

Lecture 6 – Events, Polling vs. Interrupts

Patrick Lam & Jeff Zarnett

`p.lam@ece.uwaterloo.ca` & `jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Be able to program for systems which use event-based models (eg Android).

In 10 minutes, please remind me that I'm supposed to do something.

What happens when you press this?



What happens when you press this?



Android sends an **event** to the **event listener**.

An *event* is a notification of a change to the state of your system.

Reactive, not proactive.

- To receive click events:
the application registers an event listener with the object representing the button.
- When the user clicks the button:
the system executes the click event listener.

- To receive click events:
 - the application registers an event listener with the object representing the button.
`go.setOnClickListener(...);`
- When the user clicks the button:
 - the system executes the click event listener.

Implementing Event Listeners (painfully)

We need to pass something to `setOnClickListener()`. What?

This method takes a `View.OnClickListener` object.

You could declare one:

```
class MyClickListener
    extends View.OnClickListener {
    public void onClick(View v) {
        Log.d("A2", "clicked!");
    }
}

...
go.setOnClickListener(new MyClickListener());
```

```
go.setOnClickListener(new View.OnClickListener() {  
    public void onClick(View v) {  
        Log.d("A2", "clicked!");  
    }  
});
```

This is called an **inner class**.

Advantages of Inner Classes

```
class MainActivity {  
    int i;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        Button go = (Button) findViewById(R.id.go);  
        final int j = 2;  
        go.setOnClickListener(new View.OnClickListener() {  
            public void onClick(View v) {  
                Log.d("A2", "i is "+i+" and j is "+j);  
            }  
        });  
    }  
}
```

- They don't litter your code with one-time-use classes.
- They can access fields and (final) local variables.

You have another option. From the Android documentation¹:

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:text="@string/self_destruct"
    android:onClick="selfDestruct" />
```

Then, in your activity, you must include the method:

```
public void selfDestruct(View view) {
    // Kaboom
}
```

¹<http://developer.android.com/reference/android/widget/Button.html>

We've been programming with **callback methods**.

This is also known as “inversion of control”.

Key idea: system (user) decides what happens when.

You can also structure your program with callback methods. Say you have a time-consuming task (TCT).

- 1 register a callback upon completion of TCT;
- 2 spawn the TCT in another thread, don't wait for it;
- 3 continue normally.

Once the TCT finishes, the callback notifies the main application, which collects results.

Also known as asynchronous, or non-blocking, execution.

Synchronous versus Asynchronous Execution

ECE150: Synchronous, or sequential, programs:

- all instructions execute in sequence;
- an instruction only executes after its predecessor completes.

Also true for function calls.

ECE155, ECE254: Asynchronous, or concurrent, programs:

- most instructions execute in sequence; but
- main program may spawn a function to run concurrently with it.
- Communication via shared memory or via events.

Permits higher performance on multicores, or more relevant structuring. Callbacks are a tool.

Where do events come from?

Imagine the following situation:

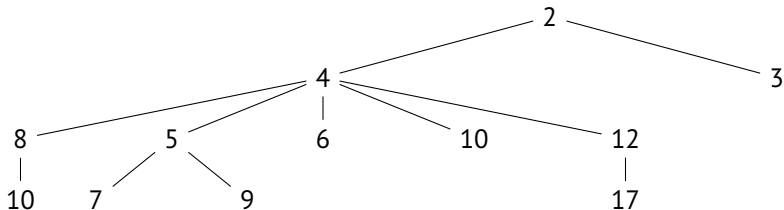
- your mom calls;
- supper is burning;
- laundry is done.

What do you do?

Implementing Priorities

Associate a priority with each event.

Use a *priority queue* data structure to get the highest-priority event.



Remember: reactive, not proactive.

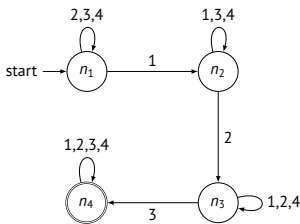
How can the application do what it wants?

Use **Finite-State Machines!**

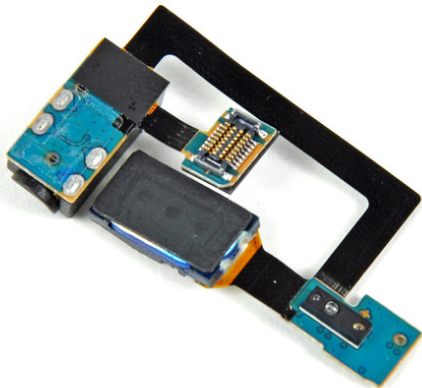
Remember: reactive, not proactive.

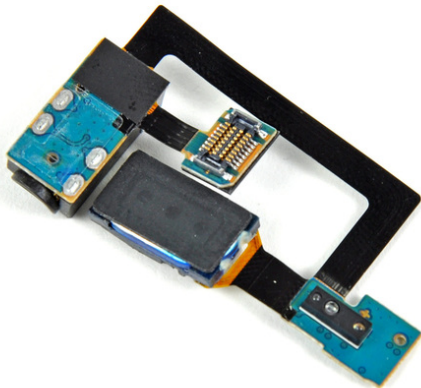
How can the application do what it wants?

Use **Finite-State Machines**!



Where Events Come From





Analog-to-Digital Converter. Then what?

“Are we there yet?”

“Are we there yet?”

— example of **polling**.

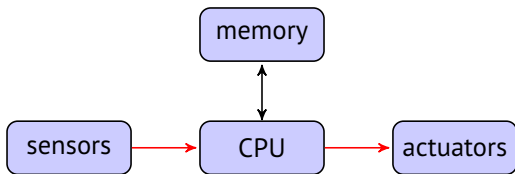
Polling: processor requests readings from the device at its convenience.

“What is the current light level?”

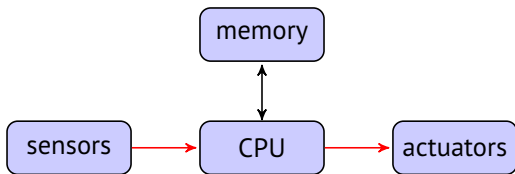
(also known as “passive synchronization”)

It depends:

- whenever convenient (occasional polling);
- at fixed time intervals (periodic polling); or
- constantly (tight polling).



What's the mechanism?



What's the mechanism?

- port-mapped I/O; or,
- memory-mapped I/O

Port-mapped I/O: Special CPU Instructions

The Intel ia32 processors provide special I/O instructions:

