Wiki para las memorias de prácticas

La *wiki* colaborativa tiene que servir para aprender a trabajar en equipos heterogéneos y con gente que tenga diversas actitudes y aptitudes, lo cual es muy importante en ingeniería hoy en día. Por tanto:

- Es muy importante ser consciente en todo momento de que la nota completa de la *wiki* **influye en la nota de todos y cada uno de los alumnos**. Si algún alumno decide no colaborar, debería ser honesto y pasarse cuanto antes al modo de evaluación no-continua, permitiendo así que el resto se reparta de nuevo el trabajo para obtener la mejor calificación posible.
- El trabajo de hacer las memorias para la *wiki* no consiste en un simple reparto de prácticas, aislándose los grupos entre sí: los alumnos **pueden ayudar a otros a mejorar el trabajo**. Esto es importante porque la *wiki* es un recurso que todos tendréis disponible en el examen, así que debería estar lo mejor hecho posible desde el punto de vista de cada uno de vosotros.
- Aunque un alumno participe en subir a la *wiki* una práctica, no es esa práctica la única que debe realizar: todas las prácticas deben ser resueltas por todos los alumnos, aunque el trabajo de elaborar memorias y subirlas a la *wiki* sea colaborativo. **No confundáis el trabajo de elaborar memorias con el de resolver las prácticas y ejercicios.**

En cuanto a los aspectos técnicos:

- En la página inicial debe haber un índice con una entrada por cada práctica o relación de problemas, y cada entrada debe ir a una página propia para ella. **No uséis una estructura distinta de ésa.**
- En la página de cada práctica o relación de problemas debéis indicar quiénes se han encargado de subirla a la *wiki*.
- Las memorias no pueden contener únicamente el código de los programas realizados, o sólo los resultados finales: deben incluir los pasos, explicaciones y comentarios oportunos para resolver cada práctica o relación de problemas.
- **No se pueden subir a la wiki partes relevantes de apuntes de clase o teoría de libros, etc.; únicamente las explicaciones necesarias de cómo se ha resuelto cada práctica o relación de problemas, trozos de código si hace falta, figuras, ecuaciones usadas, etc.**

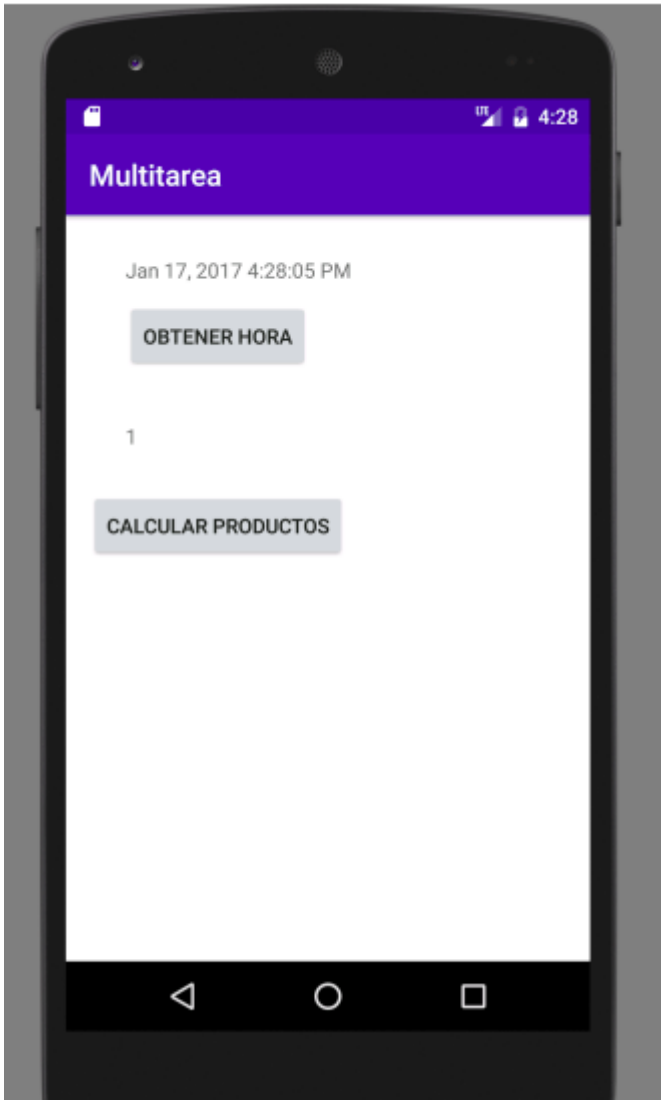
Práctica de multitarea en JAVA para dispositivos móviles

Practica número 6.

