THE UNIVERSITY OF MELBOURNE

COMP90018

Mobile Computing Systems Programming

WEEK 3 – Threads and Service

Atiq Shaikh

atiq.shaikh@unimelb.edu.au

Intended Learning Outcomes

- Threads
- Services

Threads (Java)

Threads – What is Thread in Java?

- A thread is a ***lightweight*** subprocess, the ***smallest unit of processing***;
- Threads are ***independent***. *(If there occurs exception in one thread, it doesn't affect other threads.)*
- It uses ***a shared memory area***.

Threads – Why Thread is important?

Multitasking

Multitasking is the concurrent execution of ***multiple tasks*** over a certain period of time.

New tasks can ***interrupt already started ones before they finish***, *instead of waiting for them to end.*

Threads

- *Android*

Threads in Android

By default, all components *(activity, service, receiver and provider)* of the same application run in the ***same process*** *and* ***thread (called the "main" thread)***.

https://developer.android.com/guide/components/processes-and-threads.html

Threads in Android

*However, you can create **additional threads** for **any process**.*

Threads in Android

Two important principles:

1. ***Do not block (overburden) the UI thread;***

Threads in Android

When the UI thread of an Android app is ***blocked for too long***, an ***"Application Not Responding" (ANR)*** error is triggered.

https://developer.android.com/topic/performance/vitals/anr

Threads in Android

Another important principle comes from:

What will happen when ***other threads*** *access a* ***TextView from the main (UI) thread?***

Threads in Android

2. ***Do not access UI from outside the UI thread***

Threads

- *Communication between Threads*

Threads Communication *(Message via Handler)*

1. *Message;*
2. *Handler;*
3. *MessageQueue;*
4. *Looper*

Message

Message:

Defines a message containing ***a description and arbitrary data object*** *that can be sent to a Handler.*

https://developer.android.com/reference/android/os/Message

Handler

Handler:

*A Handler allows you to **send and process** Message associated with a thread's MessageQueue.*

- *Send and post a message: post(), sendMessage(), …;*
- *Process a message: handlerMessage()*

https://developer.android.com/reference/android/os/Handler.html

MessageQueue

MessageQueue:

Low-level class holding the list of messages to be dispatched by a Looper.

https://developer.android.com/reference/android/os/MessageQueue.html

Looper

Looper:

Class used to run a message loop for a thread.

Looper is a worker that keeps a thread alive, loops through MessageQueue and sends messages to the corresponding handler to process.

https://developer.android.com/reference/android/os/Looper.html

Demonstration

```java
private Handler handler = handleMessage(msg) -> {
        switch (msg.what) {
            case UPDATE_TEXT_RUNNABLE: {
                text.setText("Receive the message from other threads implements Runnable Interface!");
                break;
            }
```

MainActivity.java

```java
new Thread((Runnable) () -> {
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Send the Message object towards UI threads
        Message message = new Message();
        message.what = UPDATE_TEXT_RUNNABLE;
        handler.sendMessage(message);
}).start();
```

Can be anywhere
(when holding handler instance)

Looper does not need to initialize in the main thread.

Threads Communication (*EventBus*)

EventBus is an open-source library for Android and Java using the ***publisher/subscriber*** pattern for loose coupling.

http://greenrobot.org/eventbus/

https://github.com/greenrobot/EventBus

EventBus –Library Setting

build.gradle(app level)

EventBus


```java
public class MessageEvent {

    public final String message;

    public MessageEvent(String message) {
        this.message = message;
    }
}
```

MessageEvent.java

Step 1: Define events

Events are *POJO (plain old Java object)* without any specific requirements.

EventBus


```
@Subscribe(threadMode = ThreadMode.MAIN)
public void onMessageEvent(MessageEvent event) {
    text.setText(event.message);
}
```

MainActivity.java

Step 2: Prepare subscribers

Subscribers implement ***event handling methods*** (also called "subscriber methods") that will be *called when an event is posted*.

These are defined with the @Subscribe annotation.

