1: Android process
Follow following video tutorials to learn more about threads and process
https://www.youtube.com/playlist?list=PLonJJ3BVjZW6hmkEaYIvLLm5IEGM0kpwU

And pluralsight video (if you have account of it)
http://www.pluralsight.com/courses/android-processes-threads

Important links:
http://stackoverflow.com/questions/17482469/understanding-what-looper-is-about-in-android
http://codetheory.in/android-handlers-runnables-loopers-messagequeue-handlerthread/
http://stackoverflow.com/questions/14342773/meaning-of-pid-ppid-and-tgid

http://elinux.org/Using_smem_on_Android
http://unix.stackexchange.com/questions/18266/why-the-value-of-vsize-in-top-is-different-from-the-value-of-vsz-virtual-set-si
http://www.tutorialspoint.com/unix/unix-file-permission.htm

https://software.intel.com/en-us/android/articles/performance-debugging-of-android-applications
https://en.wikipedia.org/wiki/Jiffy_(time)
http://www.makelinux.net/books/lkd2/ch10lev1sec3
https://groups.google.com/forum/#!topic/android-developers/ZC_JPsryBUM

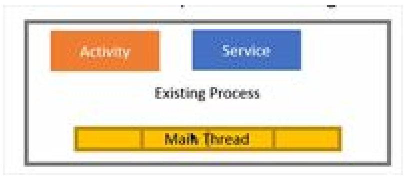
http://stackoverflow.com/questions/704311/android-how-do-i-investigate-an-anr

http://stackoverflow.com/questions/9737420/how-to-stop-or-destroy-a-running-thread
http://stackoverflow.com/questions/10089911/android-how-to-stop-the-running-thread-safely
http://stackoverflow.com/questions/15962646/android-thread-join-causes-application-to-hang
http://stackoverflow.com/questions/5382247/android-gameloop-thread-join-hangs-application

http://www.techrepublic.com/article/a-painless-way-to-troubleshoot-androids-dreaded-anr/
http://developer.android.com/training/articles/perf-anr.html
https://github.com/SalomonBrys/ANR-WatchDog
https://quequero.org/2012/01/debugging-deadlocks-on-android/

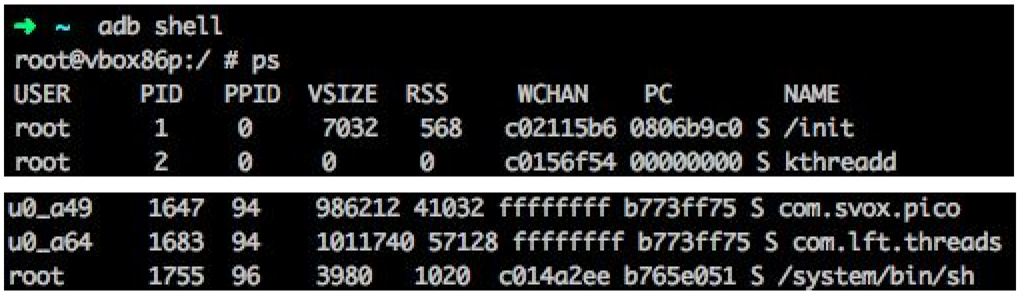
Process:
- An app is a player and a process is its playing ground
- Meaning in order for app to run and play, a playing ground is needed. It is known as process

What does a playing ground constitutes
- A main thread



Run an app

1. Go to terminal or command line
2. Type adb shell
3. type ps
   command to list all process



```
➜ ~  adb shell
root@vbox86p:/ # ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     7032   568     c02115b6 0806b9c0 S /init
root      2     0     0      0       c0156f54 00000000 S kthreadd
u0_a49    1647  94    986212 41032 ffffffff b773ff75 S com.svox.pico
u0_a64    1683  94    1011740 57128 ffffffff b773ff75 S com.lft.threads
root      1755  96    3980   1020   c014a2ee b765e051 S /system/bin/sh
```

user id	PID	PPID	VSIZE RSS	NAME
u0_a64	Process Id	Parent Process ID	Memory related values	package name (com.lft.threads)

