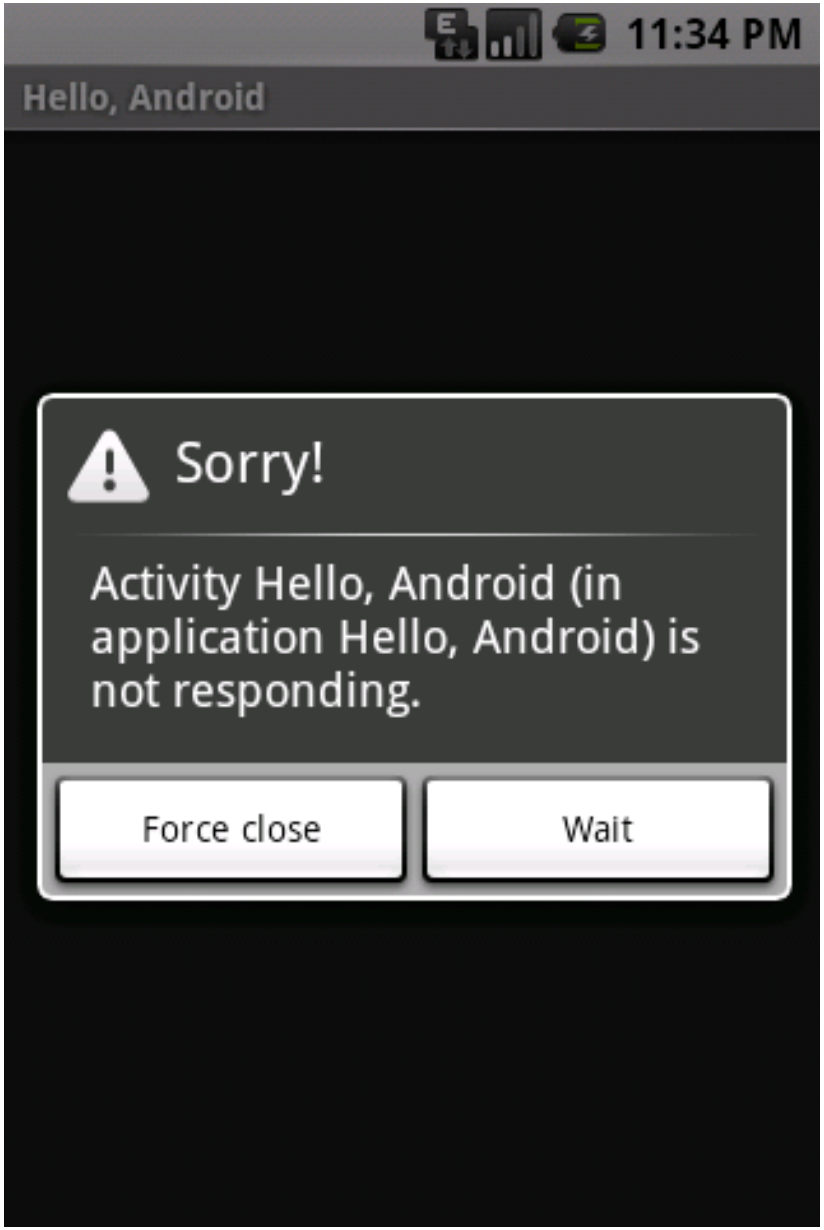
- Brad Fitzpatrick
- Pre-Google:
  - Danga.com / LiveJournal / Six Apart
    - memcached, OpenID, MogileFS, Gearman, perlbal, djabberd, ...
    - … mix of Social + infrastructure
    - <3 Open Source
- Google:
  - Social Graph API, ~Google Profiles
  - PubSubHubbub, WebFinger
- Now:
  - Android performance
  - working on Open Source again!

Jank

Jank

- Chrome team's term for stalling the event loop
  - Chrome's fanatically anti-jank
  - “Janky”: not being immediately responsive to input
- “Eliminating jank”
  - Reacting to events quickly,
  - Don't hog the event loop (“main” / UI) thread!
  - Getting back into the select() / epoll_wait() call ASAP, so...
  - … you can react to future events quickly (touches, drags)
- Else, ...

