

Модуль 3. Основы программирования Android приложений

Тема 3.4. Параллелизм и синхронизация. Потоки

2 часа

Оглавление

3.4. Параллелизм и синхронизация. Потоки.....	2
3.4.1. Общие понятия	2
3.4.2. Альтернативные способы создания потокового класса	5
3.4.3. Реализация логики потока.....	7
3.4.4. Синхронизация потоков.....	8
3.4.5. Блокировки	12
Упражнение 3.4.1	12
Задание 3.4.1.....	12
Благодарности	12

3.4. Параллелизм и синхронизация. Потоки

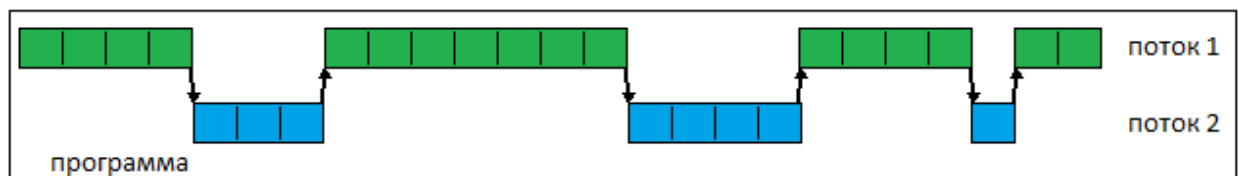
3.4.1. Общие понятия

С постоянным повышением сложности задач повышается и требование к вычислительным способностям (мощностям) компьютеров. Компьютерные технологии постоянно совершенствуются, и одним из требований к производству вычислительных систем является соблюдение закона Гордона Мура (основателя корпорации Intel), в соответствии с которым «производительность вычислительных систем должна удваиваться каждые 18 месяцев». Это способствовало появлению многоядерных процессоров. Если когда-то многоядерность была не распространена среди персональных компьютеров, а использовалась для специализированных задач, то на сей день даже мобильные устройства содержат в себе многоядерные процессоры (2-х, 4-х ядерные), а компьютерные процессоры насчитывают десятки ядер.

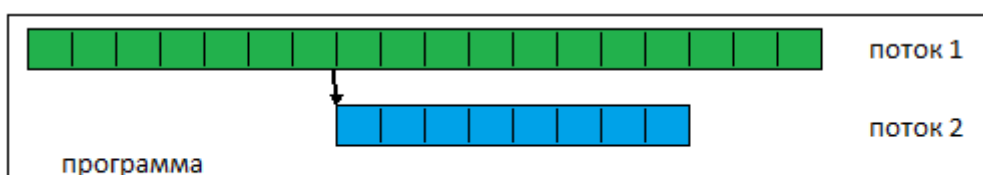
С приходом многоядерности появилось понятие параллелизма – одновременного выполнения нескольких вычислений. До этого все задачи процессор выполнял последовательно. Конечно и при одноядерных процессорах можно было просматривать сайты в браузере, слушать музыкальный плеер и в это время копировать файлы на флешку, и для пользователя всё это выглядело как одновременное (параллельное) выполнение, но на самом деле над этими процессами работал всего один процессор. Для того, чтобы создать видимость параллельного выполнения, процессы (программы) разделялись на **потоки**. Процессор поочередно выполнял потоки из разных процессов, таким образом создавалось впечатление параллелизма, это называется – псевдопараллелизмом.

Давайте разбираться дальше, что же такое поток?

Поток можно представить как последовательность команд программы, которая претендует на использование процессора вычислительной системы для своего выполнения. Потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют (совместно используют) данные программы.



в каждый момент времени выполняется один поток (псевдопараллелизм)



параллельное выполнение потоков

Разработка многопоточных программ непростой процесс. При планировании многопоточной программы оптимальным вариантом является разделение задачи на потоки, число которых равно количеству процессоров. Если сделать много потоков, то это не ускорит выполнение программы, а затормозит её, потому что переключение между потоками требует некоторого дополнительного времени.

