Timesharing Operating Systems

• All modern operating systems are capable of sharing processor(s) between multiple users and processes
  – Process scheduling algorithms

• There are often lots (hundreds) of processes running
  – Many are managed by the system
  – Some are user executed
Processes vs Threads

• A process is typically defined as execution of a single program on a computing device
  – Processes are scheduled by the OS
  – Processes are owned by a user or the OS (root)
  – Processes may have a long lifetime (a web server) or a short lifetime (one run of a short program)
  – Processes do not share memory between each other
  – Most OS support inter-process communication
Processes vs Threads

• A *thread* is a component of a process
  – Usually considered the smallest unit that can be scheduled by an OS
  – Can share memory and instructions with other threads in the same process
  – Most OS support inter-thread communication
Timesharing Operating Systems

• On a single processor system,
  – “Round-robin” scheduling system switches between threads and processes fast enough that it appears that multiple programs are running
    – Only one is every truly executing at a time
• On multi-core or multi-processor systems,
  – Threads or processes can be allocated across the processors
    • True parallel processing
  – “Round-robin” scheduling still occurs
Slow Android Apps

• By default on Android, all work is done in a single thread, the “main application” thread

• If a component of the work takes a long time, the rest of the work will be “blocked”
  – For example, a long time to access data across the network prevents responding to any GUI events

• In the Android OS, if a GUI doesn’t respond to an input event in < five seconds, then it is considered *unresponsive* and the OS will try to **kill it**!
Android Thread Design

• Only perform GUI actions on main application thread
• Spawn separate threads to perform data-intensive or slow actions
• Make these threads asynchronous
  – First (main) thread does not have to wait for/check on other threads
  – Instead, those threads run as they need to and report back to the original thread
• Any changes made to the UI should go through the UI thread
Threads

• A thread (in Java):
  – Can belong to a ThreadGroup (a group of related threads)
    • We’ll ignore this for now
  – Has a target Runnable implementation
    • This is the code run on the thread
    • See next slide
  – Has a name of your choosing
  – Initiated by calling \textit{start} method

Think of a thread as a generic container that can hold any work we want to do – we can associate it with any Runnable
Runnables

• A Runnable is an object that implements a `run` method... and that’s pretty much it.

• We will employ in examples as an anonymous inner class
  – A Runnable class that we define “on-the-fly”
  – Has full access to variables and methods in our class
Runnable Example Code

```java
public class ThreadExample1Activity extends Activity implements View.OnClickListener {

    Button startButton;
    Runnable countingRunnable;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        startButton = (Button)findViewByID(R.id.startButton);
        startButton.setOnClickListener(this);

        countingRunnable = new Runnable() {
            public void run() {
                count("Counter2");
            }
        };
    }

    public void onClick(View view) {
        Thread countingThread = new Thread(null, countingRunnable, "Counter2");
        countingThread.start();
        count("Counter1");
    }

    public void count(String name) {
        for (int j = 0; j < 10000000; j++) {
            if ((j % 1000000) == 0)
                Log.v(name, "TICK " + j);
        }
    }
}
```

Declare a Runnable variable

- Define an anon inner class which supports Runnable interface method `run`
- Allocate and assign to Runnable variable

- When button is pressed, spawn a 2\textsuperscript{nd} thread (countingThread) to run the runnable
- Do same counting on GUI thread as well

`count` method counts to 10 million, outputting every million

Function `count()` is visible to and accessed by both threads!
Runnable Example Output

Note the ordering of Counter1 and Counter2 changes
They are not synchronized (notice differences in time)
Handlers

• Notice, I was writing to the Log output, not to the GUI

• GUI components are not considered “thread-safe”
  – Manipulation by two or more threads concurrently can cause the underlying data to become incorrectly managed
Handlers

• A **Handler**
  – Can be associated with each thread
    • Associated with thread active when allocated
  – Is similar to an administrative assistant for an executive
    • Receives messages for the executive
    • Can schedule activities for the executive
  – Other threads can post Runnable code to be executed by the receiving Thread
Handlers

• Typical use of a Handler:
  – Create a handler for main GUI thread
  – Create a thread for some intensive background work, employing a Runnable
  – When that background thread is done with its work, it can post a 2\textsuperscript{nd} Runnable on the main GUI handler indicating an update it requests to be performed on the GUI
  – The actual (2\textsuperscript{nd}) Runnable received will be run by the main thread, ensuring only one thread is changing the GUI
Handler Example

public class HandlerExampleActivity extends Activity implements View.OnClickListener {

    Button button;
    TextView textView;
    Handler guiHandler;
    Runnable countingRunnable;
    Runnable updateRunnable;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        button = (Button) findViewById(R.id.button);
        textView = (TextView) findViewById(R.id.text_view);
        button.setOnClickListener(this);

        // handler for GUI thread
        guiHandler = new Handler();

        // code that GUI thread should run (requested by 2nd thread)
        updateRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "Counter2 Done");
            }
        };

        // actual 2nd thread work, and request to update GUI
        countingRunnable = new Runnable() {
            public void run() {
                count("Counter2");
                guiHandler.post(updateRunnable);
            }
        };
    }

    public void onClick(View view) {
        // start the 2nd thread counting
        Thread countingThread = new Thread(null, countingRunnable, "BackgroundCounter");
        countingThread.start();

        // the GUI thread also does counting
        count("Counter1");
        // the GUI thread can directly access the GUI
        textView.setText(textView.getText() + "\n" + "Counter1 Done");
    }

    public void count(String name) {
        for (int j = 0; j < 10000000; j++)
        {
            if ((j % 1000000) == 0) {
                Log.v(name, "TICK " + j);
            }
        }
    }

    This code still “freezes”
    the button, as the
    GUI is also counting
    See next slide...
Handler Example

In this example, GUI thread doesn’t count – button is responsive; I can trigger interleaved counting threads

```java
public class HandlerExampleActivity extends Activity implements View.OnClickListener {

    Button button;
    TextView textView;
    Handler guiHandler;
    Runnable countingRunnable;
    Runnable updateStartRunnable;
    Runnable updateDoneRunnable;

    public void onClick(View view) {

        // start the 2nd thread counting
        Thread countingThread = new Thread(null, countingRunnable, "BackgroundCounter");
        countingThread.start();
    }

    public void count(String name) {
        for (int j = 0; j < 10000000; j++) {
            if ((j % 1000000) == 0) {
                Log.v(name, "TICK " + j);
            }
        }
    }

    public void updateStart() {
        updateStartRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Start");
            }
        };
    }

    public void updateDone() {
        updateDoneRunnable = new Runnable() {
            public void run() {
                textView.setText(textView.getText() + "\n" + "CounterThread Done");
            }
        };
    }

    // actual 2nd thread work, and request to update GUI
    countingRunnable = new Runnable() {
        public void run() {
            guiHandler.post(updateStartRunnable);
            count("CounterThread");
            guiHandler.post(updateDoneRunnable);
        }
    };
}
```