Mobile Programming

Ian Batten
igb@batten.eu.org

Today's Content

- How did the exercise go?
  - A variety of things mean I haven't yet looked at the submissions in detail, but what I have seen is very good.
- Thread and Process Models (general)
- Thread and Process Models (Android)
- Services
- Broadcast

Process Models

- Android is Linux, and although the userland is very different, the kernel and the basic structure of processes is the same.

- So it is worth our while walking through the Unix process model to see how Android applications fit in.

Why do we care?

- We want to be able to (for example)
  - run computation while still responding to UI events
  - read from multiple sensors in arbitrary order
- In more complex cases we want to:
  - run services which distribute updates to multiple applications
  - share computations amongst multiple clients
- In general, we want to block on some “wait channel” while continuing to execute other code

Unix processes

- Process 1 (historically "`init`") is created ex nihilo by the kernel. The kernel knows what a process looks like, so can make one from scratch.

- All other processes are the result of calling `fork()` or one of its equivalents to copy the existing process.

- Processes have their own address space, their own file descriptors, their own signal handlers.

- Processes are also "expensive": `fork()` is slow and significant resources are consumed even by quiescent processes.

fork()

- `fork()` can be called at any time, by any process.
- Assuming there is no error, `fork()` returns **twice**:
  - It returns a “process identifier”, a small integer of type pid_t, into the calling (“parent”) process;
  - It returns 0 into a new (“child”) process which has the process id returned into the parent.
  - Modulo some minor details, the processes are otherwise the same.

`fork()` and `exec()`

- The usual thing that happens is that `fork()` is called, then some minor housekeeping is done to arrange the correct file descriptors on stdin (0), stdout (1) and stderr (2), and then a variant of `exec()` is called.

- `exec()` replaces the running process with a new binary, but leaves open file descriptors (and other process information) unchanged.

In action

```
/* headers deleted */

int
main (int argc, char **argv) {
  int outfd = open ("/tmp/outputfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  int pid = fork ();
  if (pid == 0) {
     ++argv;
     dup2 (outfd, 1);
     dup2 (outfd, 2);
     execvp (argv[0], argv);
  } else if (pid > 0) {
     int loc;
     wait (&loc);
     fprintf (stderr, "program %s terminated, 0x%x\n", argv[1], loc);
  }
}
```

```
ians-macbook-air:MUC13 igb$ date; ./exec date
Tue 18 Feb 2014 10:32:53 GMT
program date terminated, 0x0
ians-macbook-air:MUC13 igb$ cat /tmp/outputfile
Tue 18 Feb 2014 10:32:53 GMT
ians-macbook-air:MUC13 igb$
```

This is the “old way” to…

- Write a daemon (open a socket, listen on that socket, when a new connection arrives fork and process the connection, in the parent close and got back to listening)

- Process in the background (fork, start the background task going, communicate via interprocess-communication)

Interprocess Communications

- pipes: pseudo files that can be written in one process and read in another process, provided the processes are related
- named pipes (aka fifos) which can be found in the filesystem, although they are not files
- Unix domain sockets, which are bidirectional
- Message queues
- Using TCP (or other networking) in loopback
- etc

Pipes

```
int
main (int argc, char **argv) {
  int pipefd[2];
  pipe (pipefd);
  int pid = fork ();
  if (pid == 0) {
     close (pipefd[0]);
     dup2 (pipefd[1], 1);
     dup2 (pipefd[1], 2);
     fprintf (stdout, "Hello to the other process\n");
     exit (0);
  } else if (pid > 0) {
     char buffer[BUFSIZ];
     char *cr;
     int bytes;
     close (pipefd[1]);
     while ((bytes = read (pipefd[0], buffer, sizeof (buffer))) > 0) {
       buffer[bytes - 1] = '\0';
       fprintf (stdout, "the other end sent me: \"%s\" (that was nice)\n", buffer);
     }
     int loc;
     wait (&loc);
     fprintf (stderr, "other end terminated, 0x%x\n", loc);
  }
}
```

```
ians-macbook-air:MUC13 igb$ ./pipe
the other end sent me: "Hello to the other process" (that was nice)
other end terminated, 0x0
ians-macbook-air:MUC13 igb$
```

What you really want is two “things” working on the same memory

- Unix has explicit “shared memory”, which is a legacy of work done by AT&T in the late 1980s and is usually regarded as a bit messy.
  - Two processes can obtain a “key” to use a region of memory as though it is their own. Shared memory has its own namespace.
  - There are some locking facilities available, in the shape of semaphores, again in their own namespace.
- It’s also possible to do this with creative use of `mmap()`

Hence threads

- Multiple **threads** run in
  - the same address space
  - over the same memory
  - but with their own stack and their own program counter
- So method-local variables are thread-local, while global (or class-local) variables are shared between all threads

Java (general-style)

- A class is written which implements “Runnable” or extends “Thread”
- In each case, it has a “start” method which starts it running in parallel with the lead thread.
- “synchronized” methods or “synchronized (obj) {…}” statements provide mutual exclusion
- You get to write portable code, but quite what is does is not clearly defined

Threads: Why?

- We said at the outset that we want threads so that we can block on some event while continuing to execute code

- But this isn’t the only reason.