Eduardo Pérez González y Verónica Tapia García.

La práctica de multitarea para dispositivos móviles se encuentra dividida en dos partes. Para la primera parte se pide implementar una aplicación con un interfaz con dos parejas de *TextView* y botón. El funcionamiento es el siguiente:
Al pulsar el primer botón, se tomará la hora del sistema y se mostrará por el primer *TextView.* Esto se ha de realizar en el *thread* principal.
Al pulsar el segundo botón se lanzará otro *thread* (un *thread worker* secundario) que calcule el producto de los números comprendidos entre 0 y 5^{e09}. El *thread worker*, hará uso de *handlers* para enviar un 0 en caso de que el índice del bucle del cálculo sea par y un 1 en caso de que sea impar. El UI *thread (thread* principal) imprimirá en el segundo *TextView* el número recibido por el *thread worker*.

En base a cumplir las especificaciones el interfaz de la aplicación queda de la siguiente forma:



Implementación de la funcionalidad:

Para implementar la funcionalidad del primer botón, se le asocia una función llamada *showTime* la cual obtiene la hora del sistema y la muestra por el primer *TextView.*

```
<Button
    android:text="Obtener hora"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/text_message1"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_marginLeft="25dp"
    android:layout_marginStart="25dp"
    android:layout_marginTop="12dp"
    android:id="@+id/button"
    android:onClick="showTime"/>
```

```
public void showTime(View view) {
    TextView tv = (TextView)findViewById(R.id.text_message1);
    String currentDateTimeString = DateFormat.getDateTimeInstance().format(new Date());
    tv.setText(currentDateTimeString);
}
```

Esta función está implementada dentro de la *mainactivity* y es ejecutada por el *UIThread.*

Para implementar la funcionalidad del segundo botón, la dificultad está en hacer uso de la multitarea para ejecutar la función asociada al botón en un *thread worker* y hacer uso de un *handler* para la comunicación entre el UI *thread* y el *thread* worker. Los *threads* comparten el mismo espacio de memoria y pueden generar problemas. Para evitar conflictos solamente el *UI thread* puede escribir en el *TextView.* Los índices de los productos estarán en el *thread worker* y han de pasarse al *thread* principal para que este los muestre por el *TextView*. Esto se consigue mediante el uso de *handlers*.

```
<Button
    android:text="Calcular productos"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/text_message2"
    android:layout_alignParentLeft="true"
    android:layout_alignParentStart="true"
    android:layout_marginTop="28dp"
    android:id="@+id/button2"
    android:onClick="startThinking"/>
```

Se asocia el segundo botón con la función *startThinking.*
En esta segunda función, como se ha comentado previamente, se tendrá que hacer uso de un *handler* para comunicar ambos *thread.* En consecuencia se procede en primer lugar a crear el *handler.* Hay dos formas de hacerlo, en este caso se ha optado por crear una clase a que extiende la clase *Handler* de Android (como se ha visto en las transparencias) e instanciar un *Handler* llamado puente a partir de esa clase a.