EventBus


```java
@Override
public void onStart() {
    super.onStart();
    EventBus.getDefault().register( subscriber: this);
}

@Override
public void onStop() {
    EventBus.getDefault().unregister( subscriber: this);
    super.onStop();
}
```

MainActivity.java

Step 2: Prepare subscribers

Subscribers also need to ***register themselves to and unregister from the bus***. Only while subscribers are registered, they ***will receive events.***

In Android, in activities and fragments you should usually register ***according to their life cycle***.

EventBus


```
(Thread) run() -> {
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        EventBus.getDefault().post(new MessageEvent("Receive the message from other threads extends Thread through EventBus!"));
}.start();
```

Can be anywhere

Step 3: Post events

Post an event from any part of your code.

All currently registered subscribers *matching the event type will receive it*.

Threads

- *Implements in Android*

Implements

- *Thread/Runnable*
- *FutureTask*
- *AsyncTask*
- *HandlerThread*
- *IntentService*
- ……

Implements – Thread

```java
new Thread() {
    @Override
    public void run() {
        try {
            Thread.sleep(millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Send the Message object towards UI threads
        Message message = new Message();
        message.what = UPDATE_TEXT_THREAD;
        handler.sendMessage(message);
    }
}.start();
```

Subclass of Thread

Declare a class to be a subclass of Thread. This subclass should ***override the run method*** of class Thread.

https://developer.android.com/reference/java/lang/Thread

Implements – Runnable

```java
new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            Thread.sleep(millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Send the Message object towards UI threads
        Message message = new Message();
        message.what = UPDATE_TEXT_RUNNABLE;
        handler.sendMessage(message);
    }
}).start();
```

Implements the Runnable interface

Create a thread is to declare a class that implements the Runnable interface. That class then implements the run method.

https://developer.android.com/reference/java/lang/Thread

Implements – FutureTask

```java
future = new FutureTask<String> new Callable<String>() {
    @Override
    public String call() throws Exception {
        Thread.sleep( millis: 2000);
        return "Receive the message from other threads by FutureTask!";
    }
}) {
    @Override
    protected void done() {
        String textContent = null;
        try {
            textContent = future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        Message message = new Message();
        message.what = UPDATE_TEXT_FUTURETASK;
        message.obj = textContent;
        handler.sendMessage(message);
    }
};
new Thread(future).start();
```

Thread with FutureTask

This class provides a base implementation of ***Future***, with methods to ***start and cancel*** a computation, ***query*** to see if the computation is ***complete***, and ***retrieve the result*** of the computation.

https://developer.android.com/reference/java/util/concurrent/FutureTask

For Reading

Callable

https://developer.android.com/reference/java/util/concurrent/Callable

Future

https://developer.android.com/reference/java/util/concurrent/Future

RunnableFuture

https://developer.android.com/reference/java/util/concurrent/RunnableFuture

FutureTask

https://developer.android.com/reference/java/util/concurrent/FutureTask

Implements – AsyncTask

```java
new AsyncTask<Integer, Integer, String>() {
    @Override
    protected String doInBackground(Integer... integers) {
        for (int i = 0; i < integers[0]; i++) {
            try {
                Thread.sleep( millis: 200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            publishProgress(i);
        }
        return "Receive the message from other threads by AsyncTask!";
    }

    @Override
    protected void onProgressUpdate(Integer... progress) {
        text.setText("Progress: " + progress[0] * 10 + "%");
        super.onProgressUpdate(progress);
    }

    @Override
    protected void onPostExecute(String result) { text.setText(result); }
.execute(10);
break;
```

Instance of AsyncTask

This class allows you to ***perform background operations and publish results*** on the ***UI thread*** without having to manipulate threads and/or handlers.

https://developer.android.com/reference/java/lang/Thread

Implements – HandlerThread

```java
exampleHandlerThread = new HandlerThread( name: "example");
exampleHandlerThread.start();

Message msg = new Message();
msg.what = UPDATE_TEXT_HANDLER_SEND;

new Handler(exampleHandlerThread.getLooper(), new Handler.Callback() {
    @Override
    public boolean handleMessage(Message message) {
        switch (message.what) {
            case UPDATE_TEXT_HANDLER_SEND: {
                try {
                    Thread.sleep( millis: 2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                Message msg = new Message();
                msg.what = UPDATE_TEXT_HANDLER_RECEIVE;
                handler.sendMessage(msg);
                break;
            }
        }
        return false;
    }
}).sendMessage(msg);
```

Instance of HandlerThread

A Thread that ***has a Looper***. The Looper can then be used to ***create Handlers***.