```
outb    ax, 0x3f8  
inw     dx, ax
```

May use a special bus, or set a specific signal on the bus.

Memory-mapped I/O Example

CPU just reads and writes to “memory”.

```
while (statusRegister == 0x0000) {  
    // Do nothing until statusRegister changes value  
}  
// Read data that has changed from a dataRegister  
// and store in memory  
incomingData = dataRegister;
```

Devices listen on the bus and respond.

Memory-mapped I/O Example

```
while (statusRegister == 0x0000) {  
    // Do nothing until statusRegister changes value  
}  
// Read data that has changed from a dataRegister  
// and store in memory  
incomingData = dataRegister;
```

This is a *tight polling loop*.

- Expect the hardware specification to promise that statusRegister eventually changes due to an external event.
- Data exchange occurs once device is ready: **polling synchronization**.

So far: processor controls when to read data from a device.

Instead: device may tell the processor when device is ready, using an **interrupt**.

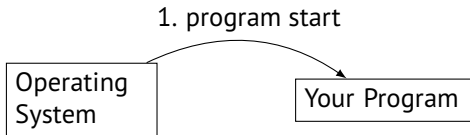
This constitutes **active synchronization**.

Interrupt tells the processor: “Something’s happening!”

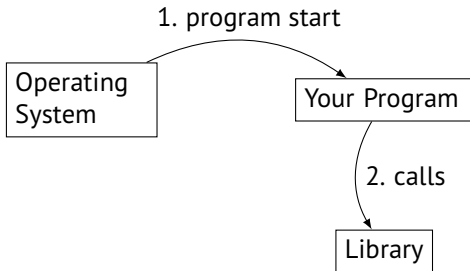
Upon receipt of an interrupt, the processor:

- stops what it’s currently doing and saves its state;
- starts executing pre-defined **interrupt handler**, which:
 - reads the event information; and
 - stores it somewhere accessible.
- upon return from handler, resumes previous state.