ANR!


11:34 PM
Hello, Android
Sorry!
Activity Hello, Android (in application Hello, Android) is not responding.
Force close
Wait

ANRs (“Application Not Responding”)

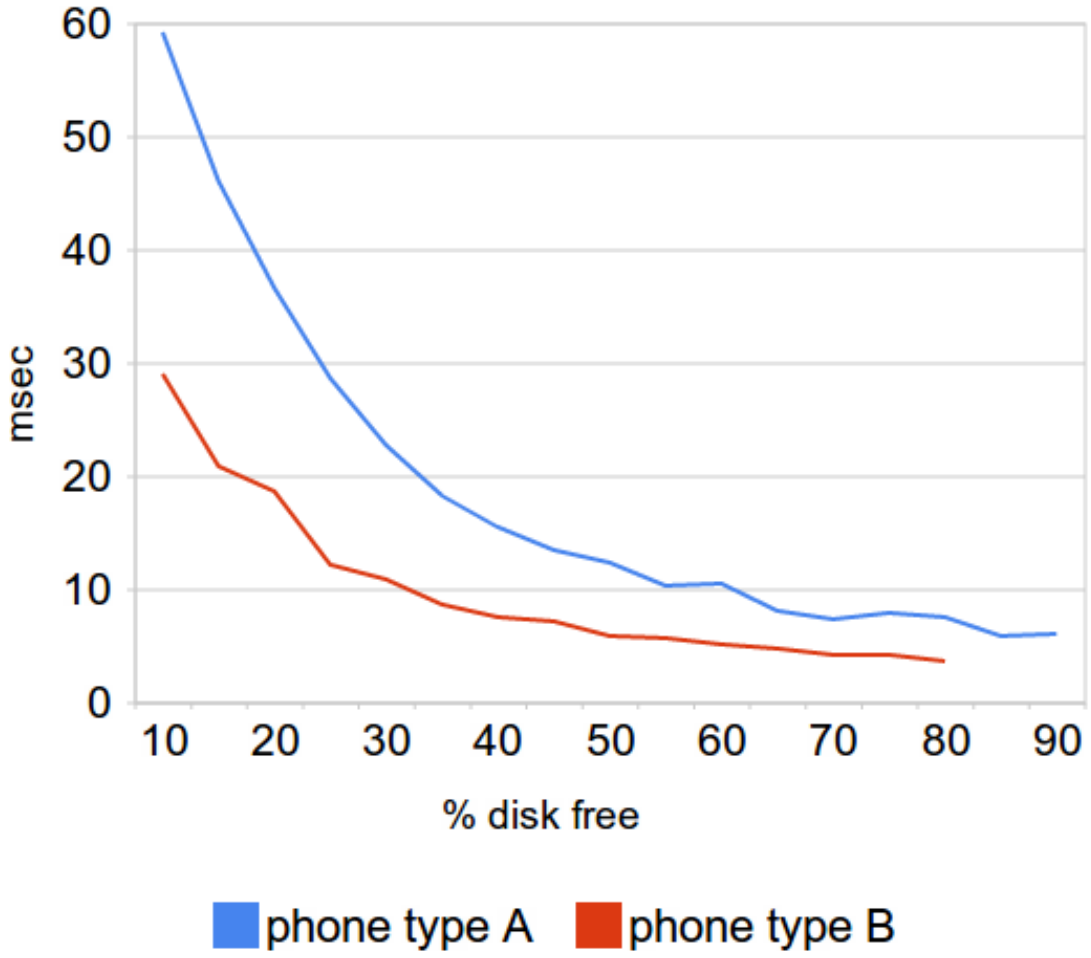
- ANRs happen when,
  - main thread (“event thread” / “UI thread”) doesn't respond to input event in 5 seconds,
  - a BroadcastReceiver doesn't finish in 10 seconds
  - typically, when
    - doing network operations on main thread,
    - doing slow 'disk' operations (un-optimized SQL) on the main thread
- Less than 5 or 10 seconds, though...
  - Users: “This app feels janky.” (or “sluggish”, “slow”, …)