(Note that just like with a regular Thread, Thread.start() must still be called.)

https://developer.android.com/reference/java/lang/Thread

For Reading – Thread Guide

Making adept use of threads on Android can help you boost your app's performance.

The page discusses several aspects of working with threads:

1. ***working with the UI**, or main, thread;*
2. *the **relationship** between app lifecycle and thread priority;*
3. *methods that the platform provides to **help manage thread complexity**.*

https://developer.android.com/topic/performance/threads

For Reading – ThreadPoolExecutor

To automatically run tasks ***as resources become available, or to allow multiple tasks to run at the same time (or both)***, you need to provide a managed collection of threads.

```
ExecutorService executor = new ScheduledThreadPoolExecutor(2);
```

```
executor.execute(Thread implement runnable or override run() method);
```

To do this, use an instance of ***ThreadPoolExecutor***, which runs a task from a queue when a thread in its pool becomes free. To run a task, ***all you have to do is add it to the queue***.

https://developer.android.com/training/multiple-threads/create-threadpool.html

Service

Service – What is Service in Android?

A ***Service*** is an application ***component*** that can ***perform long-running operations*** in the background, and it doesn't provide a user interface.

https://developer.android.com/guide/components/services

Service

- *Create Service*

Create Service

If you do use a service, ***it still runs in your application's main thread*** by default, so you should still ***create a new thread*** within the service if it performs intensive or blocking operations.

However, it is possible to create service in another process by changing android:process.

Create Service

```java
@Override
public void onCreate() {
    super.onCreate();
    Log.d( tag: "MyService",  msg: "onCreate executed");
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.d( tag: "MyService",  msg: "onStartCommand executed");
    return super.onStartCommand(intent, flags, startId);
}
```

MyService.java

```java
Intent startIntent = new Intent( packageContext: this, MyService.class);
startService(startIntent); // Start Service
```

MainActivity.java

Start a Service

The system invokes this method by calling ***startService()*** when another component (such as an activity) requests that the service be started.

When this method executes, the service is ***started and can run in the background indefinitely***.

Create Service

```
@Override
public void onDestroy() {
    super.onDestroy();
    Log.d( tag: "MyService",  msg: "onDestroy executed");
}
```

MyService.java

```
Intent stopIntent = new Intent( packageContext: this, MyService.class);
stopService(stopIntent); // Stop Service
```

MainActivity.jav
a

Stop a Service
The system invokes this method when the service is no longer used and is being destroyed by invoking ***stopService()***.

Your service should implement this to clean up any resources such as ***threads, registered listeners, or receivers***.

Service

- *Bind Service*

Bind Service

A service is bound when an application component binds to it by calling ***bindService()***.

A bound service offers a client-server interface that allows components to ***interact with the service, send requests, receive results***, and even do so across processes with ***interprocess communication (IPC)***.

Bind Service

```java
private DownloadBinder mBinder = new DownloadBinder();

class DownloadBinder extends Binder {

    public void startDownload() { Log.d( tag: "MyService",  msg: "startDownload executed"); }

    public int getProgress() {
        Log.d( tag: "MyService",  msg: "getProgress executed");
        return 0;
    }

}

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

MyService.java

To create a bound service, implement the ***onBind()*** callback method to ***return an IBinder*** that defines the interface for communication with the service.

Bind Service

```java
private DownloadBinder mBinder = new DownloadBinder();

class DownloadBinder extends Binder {

    public void startDownload() { Log.d( tag: "MyService",  msg: "startDownload executed"); }

    public int getProgress() {
        Log.d( tag: "MyService",  msg: "getProgress executed");
        return 0;
    }

}

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

MyService.java

To create a bound service, you must define the interface that specifies ***how a client can communicate with the service***.

