مالتی تردینگ

مالتی تردینگ یکی از ویژگیها یی هست که زبان جوا از ابتدا در دسترس گذاشته بود و باعث می‌شه که ۲ یا چند پرسس باهم انجام شود که باعث استفادهٔ بهینه از سی پی یو میشود.

هر قسمت آذین همروندی در اجرای برنامه یک ترد نامیده میشود.

در واقع در حالت عادی کد زنی هم یک ترد ساخت میشود که همه کارها بوسیله آن ترد انجام میشود که به این ترد ، ترد اصلی (main thread) گفت میشود .

در جاوا از دو مدل میتونیم از ترد استفاده کنیم روس اول استفاده از ارث بری و روس دوم استفاده از اینترفس است.دو روش و به طور مختصر پایین توضیح دادم و چند نمونه کد را که به نظرم بهتر بود را قرار دادم که باعث فهمه بهتر میشود

روش اول استفاده از ارث بری از کلاس java.lang.Theared

این کلاس اجازه ی override متودی به نام run را به ما می دهد وقتی ما ازین کلاس object می سازیم می توان با استفاده از آن متود start را فراخوانی کنیم .

```
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            System.out.println ("Thread " +
                  Thread.currentThread().getId() +
                  " is running");

        }
        catch (Exception e)
        {
            System.out.println ("Exception is caught");
        }
    }
}

public class Multithread
{
    public static void main(String[] args)
```

```
    {
        int n = 8;
        for (int i=0; i<n; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
```

روش دوم استفاده از interface Runnable

وقتی در جاوا صحبت از interface میشود منظور کلاس هایی هستند که متد های آن بصورت بدون بدنه نوشته شده اند و فقط امضای آن ها مشخص است. اگر این interface ها را در کلاسمان implement کنیم باید متد های آن را بازنویسی کنیم وگرنه با خطای کامپایلر مواجه می شویم .

این interface هم اینگونه است و در هنگام پیاده سازی باید متد run را override کنیم وبا ساختن ترد به شکل زیر متود start را فراخوانی کنیم .

```
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
             System.out.println ("Exception is caught");
        }
    }
}

class Multithread
{
    public static void main(String[] args)
    {
        int n = 8;
        for (int i=0; i<n; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
```

```
            object.start();
        }
    }
}
```

فرق بین روش اول و روش دوم :

۱- اگر از روش اول یعنی ارث بری استفاده کنیم کلاس ما دیگر نمی تواند از کلاس دیگری ارث بری کند زیرا همانطور که می دانید در جاوا بر عکس c++ ارث بری چند گانه نداریم پس اگر مایلیم کلاس ما از کلاس دیگری ارث بری کنه باید از روش دوم استفاده کنیم.

۲- با استفاده از روش اول می توانیم یکسری متود پایه که در کلاس Thread از قبل نوشته شده رو استفاده کنیم قاعدتا interface این ویژگی رو نداره. دو متد که در کلاس Thread هست که کمتر استفاده شده عبارت است از: yield() و intrupt که توضیح و نحوه استفاده ازشون رو نوشتم.

Yield :

این متود یک متود استاتیک است , زمانی که استفاده می کنیم به ما هجازه می دهد که محاسبات تردی که در حال اجرا است و این متد برای آن ترد صدا زده شده است متوقف شود و تردی دیگد با اولوییت یکسان با این ترد شروع به کار کند. اگر زمانی که این متد فراخوانی می شود ترد دیگری وجود نداشته باشد یا اینکه بقیه ترد ها اولویت کمتری داشته باشند همین ترد مجدد به کار خود ادامه می دهد.
ویژگی خوب این متد این است که شانس اینرو به ترد منتظر مانده ی دیگری که اولویت یکسان داره ئیدهد تا اجرا شود .
برای مثال اگر زمان اجرای یک ترد از یک حدی بیشتر شود اجازه می دهد که ترد منتظر مانده ی دیگر cpu را در دست بگیرد و کارش را انجام دهد.
مثال زیر به زیبایی کار این متود را به نظر من نشان می دهد :

```
class MyThread extends Thread {

   public void run() {

      for (int i = 0; i < 5; ++i) {

         Thread.yield();

         System.out.println("Thread started:" +
Thread.currentThread().getName());

      }

      System.out.println("Thread ended:" +
Thread.currentThread().getName());

   }

}

public class YieldMethodTest {

   public static void main(String[] args) {

      MyThread thread = new MyThread();

      thread.start();

      for (int i = 0; i < 5; ++i) {

         System.out.println("Thread started:" +
Thread.currentThread().getName());

      }

      System.out.println("Thread ended:" +
Thread.currentThread().getName());

   }

}
```