This proves that our app is run in a process with PID 1683(as in example)

user id : user + app
u0 : user 0
a64: app id or name

Android being built on top of Linux Kernel, these are related to Linux process

user id == each app being run is given a user id,
user id == app id i.e. only that app is able to modify its file system

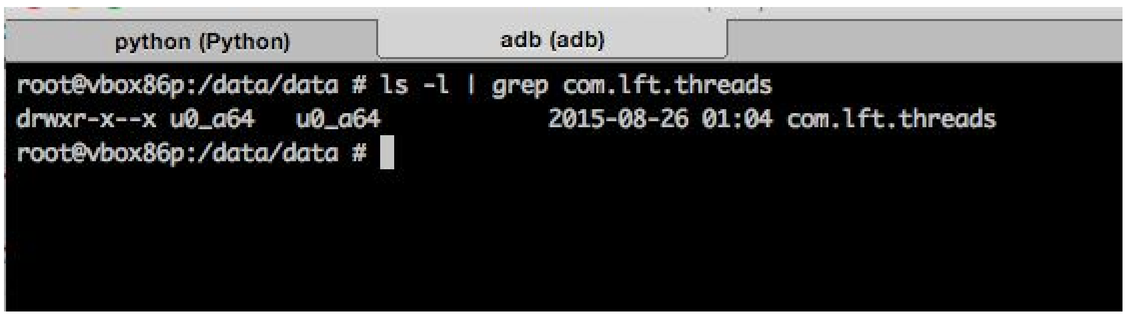
What is FileSystem
There are two types of memory ,
- runtime memory == similar to RAM
- file storage == similar to hard disk

FileSystem : storage or hard disk
So storage is given to the app and that app only has access to those storage . Though there is IPC (Inter Process Communication) which allows other process to access the app storage as well

User id : comes in play here, each app has user id or app id , that can only access the storage.

Action
- go to terminal or command line
- type adb shell (Use Emulator in this case as in normal phone (not rooted), the normal user doesn't have access to secure file location)
- cd /data/data (This is the secure location for all apps to store their data , though sdcard storage can also be used)
- type ls -l , it will list all files with permission (http://www.tutorialspoint.com/unix/unix-file-permission.htm)

d directory	rwx	r-x	—x
- file	user permission	group permission	world permission



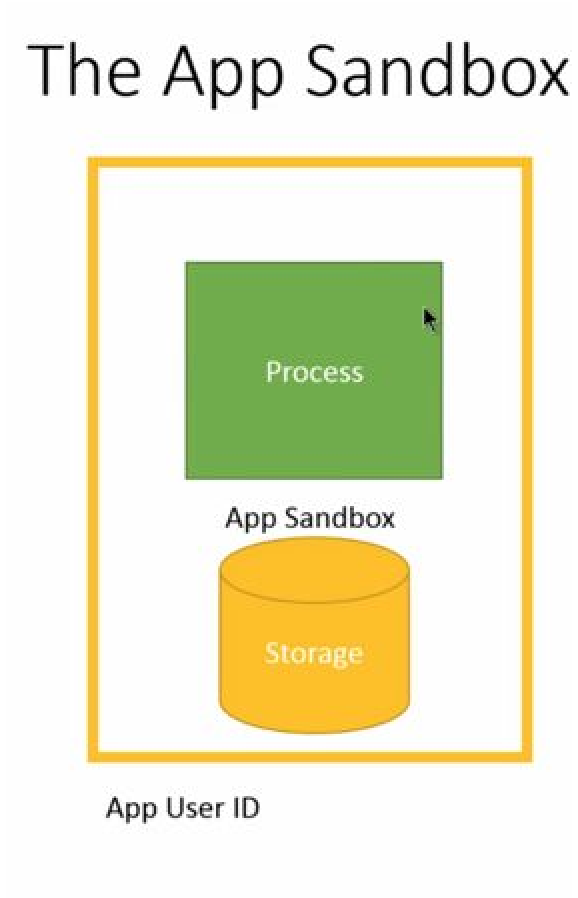
```
root@vbox86p:/data/data # ls -l | grep com.lft.threads
drwxr-x--x u0_a64   u0_a64            2015-08-26 01:04 com.lft.threads
root@vbox86p:/data/data #
```

com.lft.threads is a directory, so u0_a64 only has access to this folder location