This interface between the service and a client must be an implementation of ***IBinder*** and is what your service must return from the ***onBind() callback method***.

Bind Service


```java
private ServiceConnection connection = new ServiceConnection() {

    @Override
    public void onServiceDisconnected(ComponentName name) {
    }

    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        downloadBinder = (MyService.DownloadBinder) service;
        downloadBinder.startDownload();
        downloadBinder.getProgress();
    }
};
```

MainActivity.java

When binding a service, the called component must provide ***an implementation of ServiceConnection***, which ***monitors the connection with the service***.

Bind Service

A client binds to a service by calling ***bindService()***.

```
Intent bindIntent = new Intent( packageContext: this, MyService.class);
bindService(bindIntent, connection, BIND_AUTO_CREATE); // Bind Service
```

MainActivity.java

A client unbinds to a service by calling ***unbindService()***.

```
unbindService(connection); // Unbind Service
```

MainActivity.java

Service

- *Life Cycle Difference between Start and Bind Service*

Life Cycle of Service

A started Service

1. The service is created when another component calls ***startService()***.
2. The service then runs indefinitely and must stop itself by calling ***stopSelf()***. Another component can also stop the service by calling ***stopService()***.
3. When the service is stopped, the system destroys it.

Life Cycle of Service

A bound Service

1. The service is created when another component (a client) calls ***bindService()***. The client then communicates with the service through an IBinder interface.
2. The client can close the connection by calling ***unbindService()***.
3. Multiple clients can bind to the same service and when ***all of them unbind, the system destroys the service***. The service ***does not need to stop itself***.

Life Cycle of Service

Although the figure separates services that are created by ***startService() from those created by bindService()***, keep in mind that any service, no matter how it's started, ***can potentially allow clients to bind to it***.

Service

- *Communicate between Service and other Components*

EventBus

Service Side

```java
@Override
public void onCreate() {
    super.onCreate();

    EventBus.getDefault().register( subscriber: this);
    EventBus.getDefault().post(new MessageEvent(MessageEvent.SERVICE,  message: "Hello from Service!"));
}

@Override
public void onDestroy() {
    EventBus.getDefault().unregister( subscriber: this);
    super.onDestroy();
    Log.d( tag: "MyService",  msg: "onDestroy executed");
}
```

MyService.java

```java
@Subscribe
public void onMessageEventService(MessageEvent event) {
    if (event.type == MessageEvent.ACTIVITY)
        Log.d( tag: "MyService",  msg: "Activity Message Content: " + event.message);
}
```

MyService.java

EventBus

Activity Side

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    EventBus.getDefault().register( subscriber: this);
}

@Override
protected void onStop() {
    EventBus.getDefault().unregister( subscriber: this);
    super.onStop();
}
```

MainActivity.java

```java
@Subscribe(threadMode = ThreadMode.MAIN)
public void onMessageEventActivity(MessageEvent event) {
    if (event.type == MessageEvent.SERVICE) {
        Log.d( tag: "MyService",  msg: "Service Message Content: " + event.message);
        EventBus.getDefault().post(new MessageEvent(MessageEvent.ACTIVITY,  message: "Hello from Activity!"));
    }
}
```

MainActivity.java

EventBus – Result

```
2019-07-20 21:07:16.239 24059-24059/com.example.servicetest D/MyService: Service Message Content: Hello from Service!
2019-07-20 21:07:16.239 24059-24059/com.example.servicetest D/MyService: Activity Message Content: Hello from Activity!
2019-07-20 21:07:16.240 24059-24059/com.example.servicetest D/MyService: onStartCommand executed
```

Broadcast Intent


```
Intent intent = new Intent();
intent.setAction(ACTION_EXAMPLE_END);
intent.putExtra(RESULT_PARAM, value: "Receive the message from other threads by IntentService!");
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

