WHY ANDROID?

```shell
the definition of open: "mkdir android; cd android; repo init -u git://android.git.kernel.org/platform/manifest.git; repo sync; make"
```

- What does it mean to researchers?
- What does it mean to users?
OUTLINE

- Android platform architecture
- OS kernel, libraries and devices
- Android programming model
- Delvik Virtual Machine
- Energy efficiency
- How to write efficient code
FIRST THING FIRST

What is the difference between a mobile OS and a desktop/server OS?
ARCHITECTURE

Linux Kernel

- Display Driver
- Camera Driver
- Flash Memory Driver
- Binder (IPC) Driver
- Keypad Driver
- WiFi Driver
- Audio Drivers
- Power Management
ANDROID
ANDROID
ANDROID

Applications
- Home
- Contacts
- Phone
- Browser
- ...

Application Framework
- Activity Manager
- Window Manager
- Content Providers
- View System
- Telephony Manager
- Resource Manager
- Location Manager
- Notification Manager

Libraries
- Surface Manager
- OpenGL | ES
- FreeType
- GL
- SSL
- libc

Android Runtime
- Core Libraries
- Dalvik Virtual Machine

Linux Kernel
- Display Driver
- Camera Driver
- Flash Memory Driver
- Binder (IPC) Driver
- Keypad Driver
- WiFi Driver
- Audio Drivers
- Power Management
Android uses Linux 2.6 kernel as the hardware abstraction

What are the essences an OS should provide?

- Memory management, process management, IPC
- No virtual memory; specially implemented IPC

Drivers and architecture support

How to port Android to a new device?

Using Linux vs. Writing a new OS from scratch

Do all Linux kernel implementations work well on mobile devices?
GNU libs (glibc) is too big and complicated for mobile phones, so Android implements its own special version of libc - *Bionic libc*:

- Smaller size - 200K (glibc is more than 400K)
- Strip out some complicated C++ features, the most significant one - no C++ exception!
- Very special and small pthread implementation, heavily based on kernel futexes
- Bionic libc does *not* fully support POSIX and is *not* compatible with glibc
- which means ...?
What’s the difference between mobile apps cycle and desktop apps cycle?

Two key principles

- Android usually do not kill an app, i.e. apps keep running even after you switch to other apps.
- Android kills apps when the memory usage goes too high, but it saves app state for quick restart later on.

Do they make sense to mobile apps?
APPLICATION LIFE CYCLE

Activity starts
- onCreate()
- onStart()
- onRestart()
- onResume()

User navigates Back to the activity
- Activity is running

Process is killed
- Another activity comes in front of the activity
- Other applications Need memory
- The activity is no longer visible

The activity Comes to the foreground
- onPause()
- onStop()
- onDestroy()

Activity is shut down
At the “Home” screen
Start the “Mail” app and read the list
Click on one of the messages and see its content.
Click a link in the message
Now we have enough space to start the “Map” app
Start the “Map” app
EXAMPLE

Go back to the browser
The “Mail” app is resumed and shows the previous message.
EXAMPLE

Go back to the mail list
EXAMPLE

Go back to the “Home” screen
DEBATE

Swapping model
VS.
Android’s life-cycle model
# DISK I/O

<table>
<thead>
<tr>
<th>Feature</th>
<th>Flash</th>
<th>Hard Disk Drive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random access</td>
<td>~0.1ms</td>
<td>5-10ms</td>
</tr>
<tr>
<td>File fragment impact</td>
<td>No</td>
<td>Greatly impacted</td>
</tr>
<tr>
<td>Total power</td>
<td>1/2 to 1/3 of HDD</td>
<td>up to 15+ watts</td>
</tr>
<tr>
<td>Reliability</td>
<td>Reliable</td>
<td>Less reliable due to mechanical parts</td>
</tr>
<tr>
<td>Write longevity</td>
<td>Limited number of writes</td>
<td>Less of a problem</td>
</tr>
<tr>
<td>Capacity</td>
<td>&lt;= 512GB</td>
<td>2-3TB</td>
</tr>
<tr>
<td>Price</td>
<td>$1.5-2 / GB</td>
<td>$0.1-0.2 / GB</td>
</tr>
</tbody>
</table>
LIMITED WRITES?