This directory is the source for
shared preference
database
and other files

So a running app has a
- process
- storage

This is known as app sandbox

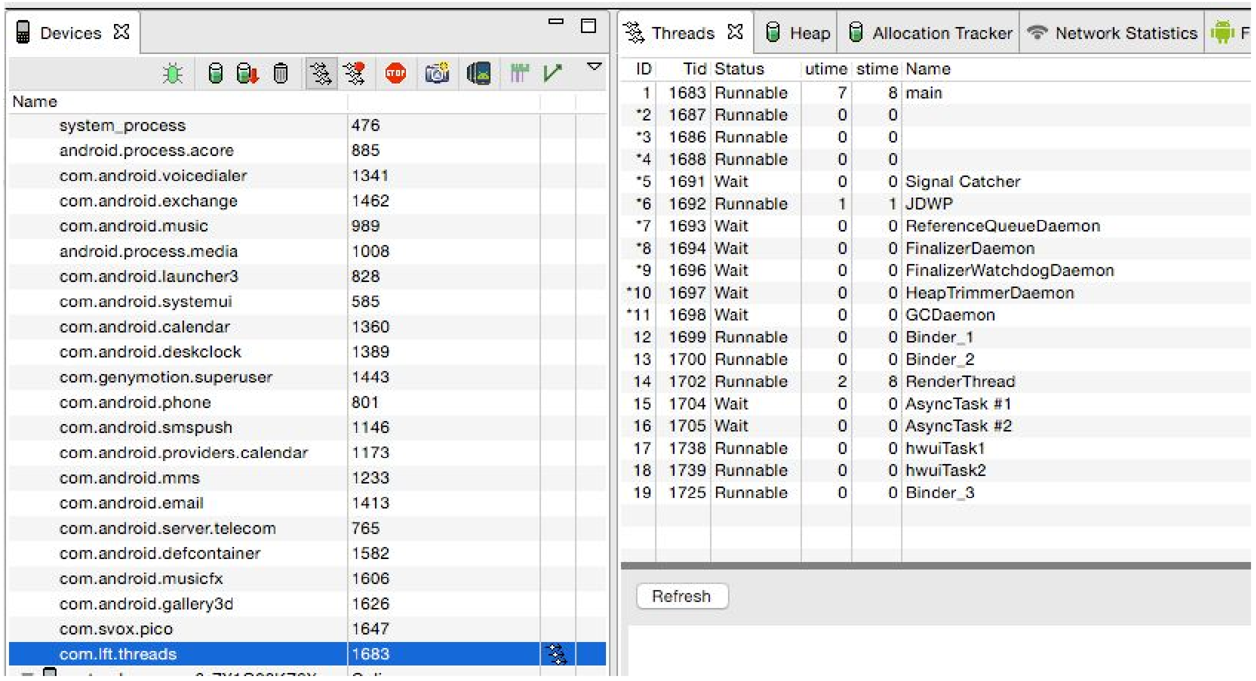


App sandbox contains
- unique user id
- unique process
- unique filesystem

Main Thread:
Each and every action that runs in an Android app is done by Main Thread (until and unless you aren’t using any background threads)
Main Thread responsible for
- activity, service, broadcast receiver, button and ui updates, each and every major and minor calculation

Action
- go to Android DDMS

- select your process (in our case it is com.lft.threads 1683)
- click on thread tab to the right of trash can
- view thread tabs



Name	
system_process	476
android.process.acore	885
com.android.voicedialer	1341
com.android.exchange	1462
com.android.music	989
android.process.media	1008
com.android.launcher3	828
com.android.systemui	585
com.android.calendar	1360
com.android.deskclock	1389
com.genymotion.superuser	1443
com.android.phone	801
com.android.smspush	1146
com.android.providers.calendar	1173
com.android.mms	1233
com.android.email	1413
com.android.server.telecom	765
com.android.defcontainer	1582
com.android.musicfx	1606
com.android.gallery3d	1626
com.svox.pico	1647
com.lft.threads	1683