Numbers

Numbers (Nexus One)

- ~0.04 ms – writing a byte on pipe process A->B, B->A
  - or reading simple /proc files (from dalvik)
- ~0.12 ms – void/void Binder RPC call A->B, B->A
- ~5-25 ms – uncached flash reading a byte
- **~5-200+(!)** ms – uncached flash *writing* tiny amount (next...)
- 16 ms – one frame of 60 fps video
- 41 ms – one frame of 24 fps video
- 100-200 ms – human perception of slow action
- 108/350/500/800 ms – ping over 3G. varies!
- ~1-6+ seconds – TCP setup + HTTP fetch of 6k over 3G

Writing to flash (yaffs2)



Source: empirical samples over Google employee phones (Mar 2010)

- Create file, 512 byte write, delete
  - ala sqlite .journal in transaction
- Flash is … *different* than disks you're likely used to
  - read, write, *erase*, wear-leveling, GC, …
- nutshell: write performance varies a lot

sqlite performance

- previous writes were what sqlite does on ~each transaction,
  - even no-op transactions (since reported & fixed upstream)
- use indexes
  - EXPLAIN vs. EXPLAIN QUERY PLAN
- log files: often much cheaper to append to a file than use a database
  - look at “adb shell cat /proc/yaffs” and look at writes/erases
  - sqlite can be pretty write-happy
- sqlite-wrapper.pl tool
  - http://code.google.com/p/zippy-android/
- <demo>

Lessons so far

- Writing to disk is “slow”
- Using the network is “slow”
- **Be paranoid. Always assume the worst.**
  - given enough users, usage, time...
  - users will find your ANRs
  - bad Market reviews, uninstalls,
  - me filing bugs on your app :-)

Tools

android.os.AsyncTask

"AsyncTask enables proper and easy use of the UI thread. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers."

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
     protected Long doInBackground(URL... urls) {  // on some background thread
         int count = urls.length; long totalSize = 0;
         for (int i = 0; i < count; i++) {
             totalSize += Downloader.downloadFile(urls[i]);
             publishProgress((int) ((i / (float) count) * 100));
         }
         return totalSize;
     }
     protected void onProgressUpdate(Integer... progress) {  // on UI thread!
         setProgressPercent(progress[0]);
     }
     protected void onPostExecute(Long result) {  // on UI thread!
         showDialog("Downloaded " + result + " bytes");
     }
 }

 new DownloadFilesTask().execute(url1, url2, url3);  // call from UI thread!
```

Fire-and-forget style

```
private boolean handleWebSearchRequest(final ContentResolver cr) {
     ...
     new AsyncTask<Void, Void, Void>() {
       protected Void doInBackground(Void... unused) {
         Browser.updateVisitedHistory(cr, newUrl, false);
         Browser.addSearchUrl(cr, newUrl);
         return null;
       }
     }.execute()
     ...
     return true;
 }
```

Source: Froyo's src/com/android/browser/BrowserActivity.java, roughly

AsyncTask Caveats

- Must be called from a main thread
  - rather, a thread with a Handler/Looper around
  - don't use AsyncTask in a library where caller could call it from their own AsyncTask. or, check first.
- If called from an activity, the activity process may exit before your AsyncTask completes
  - user leaves activity,
  - system is low on RAM,
  - system serializes activity's state for later,
  - system kills your process (“just a replaceable activity!”)
  - if work is critical, use...

android.app.IntentService

- Eclair (2.0, 2.1) docs:
  - *“An abstract Service that serializes the handling of the Intents passed upon service start and handles them on a handler thread. To use this class extend it and implement onHandleIntent(Intent). The Service will automatically be stopped when the last enqueued Intent is handled.”*
  - little confusing, thus...
  - nobody really used it
- Froyo (2.2) docs, clarified....

android.app.IntentService

“**IntentService** is a base class for **Services** that handle asynchronous requests (expressed as **Intents**) on demand. Clients send requests through **startService(Intent**) calls; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work.

This 'work queue processor' pattern is commonly used to offload tasks from an application's main thread. The **IntentService** class exists to simplify this pattern and take care of the mechanics. To use it, extend **IntentService** and implement **onHandleIntent**(**Intent**). **IntentService** will receive the Intents, launch a worker thread, and stop the service as appropriate.

All requests are handled on a single worker thread -- they may take as long as necessary (and will not block the application's main loop), but only one request will be processed at a time.”

IntentService benefits

- your activity's process, while processing that Intent, now has a Service running
- the Android process killer will try really hard not to kill you now
- but once you're done handling your piece of work, you're now properly disposable again
- very easy way to use a Service

Calendar's use of IntentService