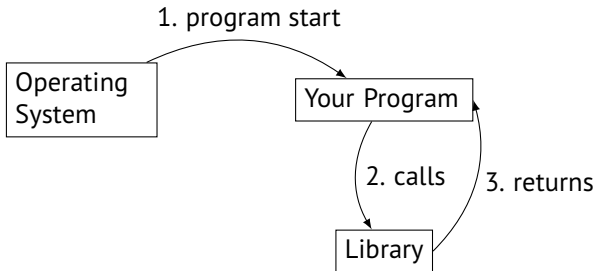
Old ECE150 paradigm:



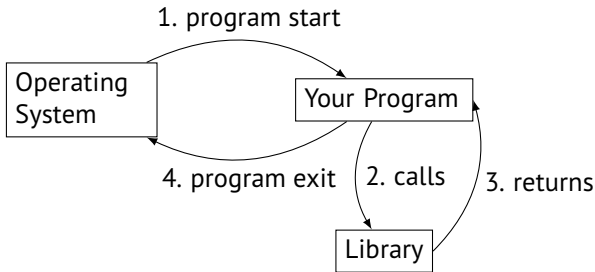
Old ECE150 paradigm:



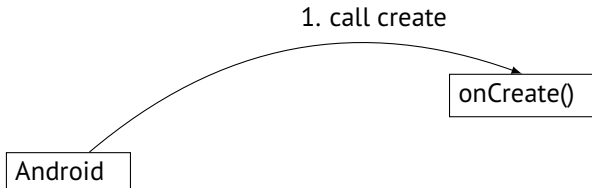
Old ECE150 paradigm:



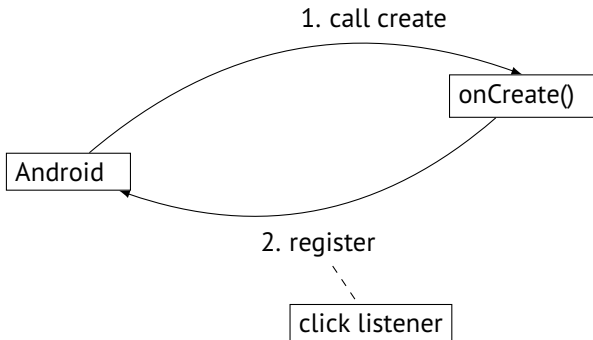
Old ECE150 paradigm:



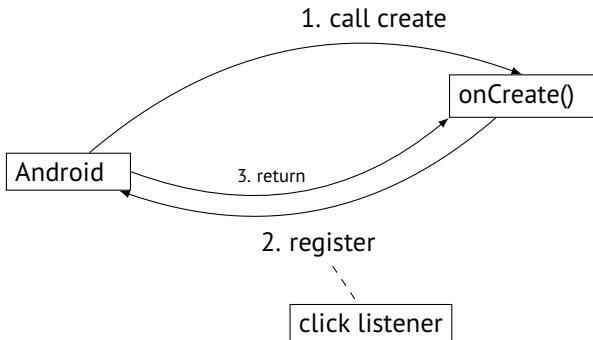
New event-driven ECE155 paradigm:



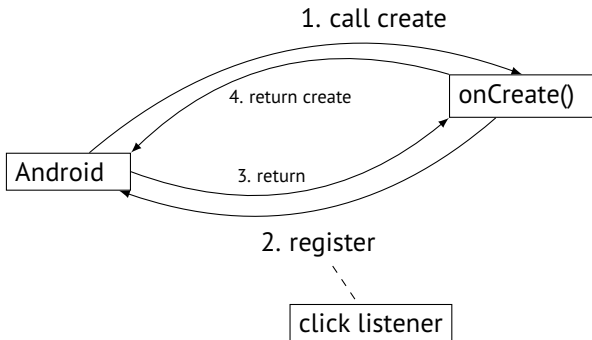
New event-driven ECE155 paradigm:



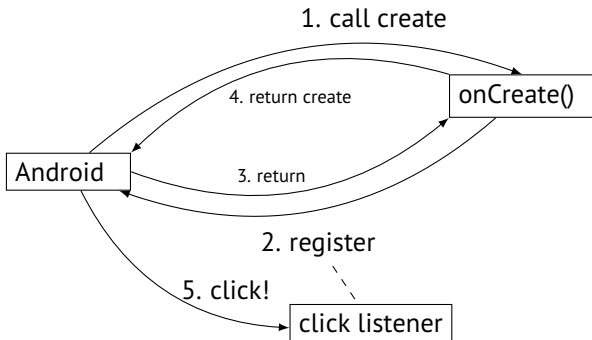
New event-driven ECE155 paradigm:



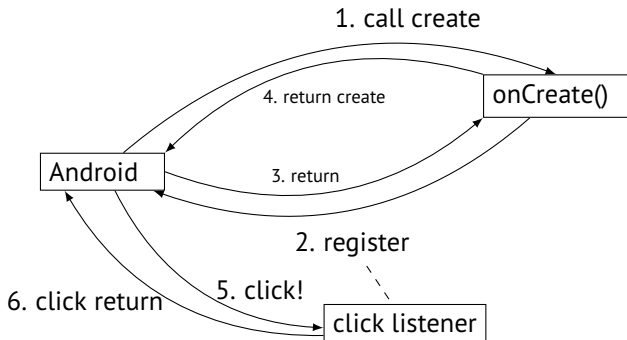
New event-driven ECE155 paradigm:



New event-driven ECE155 paradigm:



New event-driven ECE155 paradigm:



Behind the Scenes for Inversion of Control

Android is running an event loop for each thread:

```
while (!done) {  
  r <- fetch Runnable from Queue  
  dispatch r  
}
```

This is a polling loop: in particular, a **tight polling loop**, but which goes to sleep waiting for the next event (in fetch).

Must respond to an external event in a fixed amount of time.

This fixed amount of time is not necessarily small.

- may potentially be fixed and large.

Many embedded systems must satisfy real-time constraints.

In upper-year courses, you'll see both embedded systems and real-time systems in more detail.

Real-Time System Example



(credit digitaljournal . com, from flickr)

Blu-Ray player must:

- read compressed video data from a media disk;
- decompress the video; and
- output it to a HDMI interface,

all within a fixed amount of time, to avoid a degradation of video quality.