Дополнительно разработка усложняется рисками нарушения целостности данных. Зачастую потоки нуждаются в одних и тех же ресурсах, при параллельном изменении этих ресурсов могут возникнуть ошибки. Например, два потока должны записать данные в файл, если они попытаются сделать это одновременно, то в итоге получим совершенно непредсказуемый результат. Чтобы этого не произошло потоки нужно **синхронизировать**. Пока над записью в файл работает первый поток, второй поток должен ожидать своей очереди.

Одним из средств организации параллельных потоков в Android является класс **AsyncTask**. Его задача состоит в том, чтобы выполнять задачи в фоновом потоке, параллельно с UI. Этот вариант идеально подходит для легковесных задач. Также данный класс имеет метод для обновления прогресса выполнения задачи в потоке UI.

Рассмотрим некоторые методы класса AsyncTask

Метод (public)	Описание
execute()	Выполняет задачу
Метод (protected)	Описание
doInBackground ()	В этом обязательном методе описываются команды, которые нужно выполнить в фоновом потоке
onPreExecute ()	Метод запускается перед методом doInBackground()
onPostExecute ()	Метод запускается после метода doInBackground()

Ниже приведен небольшой пример с применением AsyncTask. В примере используется задержка времени в качестве имитации, к примеру, загрузки файла из интернета. На layout располагается ProgressBar и TextView для вывода состояния задачи. Запуск задачи осуществляется нажатием на кнопку Button.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/button"
        android:text="Старт"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <ProgressBar
        style="?android:attr/progressBarStyleHorizontal"
```

```
        android:layout_width="fill_parent"
        android:layout_height="20pt"
        android:id="@+id/progressBar" />
    <TextView
        android:id="@+id/text"
        android:text="Выполнено : 0 / 100"
        android:layout_width="fill_parent"
        android:layout_height="20pt" />
</LinearLayout>
```

В классе MainActivity нужно поместить следующий код

```
public class MainActivity extends Activity {

    Button bStart;
    ProgressBar progressBar;
    TextView text;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bStart = (Button) findViewById(R.id.button);
        progressBar = (ProgressBar) findViewById(R.id.progressBar);
        text = (TextView) findViewById(R.id.text);
        bStart.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                new LoadImage().execute();
            }
        });
    }

    private class LoadImage extends AsyncTask<Void, Integer, Void> {
        @Override
        protected void onPreExecute() {
            super.onPreExecute();
        }

        protected Void doInBackground(Void... args) {
            for (int i = 0; i < 100; i += 8) {
                try {
                    Thread.sleep(200);
                    publishProgress(i);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            publishProgress(100);
            return null;
        }

        protected void onPostExecute(Void image) {
            text.setText("Задача завершена");
        }
    }
}
```

```
@Override
protected void onProgressUpdate(Integer... values) {
    super.onProgressUpdate(values);
    progressBar.setProgress(values[0]);
    text.setText("Выполнено : " + values[0] + "/100");
}
}
```

Метод `onProgressUpdate` запускается в UI потоке после вызова метода `publishProgress()`. С помощью него можно обновлять данные о ходе выполнении задачи. К примеру, можно выводить их для пользователя на `ProgressBar`, как показано в примере.

Метод объекта `AsyncTask` может быть вызван только внутри UI потока. Если создать еще один объект задачи и выполнить для неё метод `execute` во время выполнения первой будут работать две задачи, такого допускать нельзя.



При повороте экрана в приложении во время выполнения задачи `AsyncTask` будет создана новая задача, которая начнет выполняться заново, но старая задача, также продолжит свое выполнение. Это произойдет из-за того, что Android при повороте экрана создает новое `Activity` и метод `onCreate` будет вызван заново.

3.4.2. Альтернативные способы создания потокового класса

Класс `AsyncTask` легок в использовании и служит для выполнения несложных операций. Для более сложных процессов в Java есть класс `Thread`, а также интерфейс `Runnable`. Предлагается два варианта создания параллельной программы. Можно создать объект, наследуемый от класса `Thread`, переопределив в нем метод `run()`, также можно создать объекты реализующие интерфейс `Runnable`.



Ознакомиться со всеми конструкторами и методами класса `Thread` можно на официальном сайте

<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>

Ниже рассмотрен вариант реализации класса `Thread`