Intrupt :

اگر تردی در حال sleep یا wait باشد با فراخوانی این متد برای آن ترد , از این دو حالت نام برده خارج می شود . اگر در حالت sleep باشد که بیدار میشود و به کار خود ادامه می دهد ولی اگر در حالت wait باشد exeptione مربوط به خود را throw می کند ولی اگر زمانی که ترد در حالت عادی به سر می برد و در حالت اجرا می باشد این متد را فراخوانی کنیم هیچ تغییری در حالت ترد ایجاد نمی شود و به طور عادی ادامه حیات می دهد فقط متغییر بولینی به نام خود این متد یعنی intrupt در این حالت true میشود که در حالت عادی false است .

چرخه ی حیات thread :

از زمانی که ترد ما استارت می خورد چرخه ی حیات ترد شروع شده و متد runnable فراخوانی می شود ودستورالعمل های مارا انجام می دهد تا زمانی که این ترد به حالت sleep در بیاید که یعنی به میزان زمانی که ما مشخص کردیم متوقف میشود سپس دوباره خودکار متود runnable آن فراخوانی می شود یا به حالت wait برود که ترد متوقف می شود و تا زمانی که دستور اجرای آنرا ندادیم در حالت standby قرار می گیرد ویا اینکه عملیات خود را به پایان میرساند وبه حالت dead در می آید و distruct می شود .

Critical Section :

وقتی صحبت از ترد می شود همیشه اسم مبحثی به نام Critical section مطرح میشود . که یکی از موقعیت های معمول هست در برنامه نویسی همزمان یعنی استفاده ی چند ترد وقتی همه ی این جند ترد بخواهند از منبع واحد و مشترکی استفاده کنند .

این موقعیت که ترد های ما اجازه ی خواندن و نوشتن روی یک منبع واحد مثل یک table از یک دیتا بیس داشته باشند این خواندن و نوشتن های همزمان ممکن هست باعث خطاهایی شود یا باعث نا هماهنگی در سیستم شود . ما

باید با ساختن مکانیزمی جلوی این خطا ها را بگیریم . این موقعیت را وضعیت مسابقه بین ترد ها برای استفاده از منبع مشترک (race condition) می گویند و زمانی اتفاق می افتد که ترد ها همگی به منبع مشترک دسترسی داشته باشند و در این مواقع نتیجه نهایی و خروجی بدین صورت تعیین میشود که کدام ترد اول به منبع مشترک دسترسی پیدا میکند که این باعث میشود در اکثر مواقع جواب ما اشتباه شود.

اتفاق ناگوار دیگر این است که یک ترد یک value را تغییر دهد و قبل از write کردن آن در منبع مشترک زمانی که اول فقط در کش خود تغییر کرده ترد دیگر وارد شود و از مقدار قدیمی و اشتباهی که وجود دارد استفاده کنو راه حل critical section است.

Critical section قسمتی از کد هست که به منابع مشترک متصل است و اجازهی دسترسی فقط به یک ترد را می دهد جاوا برای راحت تر شدن این کار مکانیزمی به نام synchronization را ایجاد کرده است .

این مکانیزم دو حالت true و false دارد وقتی false باشد ترد ها اجازه ی ورود به این بلاک را دارند واگر true باشد ترد ها منتظر میمانند تا عملیت داخل این بلاک تمام شود ومقدار false شود سپس jvm یک ترد منتظر مانده را انتخاب می کند و اجازه ی ورود به بلاگ را می دهد .