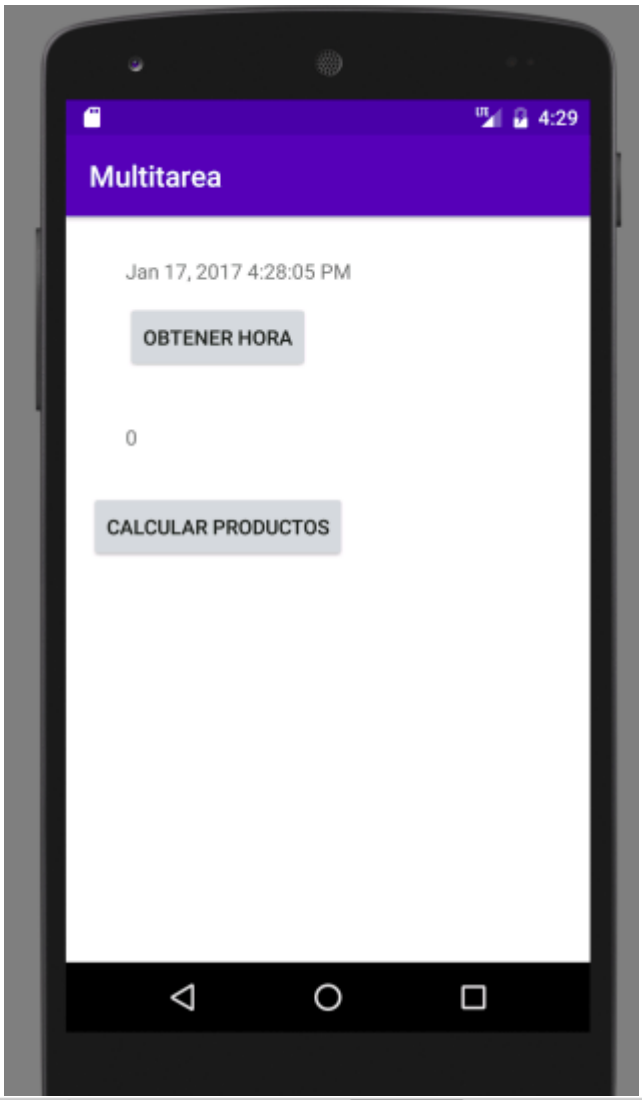
```
/* En primer lugar hay que crear un handler, que o bien puede instan
 * ciarse como una clase nueva que extiende handler y luego instanciarlo
 * o bien instanciar el metodo directamente como private*/

class a extends Handler{

}

private Handler puente = new a();
```

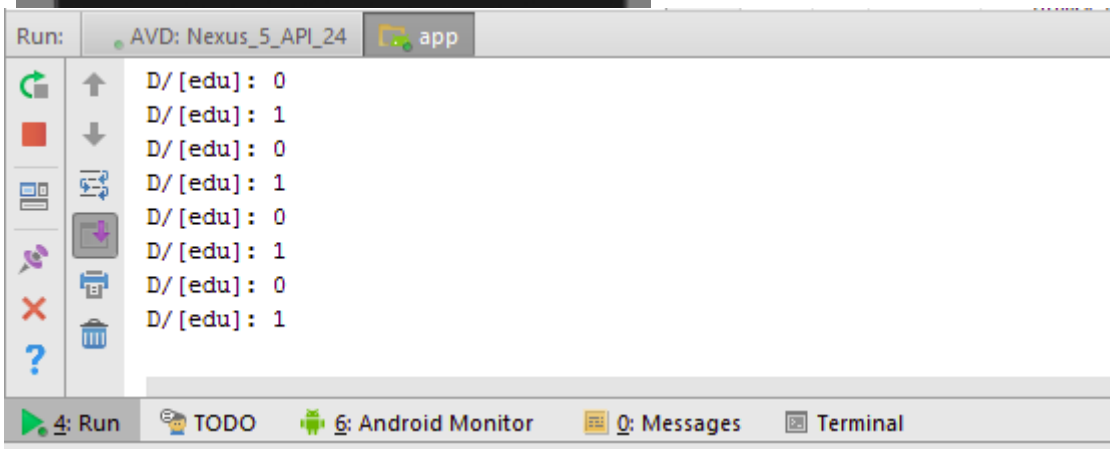
Seguidamente se procede a implementar la función *startThinking*.

```
public void startThinking(View view) {

    new Thread((Runnable) () → {

    }).start();

}
```

En esta función, lo primero es implementar un nuevo *thread* (el *thread worker*) para que las tareas de esta función se ejecuten ahí en lugar de en el UI *thread* y arrancarlo.
Seguidamente se implementa dentro del *thread* la funcionalidad deseada, en este caso calcular el cuadrado de los números comprendidos entre 0 y 5e9.

```
for(index=0; index<=500000000.0; index++){
    number = number*number;
    number++;
}
```





Para la comunicación con el *UI thread,* se crea un mensaje *msg* de tipo *Message*:

```
Message msg = new Message();
msg.obj=(index%2==0) ? 0: 1;
```

Dentro de este mensaje, se le asigna dentro de la propiedad obj, un 0 o un 1 en función de si el índice es par o impar.