ID	Tid	Status	utime	stime	Name
1	1683	Runnable	7	8	main
*2	1687	Runnable	0	0	
*3	1686	Runnable	0	0	
*4	1688	Runnable	0	0	
*5	1691	Wait	0	0	Signal Catcher
*6	1692	Runnable	1	1	JDWP
*7	1693	Wait	0	0	ReferenceQueueDaemon
*8	1694	Wait	0	0	FinalizerDaemon
*9	1696	Wait	0	0	FinalizerWatchdogDaemon
*10	1697	Wait	0	0	HeapTrimmerDaemon
*11	1698	Wait	0	0	GCDaemon
12	1699	Runnable	0	0	Binder_1
13	1700	Runnable	0	0	Binder_2
14	1702	Runnable	2	8	RenderThread
15	1704	Wait	0	0	AsyncTask #1
16	1705	Wait	0	0	AsyncTask #2
17	1738	Runnable	0	0	hwuiTask1
18	1739	Runnable	0	0	hwuiTask2
19	1725	Runnable	0	0	Binder_3

You can clearly see
ID   Tid (Thread ID)   utime and stime (relates to thread execution time in jiffies) and name (thread name)

so there is main thread

Look on the ID section, some contains * on it, which signifies that there are other worker threads in your process given by the system. These worker threads are responsible for doing out various action on process

like Garbage Collection (GCDaemon), Signal catching and so on. We don't work directly with these threads.
 The only concern we have here is main thread

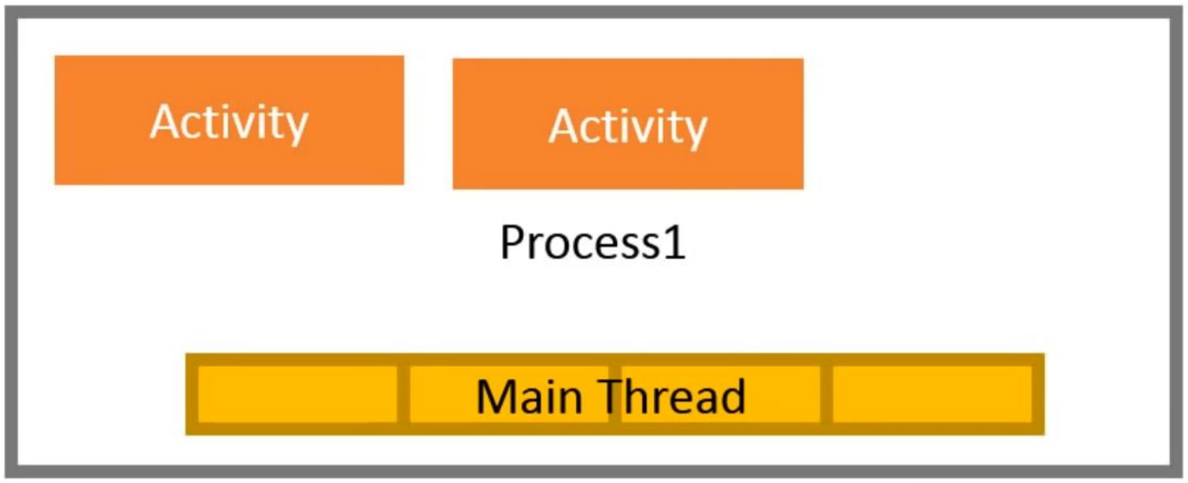
State of thread
http://stackoverflow.com/questions/704311/android-how-do-i-investigate-an-anr

State of thread (status column: not all are visible)

- running - executing application code
- sleeping - called Thread.sleep()
- monitor - waiting to acquire a monitor lock
- wait - in Object.wait()
- native - executing native code
- vmwait - waiting on a VM resource
- zombie - thread is in the process of dying
- init - thread is initializing (you shouldn't see this)
- starting - thread is about to start (you shouldn't see this either)

these status are shown in DDMS thread
so when thread is in
runnable : means it is running or can take actions to run
wait: means it is in waiting condition

So a process contains a main thread to do all app action and other threads as given by Android system to do works like Garbage Collection and threads defined by user.