```
public class MainActivity extends Activity {

    TextView text;
    String s = "";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```

super.onCreate(savedInstanceState);
setContentView(R.layout.activity_list_item);
text = (TextView) findViewById(R.id.text1);

AnotherThread t = new AnotherThread();//создание потока
t.start();//запуск потока
for (int i = 0; i < 10; i++) {
    try{
        Thread.sleep(1000);    //Приостанавливает поток на 1 секунду
    } catch (InterruptedException e){}
    s = s + "1";
    text.setText(s);
}
}

class AnotherThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try{
                Thread.sleep(1000); //Приостанавливает поток на 1
секунду
            } catch (InterruptedException e){}
            s = s + "2";
            text.setText(s);
        }
    }
}
}

```

Приостановка потока нужна для того, чтобы четче отследить параллельность выполняемых потоков, цикл из 10 итераций выполнится моментально и скорее всего в текстовом поле выведутся сначала единицы, потом двойки. После запуска метода start() свою работу начал второй поток, а именно запустился метод run(). При реализации класса, наследуемого от Thread, теряется возможность наследования от других классов. Это связано с тем, что в Java запрещено множественное наследование классов (через extends), т.е. невозможна ситуация когда класс является потомком нескольких родительских классов.

Второй вариант создания нового потока через реализацию интерфейса Runnable. Интерфейс Runnable содержит один единственный метод run(). Конструкторы класса Thread могут взаимодействовать с этим интерфейсом, один из конструкторов Thread(Runnable target).

```

public class MainActivity extends Activity {

    TextView text;
    String s = "";

    @Override
    protected void onCreate(Bundle savedInstanceState) {

```

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_list_item);
text = (TextView) findViewById(R.id.text1);
Thread t = new Thread(new Runnable() {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            s = s + "2";
            text.setText(s);
        }
    }
});
t.start();
for (int i = 0; i < 10; i++) {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
    s = s + "1";
    text.setText(s);
}
}
```

Класс также может реализовывать интерфейс `Runnable`. В конструктор `Thread` нужно передать объект этого класса. Использование интерфейса `Runnable` лучше использовать, когда не нужно наследовать от класса `Thread` ничего лишнего, а только реализовать метод `run()`. Также если ваш класс для потока должен наследоваться от другого класса, не являющимся потомком `Thread`, то использование `Runnable` подходящий вариант.

3.4.3. Реализация логики потока

Итак, при создании нового потока, мы переопределяем метод **`run()`**. В нем размещается код, который нужно выполнить в новом потоке. Поток начинает свою работу с этого метода и заканчивает своё выполнение по окончании метода. В этом методе должна быть реализована логика потока.

Метод **`start()`** запускает новый поток и вызывает метод `run()`. В таком случае мы получим **асинхронные потоки**, это значит, что мы не знаем заранее в какой последовательности будут выполняться потоки. Если мы попытаемся вызвать метод `run()` для объекта самостоятельно, то мы получим обычный синхронный вызов функции, то есть управление к потоку, из которого мы сделали вызов, вернется только после завершения метода `run()`.

3.4.4. Синхронизация потоков

Использование асинхронных потоков может привести к ошибочному выполнению.

Рассмотрим пример, есть общий баланс **money = 100** и два пользователя, которые захотели снять определенное количество средств с баланса **amount1 = 80** и **amount2 = 90**. Существует функция для проведения этой операции, в которой указано, что если пользователь пытается снять сумму, которая больше баланса, то операция выполнена не будет. При использовании асинхронных потоков возможна такая ситуация, что проверка на возможность снятия средств с баланса пройдет одновременно и при текущем значении баланса в обоих случаях провести операцию снятия будет разрешено. В итоге мы можем получить непредсказуемый результат, баланс может уйти в минус, хотя разработчик был уверен, что обезопасил себя от этого.