Para concluir se pasa a enviar el mensaje haciendo uso del *handler:*

```
puente.sendMessage(msg);
```

El código de la función queda entonces de la siguiente manera:

```
public void startThinking(View view) {

    new Thread((Runnable) () → {
            double number = 1.0;
            double index = 0.0;
            for(index=0; index<=500000000.0; index++){
                number = number*number;
                number++;

                Message msg = new Message();
                msg.obj=(index%2==0) ? 0: 1;
                Log.d("[edu]",msg.obj.toString());
                puente.sendMessage(msg);
            }
            Log.d("[edu]","se acabo el for");
    }).start();
```

Ahora el *handler*, dentro del *UI thread* recoge el mensaje, con lo que el código del *handler* (que se había definido al principio) queda de la siguiente forma:

```
class a extends Handler{
    public void handleMessage(Message msg) {
        TextView tv = (TextView)findViewById(R.id.text_message2);
        tv.setText(msg.obj.toString());
    }
}
```

En él, cuando el mensaje msg es
enviado a través del *handler* por el *thread worker* se ejecuta la función *handleMessage*, en la que se recoge el mensaje (que era un 0 o un 1) y se muestra por el *TextView*.
Al ejecutarse en dos *threads* separados las dos
funcionalidades es posible usar los dos botones en paralelo y comprobar el
funcionamiento de la multitarea.

La segunda parte de la práctica,
consiste en hacer que el segundo *TextView* muestre el tiempo que se ha requerido por parte del *thread worker* para realizar la tarea. Adicionalmente en lugar de ir
enviando un 0 o un 1, cuando finalice pondrá el valor del tiempo empleado en
una variable sincronizada y enviará mediante el *handler* una F al UI *thread* que imprimirá por pantalla el tiempo de ejecución del bucle.

Para implementarlo nos basamos en
el código anterior y se realizan las modificaciones necesarias.
En primer lugar se define una
variable global que llamaremos *estimatedTime* que se utilizará para almacenar los tiempos de ejecución del bucle.

```
private long estimatedTime;
```

Seguidamente, dentro de la
función *startThinking* que se
ejecutará al pulsar el segundo botón, se procede a calcular el tiempo necesario
para ejecutar el bucle.
Para ello se adquiere el tiempo inicial (*startTime*) mediante la función *System.nanoTime(),* se ejecuta el bucle y se asigna a la variable global *estimatedTime* el valor en ese instante menos el valor inicial.
Una vez hecho esto, se crea un
mensaje *msg* como en el caso anterior
pero esta vez se le asigna el valor F antes de enviarse por el *handler.* El código queda de la siguiente
manera:

```
public void startThinking(View view) {

    new Thread((Runnable) () → {
            double number = 1.0;
            double index = 0.0;
            long startTime = System.nanoTime();

            for(index=0; index<=500000000.0; index++){
                number = number*number;
                number++;

            }

            estimatedTime = System.nanoTime() - startTime;
            Message msg = new Message();
            msg.obj="F";
            puente.sendMessage(msg);
    }).start();
```

Ya sólo queda recibir la
información en el *UI thread* y
mostrarla en el *TextView.*
Se recibe el mensaje msg, se pasa
de nanosegundos a segundos y se muestra por pantalla

