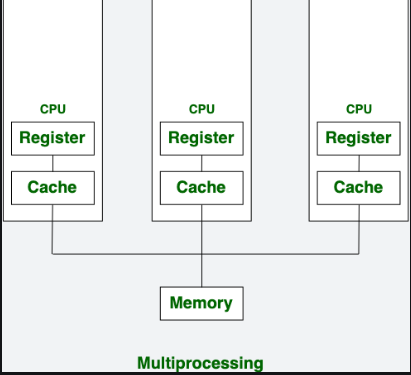
1. Giới thiệu Multitasking và multithreading

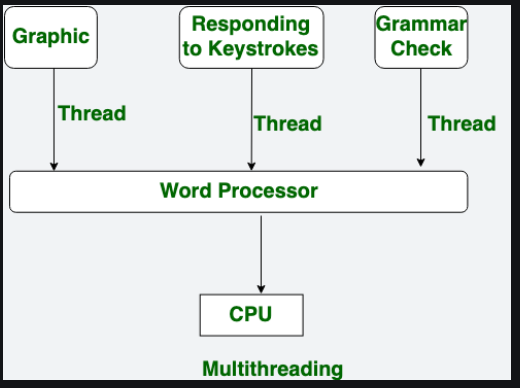
A, Multitasking

+ Multitasking là khả năng hệ thống có thể thực hiện nhiều tác vụ đồng thời, nó có thể đạt được thông qua 2 cách là sử dụng multiple process hoặc multiple thread. Các tiến trình và luồng là các phiên bản thực thi của các tác vụ này

Multiprocessing đề cập đến khả năng thực hiện đồng thời nhiều hơn một quy trình trong hệ thống máy tính.1 process có thể hiểu đơn giản là 1 chương trình khi được thực thi.Các process không sử dụng chung vùng nhớ, không truy cập trực tiếp dữ liệu của nhau. => các process không liên quan tới nhau . các process sẽ không thể trực tiếp truy cập vào vùng nhớ của nhau. Vậy, ta có thể gián tiếp làm điều đó bằng cách để các process giao tiếp với nhau. Thuật ngữ diễn tả hành động đó là **Inter-process communication (IPC)**.



Vd : khi click vào inteliji và skype để thực hiện công việc code và liên lạc => 2 process dc tạo ra, cta có thể trao đổi qua lại giữa 2 process để thực hiện công việc , nhưng dữ liệu của 2 bên không ảnh hưởng tới nhau.

B, Thread là 1 thành phần của process, nằm trong process, 1 process có thể có 1 hoặc nhiều thread. Multithreading có thể hiểu là việc một chương trình có nhiều luồng thực thi đồng thời.Và các thread chia sẻ dữ liệu với nhau, chúng có thể truy cập tới data của nhau. 

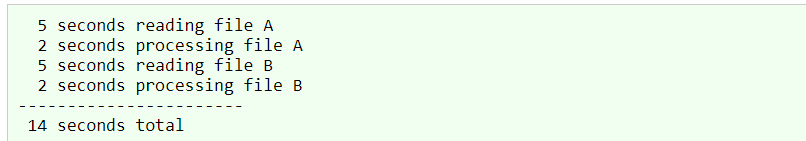
Sự khác nhau giữa thread và process:

|  |  |
| --- | --- |
| Thread | Process |
| Là 1 thành phần trong process | Process chứa 1 or nhiều thread |
| Có thể trao đổi dữ liệu giữa các thread | Không có sự chia sẻ dữ liệu giữa các process |
| Việc chuyển đổi ngữ cảnh ( context switch) yêu cầu ít tài nguyên hơn process | Việc chuyển đổi ngữ cảnh trong process yêu cầu nhiều tài nguyên hơn |

Tại sao cần sử dụng multithreading?