Шаги	Процесс 1	Процесс 2	Переменная
0	money – amount1 > 0	money – amount2 > 0	money == 100
1	money -= amount1		
2		money -= amount2	
3			money == -70

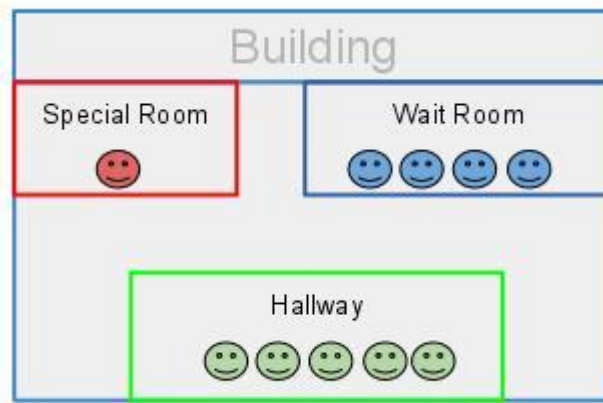
Для разрешения таких проблем в Java используется синхронизация. Механизм синхронизации основывается на концепции монитора.

Монитор - это объект специального назначения, в котором применен принцип взаимного исключения. Во время выполнения программы монитор допускает лишь поочередное выполнение процедуры, находящейся под его контролем.

У каждого объекта в Java имеется **свой собственный** неявный монитор. Когда метод типа synchronized вызывается для объекта, происходит обращение к монитору объекта чтобы определить, выполняет ли в данный момент какой-либо другой поток метод типа synchronized для данного объекта. Если нет, то текущий поток получает разрешение войти в монитор. Вход в монитор называется также *блокировкой(locking)* монитора. Если при этом другой поток уже вошел в монитор, то текущий поток должен ожидать до тех пор, пока другой поток не покинет монитор. Таким образом монитор Java вводит поочередность в параллельную обработку.

Также монитор можно представить как здание, в котором есть специальная комната. В этой комнате в каждый момент времени может находиться только один посетитель (поток). В этой комнате могут быть данные и операторы.

Если посетитель хочет занять эту комнату, то он должен вначале войти в прихожую (точка входа) и дождаться своей очереди. Администратор по своим критериям выбирает посетителя и разрешает войти в комнату пока она пуста. Если посетитель в специальной комнате по каким-то причинам заснул, то администратор перемещает его в комнату заснувших (Wait Room), чтобы они позже вновь смогли войти в специальную комнату (wait-потоки будут рассмотрены позже).



Объявление метода `synchronized` не подразумевает, что только один поток может одновременно выполнять этот метод. Имеется ввиду, что в любой момент времени только один поток может вызвать этот метод(или любой другой метод типа `synchronized`) для **конкретного объекта**. Таким образом, мониторы *Java* связаны с объектами, но не с блоками кода. Два потока могут параллельно выполнять один и тот же метод типа `synchronized` при условии, что этот метод вызван для разных объектов.

Мониторы не являются объектами языка *Java*, у них нет атрибутов или методов. Доступ к мониторам возможен на уровне собственного кода *JVM*.

Модификатор `synchronized` может быть применим для:

- метода экземпляра класса
- блока, синхронизированного на экземпляре класса

В следующем примере рассмотрена синхронизация внутри блока. В такой конструкции следует указывать объект, одновременный доступ к которому должен быть заблокирован.

```
public class MainActivity extends Activity {

    TextView text;
    String s = "";
    A a = new A();

    class A {
        public int count = 0;

        public void inc() {
            this.count++;
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_list_item);
        text = (TextView) findViewById(R.id.text1);
        Thread t = new Thread(new Runnable() {
```

```
@Override
public void run() {
    synchronized (a) {
        while (a.count < 5) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            a.inc();
            s = s + a.count;
            text.setText(s);
        }
    }
}

});
t.start();
synchronized (a) {
    while (a.count < 5) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        a.inc();
        s = s + a.count;
        text.setText(s);
    }
}
}
```

Попробуйте воспроизвести данный пример с синхронизацией и без неё. Скорее всего результат будет различным.

Применять блочную синхронизацию удобно, когда нужно синхронизировать не весь метод целиком. Например, вам нужно обеспечить чтение/изменение массива из разных потоков. Также если разработчик не имеет доступа к какому-либо методу, потому что не он его разработал. Тогда можно поместить вызов такого метода внутрь блока с синхронизацией.