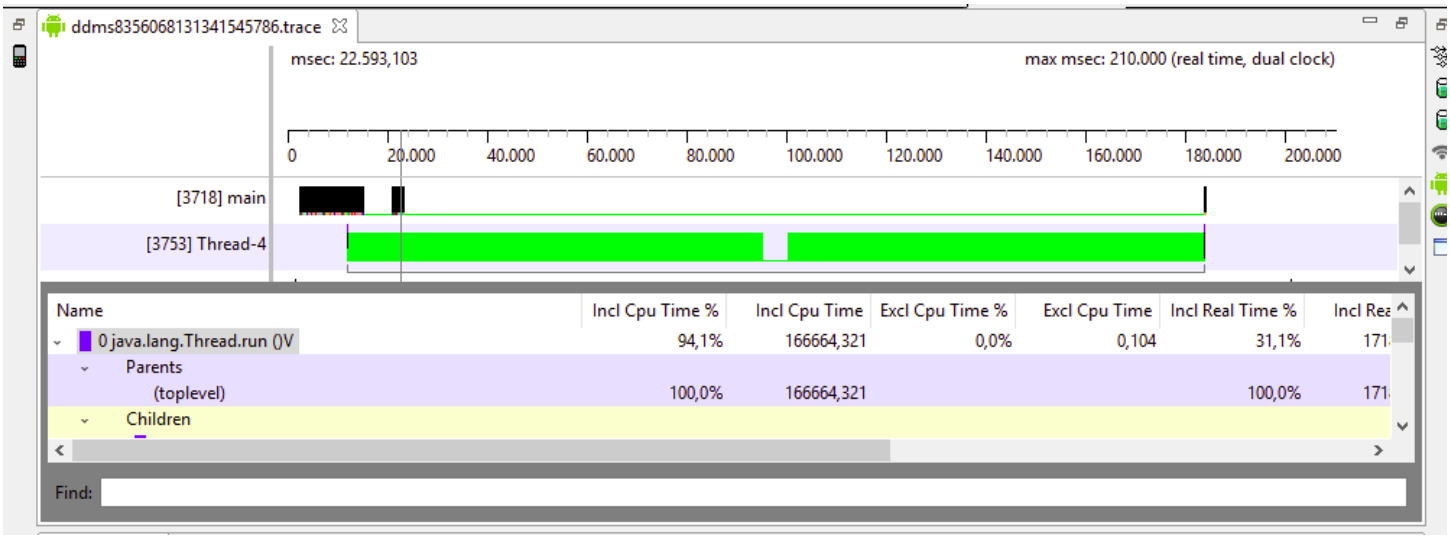
```
class a extends Handler{
    public void handleMessage(Message msg) {
        TextView tv = (TextView)findViewById(R.id.text_message2);
        double estimatedTimeseg= (double)estimatedTime / 1000000000.0;

        String cadena="El tiempo transcurrido es de : "+String.valueOf(estimatedTimeseg)+ " segundos";
        tv.setText(cadena);
    }
}
```

Una vez ejecutado los resultados
son como siguen:

Donde se puede ver que el tiempo
de ejecución necesario es de 171.81279304 segundos.

Para concluir, habría
que comparar el tiempo de cálculo
anteriormente obtenido, con el que se obtendría mediante *Android Device Manager*. Se procede por tanto a arrancar el *Android Device Manager* antes de ejecutar
la aplicación. Por simpleza de los cálculos se ha reducido el tamaño del bucle
de forma que no se tarde tanto en obtener los resultados.

El tiempo estimado para la
ejecución del bucle es de 171.81279304 segundos si nos fijamos en el tiempo
dado por la aplicación.
Si abrimos *Android Device Manager*, seleccionamos "*start method profiling"* y ejecutamos la aplicación los resultados obtenidos son los siguientes:



La interacción con la aplicación que ha dado lugar a esta traza es la que sigue:

ancar la aplicación

sar el primer botón para obtener la hora en dos ocasiones

sar el segundo botón para inicializar el cálculo y medir el tiempo de ejecución

ntras el thread worker está ejecutando el bucle, pulsar el primer botón para obtener la hora y ver que la multitarea funciona correctamente