What is Thread
Thread is a worker that does assigned work.
For instance, thread is a helper in your home. You assign task like clean house, cook meals, buy groceries etc to your helper and helper does the assigned task.
In app , it may be like Fetch Image from remote server, do long calculation and once you assign the task ,thread does it.

So thread must know what task need to be performed

```
public class FirstThreads extends Thread{
    @Override
    public void run() {
      super.run();
    }
}
```

In the above class, we create a custom thread and we haven't defined any task to run here. So when we do

```
FirstThreads firstthread  =new FirstThreads();
firstthread.start();
```

Here we only start a thread, which does nothing. But if we do the following

```
public class FirstThreads extends Thread {
    @Override
    public void run() {
      int count = 0;
      while (count < 10) {
        count++;
      }
    }
}
```

Our thread now counts up to 9 numbers . So we have given a task to the thread to count 9 numbers and now thread does that.

So task can be assigned to thread on its run() function

Runnable
A set of statement to be run.

```
public class FirstRunnable implements Runnable {

    @Override
    public void run() {
      int a = 2, b = 3;
```

```
        int sum = a + b;
        System.out.println("The sum of 2 & 3 is " + sum);
    }
}
```

Here , we are creating a runnable to add 2 and 3.

If we do
FirstRunnable firstRunnable =new FirstRunnable();
firstRunnable.run();

This code will be executed in our MainThread.

See, both Thread and Runnable has a function run() which contains a set of tasks to be run.

Let us modify our FirstThread now

```
public class FirstThreads extends Thread {
    public FirstThreads(Runnable runnable){
        super(runnable);
    }
  }
```

Now it takes Runnable as constructor argument

Now let us

```
  public class FirstRunnable implements Runnable {

    @Override
    public void run() {
        int count = 0;
        while (count < 10) {
            count++;
        }
    }
  }
```

This runnable is same as our FirstThreads as we defined earlier. So now if we do

```
FirstRunnable firstRunnable =new FirstRunnable();
FirstThreads firstThread =new FirstThreads(firstRunnable);
firstThread.start();
```

This above method will now count upto 9 numbers which is same as defining it on run() of thread.

So runnable is nothing but a set of statements. And runnable always require a thread to run

So now our thread, count upto 9 numbers. How do we know that the thread has completed counting upto 9 numbers? We need to acknowledge that thread has completed doing some action

Now comes Handler

Handler is merely a messenger. It is simply a postman, capable of passing messages as well as constructing messages.

So our postman, now has to say that counting upto 9 numbers is done.

So let us define our postman or Handler

```
public static final int COUNT_COMPLETE=1;

  public class FirstHandler extends Handler {
      @Override
      public void handleMessage(Message msg) {
          super.handleMessage(msg);
          if (msg.what == COUNT_COMPLETE)
              System.out.println("Count is complete");
      }
   }
```

and in our FirstRunnable or in FirstThreads (if you aren’t using runnable to assign as task)

```
  public class FirstRunnable implements Runnable {

      @Override
```

```
        public void run() {
            int count = 0;
            while (count < 10) {
                count++;
                System.out.println(count);
            }
            firstHandler.sendEmptyMessage(COUNT_COMPLETE);
        }
    }
```

So what this piece of code does is that once the count upto 9 is done, it will post empty message event with COUNT_COMPLETE.
This event will be listened by the Handler's , handleMessage. And we know that counting is done.

One important point is that you must declare this FirstHandler on main UI() as it is to be associated with the thread. As it is associated with mainUI() thread, whenever any background thread does the work, the mainUI() will be notified through Handler's handleMessage().

