ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 17 — February 11, 2013

Patrick Lam *version 1*

In Lecture 6 (an Android Friday) we talked about Handlers in Android, which implement the notion of a `Timer`. That discussion was very hands-on and focussed on how to code a timer. Today we'll talk about how to use a timer, and its relationship to polling and other embedded systems concepts.

Timers

Recall that in *periodic polling*, the software polls a device at fixed intervals. The usual way of implementing periodic polling is through timers. Two potential applications of timers:

- perform a task occasionally (e.g. raising alarms, or polling); or
- reset a system that's gotten stuck.

In Assignment 2, you practiced using timers on Android.

Skill. In today's lecture, you'll learn how to use Java timers to implement interval timers and watchdog timers. You'll also learn about the difference between Android timers and Java timers.

Interval Timers. Interval timers are appropriate for recurring tasks. For instance, one could implement light sensor polling with an interval timer (but you don't need to do this in Android, since the system sends you events when the value changes). We've seen the use of `Handler` to implement interval timers, which performs tasks occasionally. You can always use more than one timer at a time, each with different frequencies.

Reminder: Here's some code for executing an infinitely recurring task using `Handler`:

```
Handler h = new Handler();
Runnable r = new Runnable() {
  public void run() {
    // execute the task
    h.postDelayed(this, delayInMS);
  }
};
h.postDelayed(r, delayInMS);
```

By the way, you can cancel an upcoming task like this:

```
h.removeCallbacks(r);
```

True Interval Timers. Strictly speaking, the timers we'll be implementing aren't actually interval timers. A true interval timer interrupts the processor, rather than waiting for the executing thread to become free.

We can simulate an interval timer more closely using the `Timer` class[1]. Such a timer executes in a different thread, so it doesn't need to wait for processor availability. A serious problem is that other threads can't do updates to the user interface (UI). Timers have more overhead and generally aren't the right thing on Android, but you should know about them.

Here's how you can use `Timer` successfully. You can put this code anywhere, including `onCreate()` or in an on-click listener.

```
Timer t = new Timer();
t.schedule(new TimerTask() {
  @Override
  public void run() {
    runOnUiThread(timerTick);
  }
}, firstDelayMS, repeatIntervalMS);
```

You are creating a Java `Timer` object, which takes a delay-before-first-event and a repeat interval, both in milliseconds. If you omit the repeat interval, the `Timer` only fires once. In the `run()` method of the `TimerTask` (again, event-oriented programming), you are asking Android to run the `timerTick` `Runnable` on the UI thread. If you just call `timerTick.run()`, your app will crash!

Let's also create the `timerTick` object.

```
Runnable timerTick = new Runnable() {
  @Override
  public void run() {
    Toast.makeText(getApplicationContext(),
                   "ding!", Toast.LENGTH_SHORT).show();
  }
};
```

A recap: you have to do the following to set up a Java timer.

- instantiate a new `Timer` object;
- schedule a `TimerTask` on that `Timer`;
- inside the `TimerTask`, invoke a `Runnable` to run on the UI thread; and
- implement the `run()` method on the `Runnable` to effect UI changes.

Watchdog Timers. The other timer application I mentioned is that of a watchdog timer, which can observe that a system appears to be stuck and take some action.

You can build a watchdog timer using an interval timer.

- Set the timer interval to be the largest allowable time for a task to take. Start the timer.
- If the timer fires, the task took too long.

[1]`http://steve.odyfamily.com/?p=12`, accessed February 3, 2013.

- The timer event handler should still execute, so it can deal with the situation (for instance, by killing the task.)

Why wouldn't the event handler execute?

When designing an embedded system, you might use specialized hardware for a watchdog timer. This gives it some resilience to failures in the rest of the system. The fact that `Timer`s run in separate threads also give them some resilience.

Here's an example of an Android analogy of a watchdog timer.

```
class MainActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        final Timer watchdogTimer = new Timer();

        // When this task is run, finish the activity.
        final Runnable timerTick = new Runnable() {
          @Override
          public void run() {
            MainActivity.this.finish();
          }
        };

        // User has 10 seconds to complete this page, otherwise finish.
        watchdogTimer.schedule(new TimerTask() {
          @Override
          public void run() {
            runOnUiThread(timerTick);
          }
        }, 10000);

        Button b = (Button) findViewById(R.id.next_page);
        b.setOnClickListener(new View.OnClickListener() {
          @Override
          public void onClick(View arg0) {
            watchdogTimer.cancel();
            Intent browserIntent = new Intent(Intent.ACTION_VIEW);
            browserIntent.setData(Uri.parse("http://www.patricklam.ca"));
            startActivity(browserIntent);
          }
        });
    }
}
```

Possible Outputs. If the user clicks on the button fast enough, the app navigates to my webpage. Otherwise, the app's main activity finishes.

What sorts of tasks might you be waiting on, besides user input?