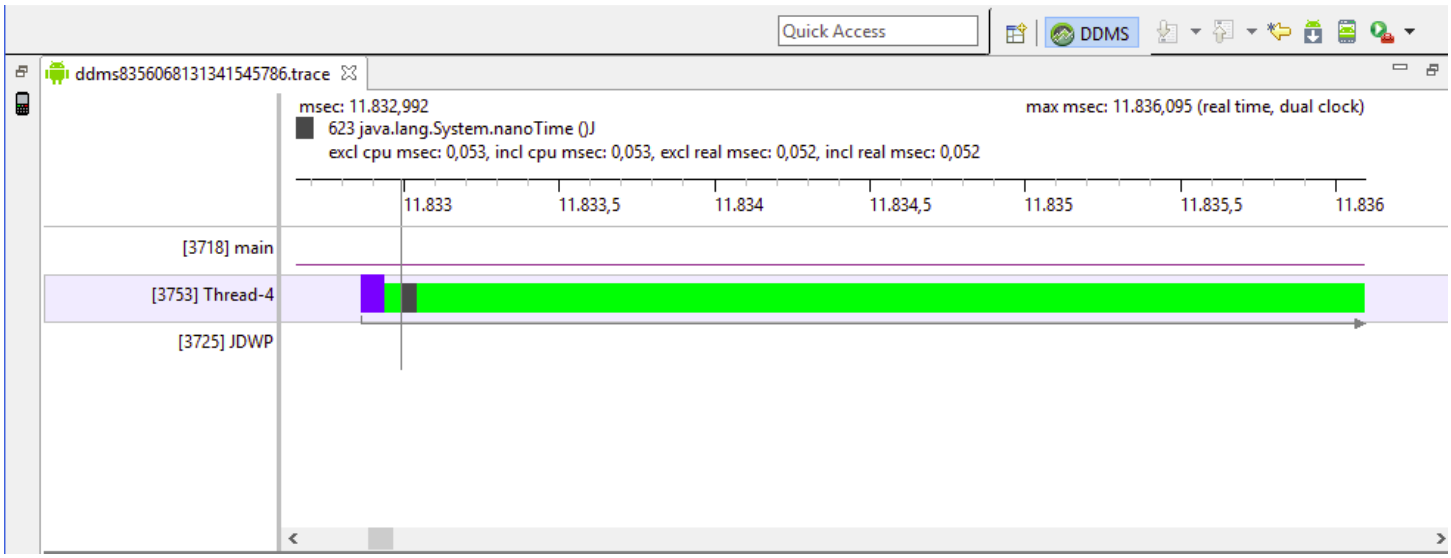
ba la ejecución y se para el method profiling

En consecuencia de lo anteriormente mencionado, la traza (que muestra el *UI thread* como *main* y el *thread worker* como *Thread 4*) muestra actividad en ambos thread, a veces a la vez en ambos (cuando se pide la hora mientras se ejecuta el bucle).

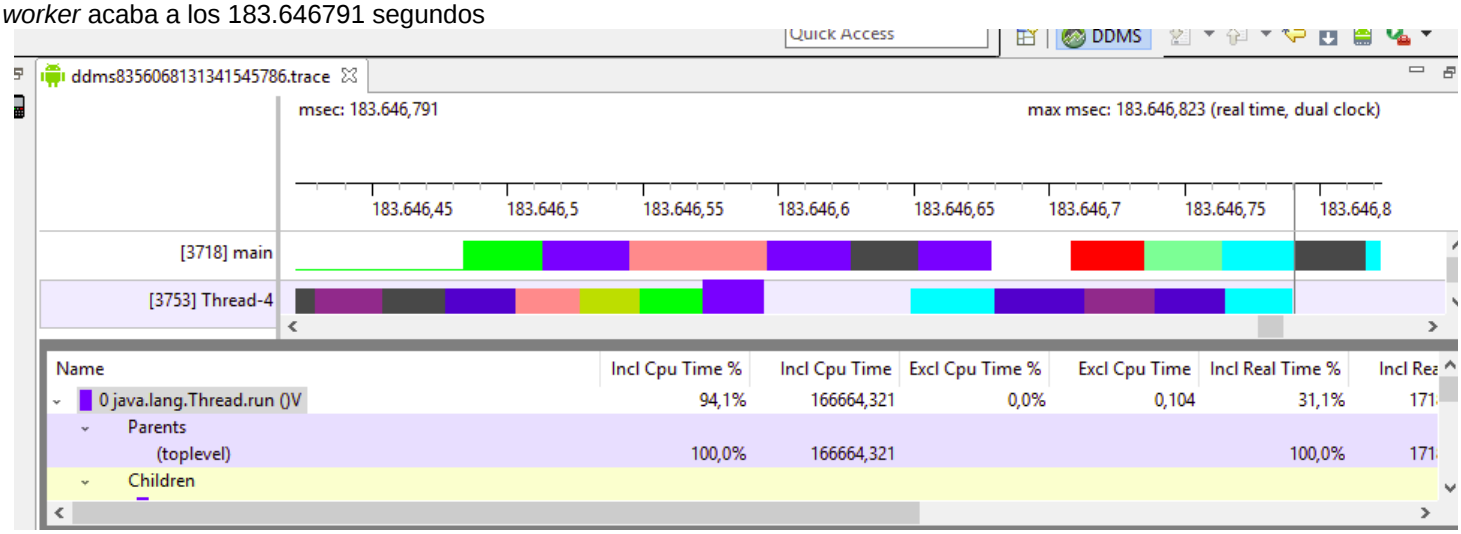
Los dos *threads* están ubicados en posiciones diferentes de memoria dentro del ordenador en el que se ha simulado el dispositivo virtual por lo que ambos *threads* se pueden ejecutar a la vez (y no de forma alterna intercalando la ejecución de ambos *threads* para dar la sensación de multitarea).