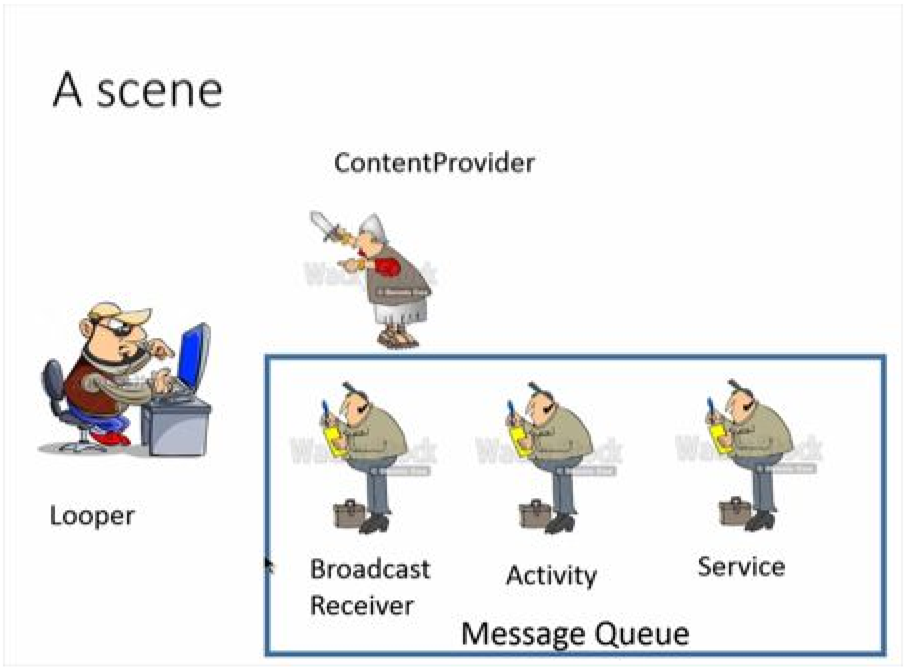
```
onCreate(){
  firstHandler =new FirstHandler();
}
```

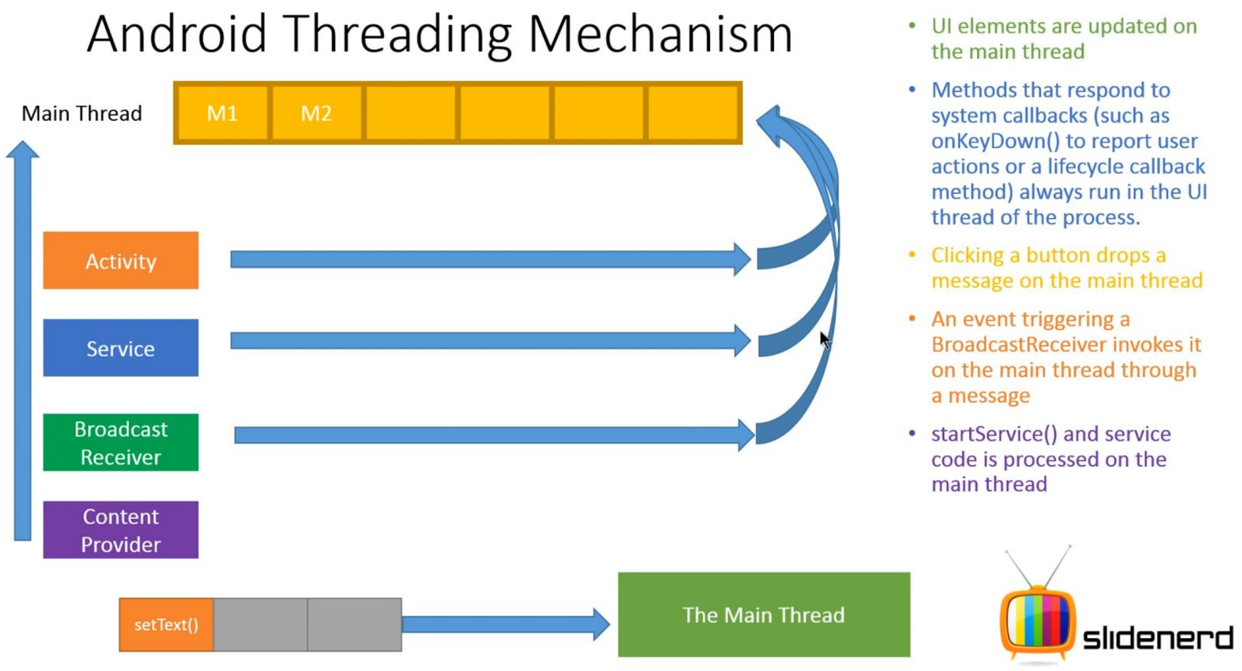
From this mechanism , what we understood is that Handler() must be associated with a thread to acknowledge the message.