У объектов также существуют несколько методов для организации работы потоков. Данные методы можно применять только внутри блока `synchronized` к блокируемому объекту.

<code>wait()</code>	Вызывается внутри синхронизированного блока или метода, останавливает выполнение текущего потока и высвобождает захваченный им объект.
<code>notify()</code>	Метод также может быть вызван только внутри синхронизированного блока.

	Возвращает блокировку объекта потоку, из которого был вызван метод wait(). Если их несколько, то поток выбирается случайно.
notifyAll()	Пробуждает все потоки, из которых был запущен метод wait() для данного объекта.

Также существует метод join() применяемый к потоку. Этот метод ожидает завершение потока для которого он вызван. В параметрах метода join() можно указать максимальное время ожидания.

Метод wait останавливает выполнение текущего потока, пока не будут вызваны notify или notifyAll. Т.к. пробудиться поток может по разным причинам (notify предназначался другому wait, был вызван notifyAll и т.д.), то поток останавливают с определенным условием. Но условие остановки должно регулироваться другим потоком и для потокобезопасного изменения данных как раз и нужны синхронизации. Если synchronized не будет прописан, то будет выброшено исключение `IllegalMonitorStateException`.

```
class MyHouse {
    private boolean pizzaArrived = false;

    public void eatPizza() {
        synchronized (this) {
            while (!pizzaArrived) {
                try {
                    wait();
                } catch (InterruptedException e) {
                }
            }
            System.out.println("yumyum..");
        }
    }

    public void pizzaGuy() {
        synchronized (this) {
            pizzaArrived = true;
            notifyAll();
        }
    }
}
```

Здесь поток, желающий поесть пиццу (метод eatPizza), будет вынужден ждать разносчика пиццы из другого потока (метод pizzaGuy), пока он не принесет ее и не оповестит об этом.

Может показаться, что ждущий пиццу заблокирует объект MyHouse (блок synchronized) и разносчик пиццы не сможет к нему, т.к. ему требуется тоже блокировка. Но это не так. При вызове wait поток отпускает блокировку. Если кто-то вызовет notify (notifyAll), то поток пробуждается, вновь захватывает блокировку и продолжает выполнение после команды wait.

3.4.5. Блокировки

Существует понятие **мертвой (взаимной) блокировки** (deadlock). Ситуация, когда потоки находятся в бесконечном ожидании ресурсов. Рассмотрим причину ее возникновения.

Например, есть два потока. И первому, и второму потоку нужны объекты **a** и **b**. Возможно такое, что первый поток сперва заблокирует, к примеру объект **a**, в это время второй поток заблокирует объект **b**. В итоге первый поток не сможет начать выполнения, пока не получит объект **b**, а второй поток уже не сможет его освободить, так как ничего не может сделать, пока не получит объект **a**. Таким образом возникнет состояние взаимной блокировки (deadlock). Потоки будут ждать нужных ресурсов бесконечно.

Шаг	Поток 1	Поток 2
0	Блокировка объекта a	Блокировка объекта b
1	Ожидание объекта b	Ожидание объекта a
2	deadlock	deadlock

В данном примере вероятность блокировки мала, но если представить, что в этом участвует гораздо больше потоков и ресурсов, то блокировка вполне вероятна.

Упражнение 3.4.1

Разобрать пример программы, совершающей загрузку картинки из интернета и устанавливающей ее на экран.



Картинку из интернета можно загрузить по ссылке. Для работы с ссылками в Android есть класс `URL`. Один из конструкторов класса `URL` позволяет передать в него текстовую ссылку.

```
URL url = new URL("http://...");
```

Используя метод `getContent()` для объекта `URL` можно получить ресурс который располагается по ссылке `URL`. Метод возвращает поток данных `InputStream`, или `null` в случае неудачи.

Графический ресурс `Bitmap` можно получить из потока `InputStream` с помощью метода `BitmapFactory.decodeStream(InputStream)`.

Задание 3.4.1

Модифицируйте программу из упражнения так, чтобы в случае ошибки происходило несколько попыток скачивания картинки из интернета.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю ИТ ШКОЛЫ SAMSUNG Вихрову Виктору Андреевичу.