Para realizar una medida del tiempo que ha tardado en ejecutarse el bucle, se procede a hacer zoom sobre la zona de interes (en la que se empieza a ejecutar el bucle sobre el segundo *thread*).

El comienzo de la actividad del segundo *thread* comienza a los 11.832863 segundos

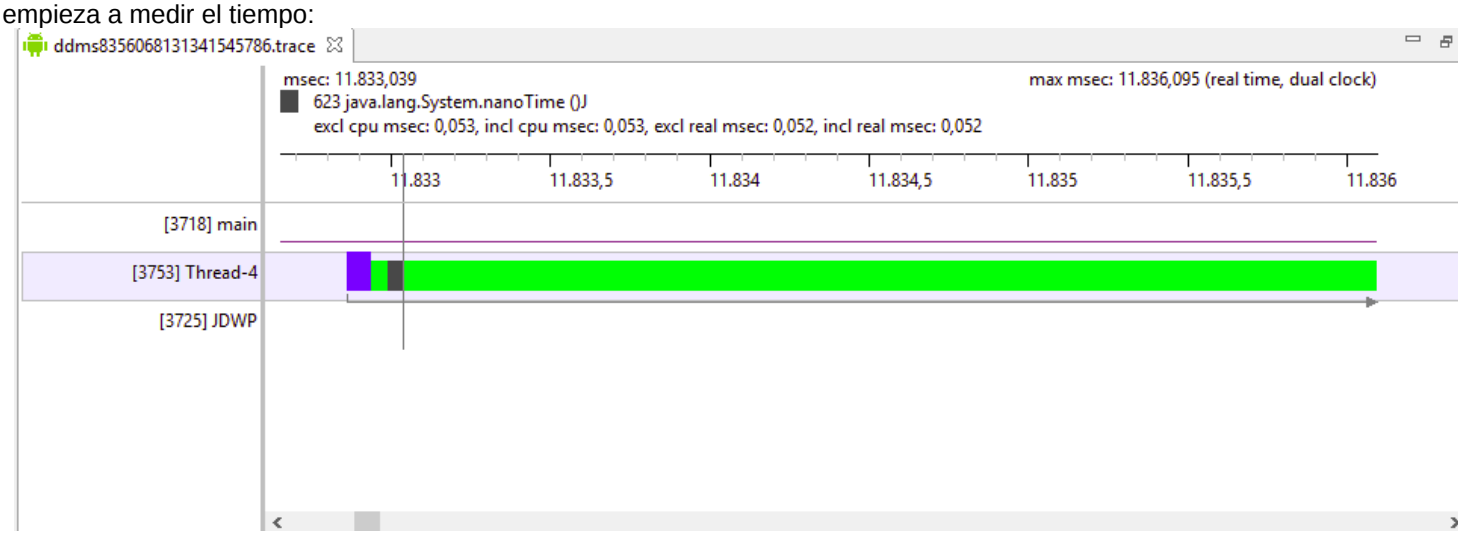


Y la actividad del *thread worker* acaba a los 183.646791 segundos

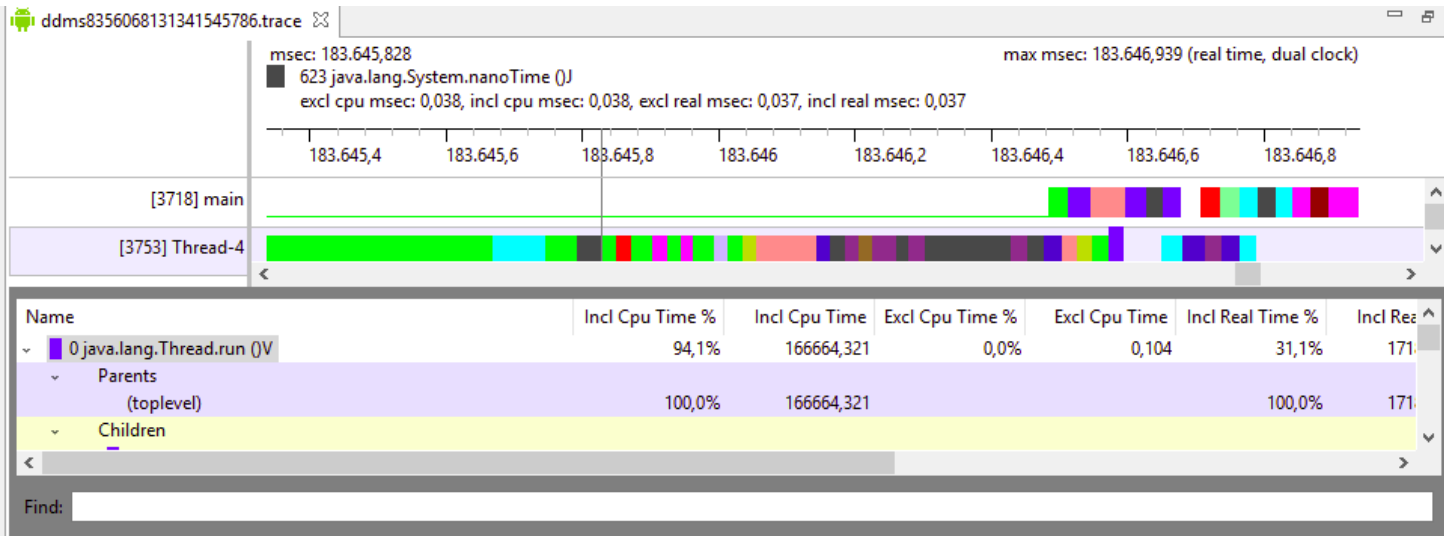


Esto muestra que, tomando el tiempo de ejecución del *thread worker* (que es el *thread* que ejecuta el bucle) como el tiempo de ejecución, el tiempo total es de 171.8139 segundos.

El tiempo mostrado por el móvil es de 171.81279304 segundos, lo cual da una diferencia entre ambos de 1.1 ms. Para justificar esta diferencia entre ambos resultados, en primer lugar habría que tener en cuenta la precisión obtenida por el marcador (mover un pixel el marcador a izquierda o derecha provoca una diferencia de microsegundos). En mayor medida y como principal motivo se procede a hacer zoom sobre la traza del *thread worker* y buscar la ejecución de la función *System.nanotime()* que es donde se empieza a medir el tiempo:



Se puede apreciar que la
ejecución de la función que toma la marca temporal se produce ligeramente
después de que comience la ejecución del *thread*
worker.



Análogamente, tomando la
ejecución del segundo *System.nanotime()* como punto final de
medida (y despreciando el tiempo de ejecución de tareas posteriores) se
obtendría que el tiempo total es de 171.812789 segundos. La diferencia con el
tiempo dado por la aplicación ahora es de tan
sólo 4 microsegundos, con lo que se puede decir que el tiempo de
ejecución es prácticamente similar.