In normal postman work (working at post office), he/she has a message stack i.e. number of messages(envelopes) that need to be delivered. Similarly our handler, may also have number of message and need to know what to do with that message.
Here message may be message as shown above or runnables.
So, to manage those message queue, we have Loopers
Loopers mainly job is to loop around messagequeue and pass those message queue to appropriate handler associated with the thread.

Let us take this analogy
There are 3 messages to be sent
Message to Mr A.
Message to Mr B.
Message to Mr C.

Looper loops through this message and send to our postman i.e. handler
Handler now performs the message task using thread.
Though this analogy may not be appropriate, Handler (our postman) need energy to deliver message or perform task, that energy is our thread.




Android Threading Mechanism

- UI elements are updated on the main thread
- Methods that respond to system callbacks (such as onKeyDown() to report user actions or a lifecycle callback method) always run in the UI thread of the process.
- Clicking a button drops a message on the main thread
- An event triggering a BroadcastReceiver invokes it on the main thread through a message
- startService() and service code is processed on the main thread

Now let us create a custom thread having its handler ,looper

Let say we have two runnables, these runnables are analogous to two messages we need to send

```
public class ThirdRunnable implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 30; i++) {
            System.out.println("4x" + i + "=" + 4 * i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}


public class FourthRunnable implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 30; i++) {
            System.out.println("5x" + i + "=" + 5 * i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

One runnable prints multiplication table of 4 and other 5. Say message 4 has to be delivered to Mr 4 and message 5 needs to be delivered to Mr 5.

We need energy to do it , which becomes our thread

```
public class CustomThread extends Thread {
   public Handler handler;

   public CustomThread(String name) {
      super(name);
   }

   public void execute(Runnable runnable) {
      handler.post(runnable);
   }

   public CustomThread(Runnable runnable) {
      super(runnable);
   }

   @Override
   public void run() {
      try {
         // preparing a looper on current thread
         // the current thread is being detected implicitly
         Looper.prepare();

         //Log.i(TAG, "DownloadThread entering the loop");

         // now, the handler will automatically bind to the
         // Looper that is attached to the current thread
         // You don't need to specify the Looper explicitly
         handler = new Handler();

         // After the following line the thread will start
         // running the message loop and will not normally
         // exit the loop unless a problem happens or you
         // quit() the looper (see below)
         Looper.loop();

      } catch (Throwable t) {
      }
   }
}
```

Here our CustomThread has its own handler (i.e. postman). To make our CustomThread associated with the handler, it needs to be defined on run() of CustomThread(). And on run(), to be associated with Looper we need to do as in above.

This is the general mechanism of mainUI thread as well. Each and every handler we define on Activity,fragment will automatically gets associated with maniUIThread.

***** Handler,runnables and Looper are not threads, but a mechanism to assign tasks or actions to threads. All action will be done by Thread alone not by Handler,runnables or Looper.

So now when we do the following

```
ThirdRunnable thirdRunnable = new ThirdRunnable();
FourthRunnable fourthRunable =new FourthRunnable();
CustomThread customThread =new CustomThread();
customThread.start();
customThread.execute(thirdRunnable);
customThread.execute(fourthRunnable);
```

Our custom thread will first execute thirdRunnable and once thirdRunnable is complete , it will execute fourthRunnable.

Remember whenever you are doing new Handler().post(someRunnable) on Activity,Service,Fragment, you are simply adding message queue to mainUI(). For instance your runnable is taking more than 100 ms i.e. more than 500 ms, then in the same time there is action to perform some ui update as well. Both the tasks are stacked on message queue. Since the first runnable is taking more than 500 ms ,the second action (UI update) has to wait for more than 500 ms. This is the classic case of ANR (Application Not Responding) i.e. main UI is busy on doing some tasks for longer period of time and as a result , it is not being able to take call from other runnables.

So the same mechanism is going on with MainUI thread as well. All handlers you all have been calling on activity,fragment or any building block is stacked up as message queue and through mainUI looper it is being executed.