Any Service.java

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(ExampleIntentService.ACTION_EXAMPLE_END);
localBroadcastManager = LocalBroadcastManager.getInstance(this);
broadcastReceiver = (BroadcastReceiver) (context, intent) -> {
        String message = intent.getStringExtra(ExampleIntentService.RESULT_PARAM);
        text.setText(message);
        localBroadcastManager.unregisterReceiver(broadcastReceiver);
};
localBroadcastManager.registerReceiver(broadcastReceiver, intentFilter);
```

MainActivity.java

For Reading – Broadcast

Android apps can ***send or receive broadcast messages*** from the ***Android system and other Android apps***.

These broadcasts are ***sent when an event of interest occurs***, such as when the system boots up or the device starts charging.

https://developer.android.com/guide/components/broadcasts

For Reading – Binder

When creating a service that provides binding, you must ***provide an IBinder*** that provides the ***programming interface*** that clients can ***use to interact with the service***.

https://developer.android.com/guide/components/bound-services#Messenger

Three method to create the interface:

- *Extending the Binder Class*

https://developer.android.com/guide/components/bound-services#Binder

- *Using a Messenger*

https://developer.android.com/guide/components/bound-services#Messenger

- *Using AIDL*

https://developer.android.com/guide/components/aidl.html

Service

- *Foreground Service*

Foreground Service

A foreground service performs some operation that is ***noticeable to the user***. *(For example, an audio app would use a foreground service to play an audio track.)*

Foreground services must ***display a Notification***.

https://developer.android.com/guide/components/services

For Reading – What is a notification?

A notification is ***a message that Android displays outside your app's UI*** to provide the user with reminders, communication from other people, or other timely information from your app.

Users can tap the notification to ***open your app or take an action directly from the notification***.

https://developer.android.com/guide/topics/ui/notifiers/notifications

Foreground Service - Code

```
Intent intent = new Intent( packageContext: this, MainActivity.class);
PendingIntent pi = PendingIntent.getActivity( context: this, requestCode: 0, intent, flags: 0);
Notification notification = new NotificationCompat.Builder( context: this)
        .setContentTitle("This is content title")
        .setContentText("This is content text")
        .setWhen(System.currentTimeMillis())
        .setSmallIcon(R.mipmap.ic_launcher)
        .setLargeIcon(BitmapFactory.decodeResource(getResources(), R.mipmap.ic_launcher))
        .setContentIntent(pi)
        .build();
startForeground( id: 1, notification);
```

MyService.java

Service

- *IntentService*

IntentService

IntentService is a base class for Services that ***handle asynchronous requests*** (expressed as Intents) on demand.

Clients send requests through ***Context.startService(Intent) calls***; the service is started as needed, handles each Intent in turn ***using a worker thread***, and stops itself when it runs out of work.

https://developer.android.com/reference/android/app/IntentService

For Reading – IntentService vs AsyncTask

IntentService allows it to handle ***long-running*** operations without ***affecting your user interface's responsiveness***.

IntentService isn't ***affected by most user interface lifecycle events***, so it continues to run in circumstances that would ***shut down an AsyncTask***.

https://developer.android.com/training/run-background-service/create-service

IntentService – Create

IntentService – Code

```java
public class ExampleIntentService extends IntentService {
    private static final String ACTION_EXAMPLE_START = "example.start";
    public static final String ACTION_EXAMPLE_END = "example.end";

    public static final String RESULT_PARAM = "result";

    public ExampleIntentService() {
        super( name: "ExampleIntentService");
    }

    public static void startExample(Context context) {
        Intent intent = new Intent(context, ExampleIntentService.class);
        intent.setAction(ACTION_EXAMPLE_START);
        context.startService(intent);
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        if (intent != null) {
            final String action = intent.getAction();
            if (ACTION_EXAMPLE_START.equals(action)) {
                handleActionExample();
            }
        }
    }

    private void handleActionExample() {
        try {
            Thread.sleep( millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Intent intent = new Intent();
        intent.setAction(ACTION_EXAMPLE_END);
        intent.putExtra(RESULT_PARAM, value: "Receive the message from other threads by IntentService!");
        LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
    }
```

ExampleIntentService.java

```java
ExampleIntentService.startExample( context: this);
```

MainActivity.java

IntentService – Result

THE UNIVERSITY OF MELBOURNE

Thank you