```
public class DismissAllAlarmsService extends IntentService {
  @Override public void onHandleIntent(Intent unusedIntent) {
    ContentResolver resolver = getContentResolver();
    ...
    resolver.update(uri, values, selection, null);
  }
}
```

in AlertReceiver extends BroadcastReceiver, onReceive(): (main thread)

```
  Intent intent = new Intent(context, DismissAllAlarmsService.class);
  context.startService(intent);
```

Source: Froyo's src/com/android/calendar/DismissAlAlarmsService.java, roughly

Other tips

- disable UI elements immediately, before kicking off your AsyncTask to finish the task
- do some animation
  - e.g. spinner in title bar
- ProgressDialog
  - for long-ish operations
  - use sparingly. can be jarring / interrupting.
- Combination of above,
  - not sure how long it will take?
  - start with disabling UI elements, animating something,
  - start timer,
  - show a ProgressDialog if it's taking over 200 ms.
  - in AsyncTask onPostExecute, cancel alarm timer

Non-jank related tools...

Traceview

- when CPU-bound
  - trace file format supports walltime clocksource, but GUI traceview tool ignores it, assumes per-thread CPU time
- toggle in code:
  - dalvik.system.VMDebug#{start,stop}MethodTracing()
- toggle at runtime:
  - adb shell am profile <PROCESS> start <FILE>
  - adb shell am profile <PROCESS> stop

print “here”

- non-@public class android.os.PerformanceCollector
  - I've seen lots of one-off variants
- log occasionally:
  - trace point name {start,stop}
  - android.os.SystemClock#uptimeMillis()
  - dalvik.system.VMDebug#threadCpuTimeNanos()
- watch your own device,
- upload, aggregate, and analyze from users?
  - often overkill

Froyo (2.2)

Performance Instrumentation in Froyo (2.2)

- We've instrumented,
  - Database queries,
  - RPC (Binder) calls,
  - ContentResolver,
  - Mutex lock contention in Dalvik,
- How long? Main thread or not? Which thread/process? Which build?
- Log a small percentage to in-memory ring-buffer
- Service (on Google employee dogfooder phones) uploading a few MB/day/phone for analysis, report generation
- Found & fixed many things, often surpising

Performance Instrumentation in Froyo (2.2)

- Not a generic debugging feature yet, but somewhat usable...
- Log vs EventLog
  - Log: text ring-buffer (adb logcat), apps use this.
  - EventLog: binary structured ring-buffer (adb logcat -b events), intended for low-level platform development work only
- EventLog records:
  - db_sample
  - binder_sample
  - content_query_sample
  - content_update_sample
  - dvm_lock_sample

Coming later

After Froyo

maybe? speculative...

- more instrumentation
  - more granular db stats, mutex lock->unlock duration stats...
- easier to use during app development
- API additions to let you force “strict mode” in your apps,
  - prevent read and/or write I/O on your main thread
  - prevent slow Binder calls to other processes doing I/O
  - prevent network I/O on the main thread
- let users opt-in to performance collection (system-wide or app-by-app? separate app?)
  - surface empirical performance data to developers through Android Market?
- sqlite WAL

Summary

Summary

- Get off the main thread!
- Disks & networks aren't instantaneous
- Know what sqlite is doing
- Instrumentation is fun :-)
- More speed goodness coming post-Froyo...

- Code, tools from this talk:

  http://code.google.com/p/zippy-android/

- Q&A? *Live Wave: http://bit.ly/aU9bpD*


Google™
10