Clock Speeds Versus Core Count

- There have for a long time been multiple-processor computers to get more power out of the same clockspeed.
    - VAX 11/782 is two VAX 11/780s in a box running VMS or various Unix derivatives
    - Honeywell DPS-8/70M could go up to six processors running Multics
    - Sequent Symmetry and its successors used up to 30 x86 processors running a hacked Unix
    - Sun Starfires had up to 64 (later 128) UltraSparc processors
- All of these are pretty exotic for their era

The problem

- The problem is that power consumption and heat usually scale non-linearly with clockspeed, and therefore doubling the clockspeed is much harder than doubling the core-count on one die.

- A common trick is to manufacture dies with (say) 8 cores in them, and then sell them as 1-, 2-, 4-, 6- and 8-way processors depending on how many actually work.

So…

- A typical machine, particularly one which needs to be power- and heat-efficient, will not have one fast processor.
- Rather, it will have multiple processors, each of them rather lower powered than the total claim the manufacturer makes.
- A single-threaded, compute-bound application will only be able to take advantage of one core.
- Conversely, an application which does only run on one core will permit other applications to run smoothly, and will (in isolation) reduce power consumption.

Running One Thread

- If you just start() a thread, you have little control over how it executes. But on Android:
  - The run() method is called within the Runnable
  - The run() method runs on another thread, that is not the UI thread
  - The thread is ideally deprioritised

```
android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);
```

Runnable does not run on the UI Thread

- A Runnable thread invoked via start() cannot directly manipulate the UI objects
  - If you want to do this easily, you use an AsyncThread as already described to run one Runnable and have the magic for handing things to the UI done for you
  - If you want to do this from a more complex thread object, you need to use Handlers and Messages, which we will discuss next week

Running multiple threads

- The fastest way to perform a computation or data reduction is to run a thread on each core
  - Switching between threads wastes resources on that core
  - Running fewer threads “wastes” (at least, leaves idle) cores.
- Standard pattern is to have a pool of threads, and execute tasks from a queue of Runnables

ThreadPoolExecutor

- Simple interfaces:
  - newCachedThreadPool () – start new threads, but reclaim ones that have finished
  - newFixedThreadPool(int) – fixed size pool
  - newSingleThreadExecutor() – run one thread in the background, but with code that is easily changed to something else
- ThreadPoolExecutor(…) allows you to tune everything: pool size, queuing, reclamation, scheduling…

What about processes?

- You can group threads into processes and control which process contains which thread, and then use IPC to communicate between them

- If you need to do this, the android:process element in the application manifest allows you to assign activities to distinct processes

OK, so what might use this?

- Services – long-running activities which are not necessarily part of just one application, and are not killed when they aren't on screen

- Services can be started so that they run indefinitely, potentially in a process of their own

- Services can be bound so that they run for as long as the service is being used, and then are reclaimed.

- An obvious example would be a location service of some sort.

Intents

- They are instructions to perform an action, or that something that happened which might trigger an action

- An Intent is an action, plus an action-specific URI as additional data (a number to call, a contact to display).

- BroadcastReceivers receive intents that have been broadcast, and therefore filter which ones they get.

- Services have intents unicast to them.

Service methods

- Services subclass Service or IntentService,
- Service overrides:
  - onStartCommand() for when someone called startService(Intent)
  - onBind() for when someone calls BindService(Intent)
  - onCreate() to be called on startup
  - onDestroy() for when the service shuts down

Problems with Service

- The service runs on the main thread, so needs to create a new thread for each start request (ie, each application that starts the service) unless you know it’s going to complete quickly
- This is like a Unix daemon, if you have written one.
- Without care, it can spawn a lot of threads.

IntentService

- IntentService creates one thread, and queues start requests up. It restricts the resources that can be used.

- Just has to override OnHandleIntent

IntentService sample

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super IntentService(String)
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      long endTime = System.currentTimeMillis() + 5*1000;
      while (System.currentTimeMillis() < endTime) {
          synchronized (this) {
              try {
                  wait(endTime - System.currentTimeMillis());
              } catch (Exception e) {
              }
          }
      }
  }
}
```

Bound Service Example

```
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder mBinder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();

    /**
     * Class used for the client Binder.  Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public
methods
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    /** method for clients */
    public int getRandomNumber() {
      return mGenerator.nextInt(100);
    }
}
```

BroadcastReceivers

- This all assumes that we are going to run services on request from clients

- Another common pattern is that a service performs some task, and then tells some number of other applications (which the service may not know by name) that something has happened.

- Example: geofencing

- Example: Battery Low condition

System Intents

```
<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >

    <receiver android:name="MyReceiver">
        <intent-filter>
            <action android:name="android.intent.action.BATTERY_LOW">
        </action>
        </intent-filter>
      </receiver>

  </application>
```

```
public class MyReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();
    }

}
```

Custom Intents

```
public void broadcastIntent(View view)
{
   Intent intent = new Intent();
   intent.setAction("com.tutorialspoint.CUSTOM_INTENT");
   sendBroadcast(intent);
}
```

```
<application
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/AppTheme" >

   <receiver android:name="MyReceiver">
      <intent-filter>
         <action android:name="com.tutorialspoint.CUSTOM_INTENT">
      </action>
      </intent-filter>
   </receiver>

</application>
```

Intents can carry data

- Constructor can include a URI, which you build according to your requirements

- We will come back to intents next week, when we talk about using them to chain applications together.