Flash drives have the well-known problem of limited number of writes in the life time - 10,000~100,000 times. Solution?

What can applications do?

How about operating system?

Controllers?

Hardware?
MEMORY MANAGEMENT

- Linux kernel does most of the job
- Page-based memory management
- Virtual address to physical address mapping
- **NO** virtual memory

  - Why do we still need “virtual to physical” address mapping?
  - Why does Android not support virtual memory?
Why does Android let developers use Java?
DALVIK VM

- A special Java virtual machine (VM) designed to run with limited system resource
- Memory efficiency
- Register machine vs. Stack machine (modern JVM)
  - fewer instructions, faster execution
  - why does the number of instructions matter?
- Running multiple VMs more efficiently
Java class files are converted into “.dex” files that Dalvik executes.

Java byte-code is converted into Dalvik byte-code during this process.
MEMORY EFFICIENCY

- Shared constant string pool
- Share clean (even some dirty) memory between processes as much as possible
- “.dex” files are mapped as read-only by mmap()
- Memory efficient JIT implementation
- JIT itself is about 100K
- Code cache and supporting data structure takes another 100K for each application
public interface Zapper {
    public String zap(String s, Object o);
}

public class Blort implements Zapper {
    public String zap(String s, Object o) {
        ....
    }
}

public class ZapUser {
    public void useZap(Zapper z) {
        z.zap(...);
    }
}
SHARED STRING POOL
PROGRAMMING MODEL

- Each application is running in its own process
- An application can have one or more components:
  - activities, services, broadcast receivers and content providers
- A task (an “application” from user’s point of view) consists of several activities from one or multiple applications
- An application keeps running until the system kills it because of memory shortage
POWER SAVING

Source: Values measured using an industrial power monitor at 5kHz sampling rate, and taking average power with lowest standard deviation.

- Baseline usage
- Specific item

mA

Airplane, 3G idle, EDGE idle, WiFi idle, LCD normal, CPU 50%, Game sensors, GPS radio, 3G full, EDGE full, WiFi full

Picture is from Google I/O 09 talk - Coding for Life -- Battery Life, That Is
Use GZIP for text data whenever possible

Compressing is implemented by native code

```java
import java.util.zip.GZIPInputStream;

HttpGet request =
    new HttpGet("http://example.com/gzipcontent");
HttpResponse resp =
    new DefaultHttpClient().execute(request);
HttpEntity entity = response.getEntity();
InputStream compressed = entity.getContent();
InputStream raw_data = new GZIPInputStream(compressed);
```
Wifi and 3G are much more energy efficient, so wait for Wifi or 3G when transferring big chunk of data

```java
// Only update if WiFi or 3G is connected and not roaming
int netType = info.getType();
int netSubtype = info.getSubtype();

if (netType == ConnectivityManager.TYPE_WIFI) {
    return info.isConnected();
} else if (netType == ConnectivityManager.TYPE_MOBILE
        && netSubtype == TelephonyManager.NETWORK_TYPE_UMTS
        && !mTelephony.isNetworkRoaming()) {
    return info.isConnected();
} else {
    return false;
}
```
UPDATE BIN

Use `setInexactRepeating()` so the system can bin your update together with others
WORK OFFLOADING

- Naive offloading
  - Speech-to-text, OCR
- More sophisticated offloading - fine-grained offloading
  - MAUI: Making Smartphones Last Longer with Code Offload (MobiSys ’10)
  - Running two versions of the app on the mobile device and a powerful server
  - Decide when/what to offload on the fly
EFFICIENT CODE

- for (int i = initializer; i >= 0; i--)
- int limit = calculate limit;
  for (int i = 0; i < limit; i++)
- Type[] array = get array;
  for (Type obj : array)
- for (int i = 0; i < array.length; i++)
- for (int i = 0; i < this.var; i++)
- Iterable<Type> list = get list;
  for (Type obj : list)
EFFICIENT CODE

- Try to rest for the most of the time
- be nice to other processes
- Avoid allocation
  - short-lived objects need to be garbaged collected
  - long-lived objects take precious memory
- Make a method `static` if it does not access member variables
- Avoid internal getter/setters
- Use floating point numbers only when you have to
- Prefer `int` over `enum`
- Use `static final` for constants