1. Sử dụng CPU tốt hơn.

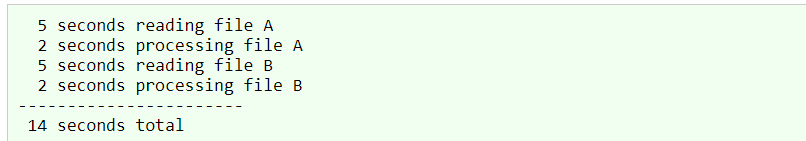
Theo dõi ví dụ sau :



Nếu ứng dụng thực hiện đọc và xử lý file theo thứ tự, thực hiện trên 1 thread , đọc và xử lý file theo thứ tự thì sẽ tốn 14s

Vậy còn cách nào có thể thực hiện tốt hơn?

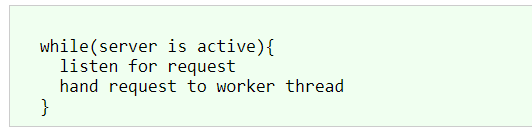
Đó là khi thực hiện đọc xong file A, thực hiện đọc luôn file B . Trong quá trình file B đang được đọc, lúc này CPU đang rảnh, có thể thực hiện xử lý file A – chỉ cần 2s, sau đó là thực hiện xử lý file B :



1 ví dụ tổng quan hơn. Trong quá trình làm 1 bài thuyết trình, cta cần tải 1 video để minh họa, quá trình tải này sẽ tốn 20 phút, thay vì phải chờ đợi tải xong video mới làm tiếp bài thuyết trình được thì cta có thể làm tiếp phần slide, và chờ tải xong video sẽ tiếp tục xử lý phần cần nội dung video đấy.

1. Tăng trải nghiệm của người dùng – đáp ứng phản hồi nhanh hơn

Giả sử khi một người dùng thực hiện gửi yêu cầu, sau đó server thực hiện xử lý, rồi gửi lại phản hồi, và người sau mới dc gửi yêu cầu tới tiếp. Nếu có 1000 người phải đợi như thế, sẽ dẫn tới sự khó chịu và khó tiếp cận . Thay vì như thế, sau khi nhận request, yêu cầu sẽ được đưa cho 1 luồng khác để thực thi và luồng nhận request sẽ ngay lập tức trở lại để nhận yêu cầu từ người dùng mới => tăng tốc độ phản hồi và tương tác



Bất lợi :

1. Độ phức tạp cao hơn

Bởi vì các thread chia sẻ tài nguyên dùng chung, giả sử 2 thread A, B , cùng thực hiện đọc và thay đổi giá trị của biến đấy, vậy thì lúc này, biến đấy sẽ lấy giá trị tại thread A, hay thread B

* Vấn đề này sẽ gây ra race condition. Tức là các luồng có thể cạnh tranh truy cập vào tài nguyên chung

1. Tăng việc tiêu thụ tài nguyên

* Tạo và quản lý nhiều luồng sẽ tiêu tốn tài nguyên hệ thống như bộ nhớ và CPU, và việc chuyển đổi giữa ngữ cảnh các luồng ( context swtiching ) có thể gây chi phí cao.
* Context swtiching là việc khi mà cần lưu lại dữ liệu của 1 luồng đang thực hiện trước khi chuyển qua luồng khác để thực hiện và sau đó lấy lại dữ liệu của luồng cũ để thực hiện tiếp

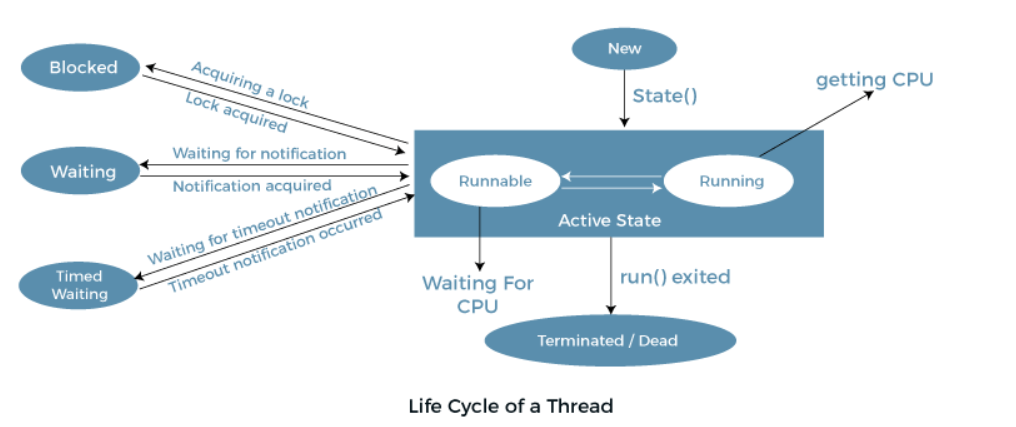
II. Thread

1. 1 thread là 1 luồng thực thi trong chương trình. JVM cho phép 1 chương trình có nhiều thread thực thi đồng thời. Và mỗi thread có độ ưu tiên riêng, thread có độ ưu tiên lớn hơn sẽ dc thực thi trước.

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

**Priority:** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

**Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.



1. New : Khi 1 thread được tạo, nó ở trạng thái new. Các đoạn code chưa được thực thi khi ở trạng thái này. Vd : final var firstThread = new Thread(() -> System.out.println("First thread"));
2. Active: Khi 1 thread gọi phương thức start(). Sẽ chuyển từ trạng thái new => active. Trạng thái active bao gồm 2 trạng thái là Runable và Running.

+ Runable: Một thead sẵn sàng để chạy thì sẽ chuyển vào Runable state

+ Running: Khi thread đạt dc CPU, nó chuyển từ trạng thái runable sang running

1. Blocked or waiting

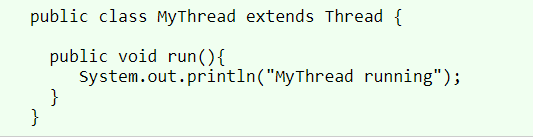
Thread ,khi nó không hoạt động nó sẽ ở trạng thái blocked ( nghĩa là khi 1 thread truy cập cào tài nguyên, thì các thread khác sẽ có trạng thái blocked,chờ đến khi thread khác thực hiện xong, sau đó nó đưa ra 1 lock để các thread khác truy cập lại vào đoạn code) vd khi việc sử dụng nhà tắm, 1 người dùng, khóa cửa nhà tắm, thì 4 người còn lại sẽ bị blocked hoặc **Waiting** là trạng thái thread này chờ thread khác và không có thời gian cụ thể.

1. Timed waiting : Nó tương tự ở trạng thái waiting nhưng sẽ được gọi lại để chạy sau một khoảng thời gian cụ thể bằng cách sử dụng sleep()
2. Terminated: Thread kết thúc khi hoàn thành công việc của nó,.
3. The creation in Java

Trong java, có thể thực hiện tạo thread bằng 2 cách : extend Thread Class or implement Runable interface.

* 1. Tạo thread sử dụng extend Thread Class

1. Tạo subclass extend Thread class và override run() method. Phương thức run() sẽ được thực thi sau khi thread gọi phương thức start().

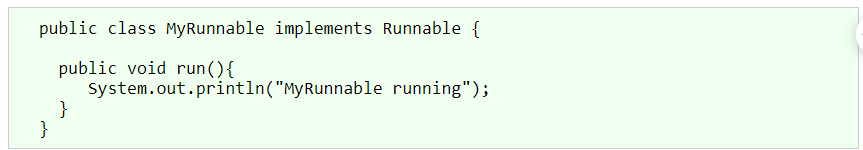


1. Tạo 1 thread và thực hiện chạy bằng phương thức start():

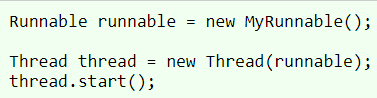


* 1. Tạo thread bằng cách implementation Runnable interface

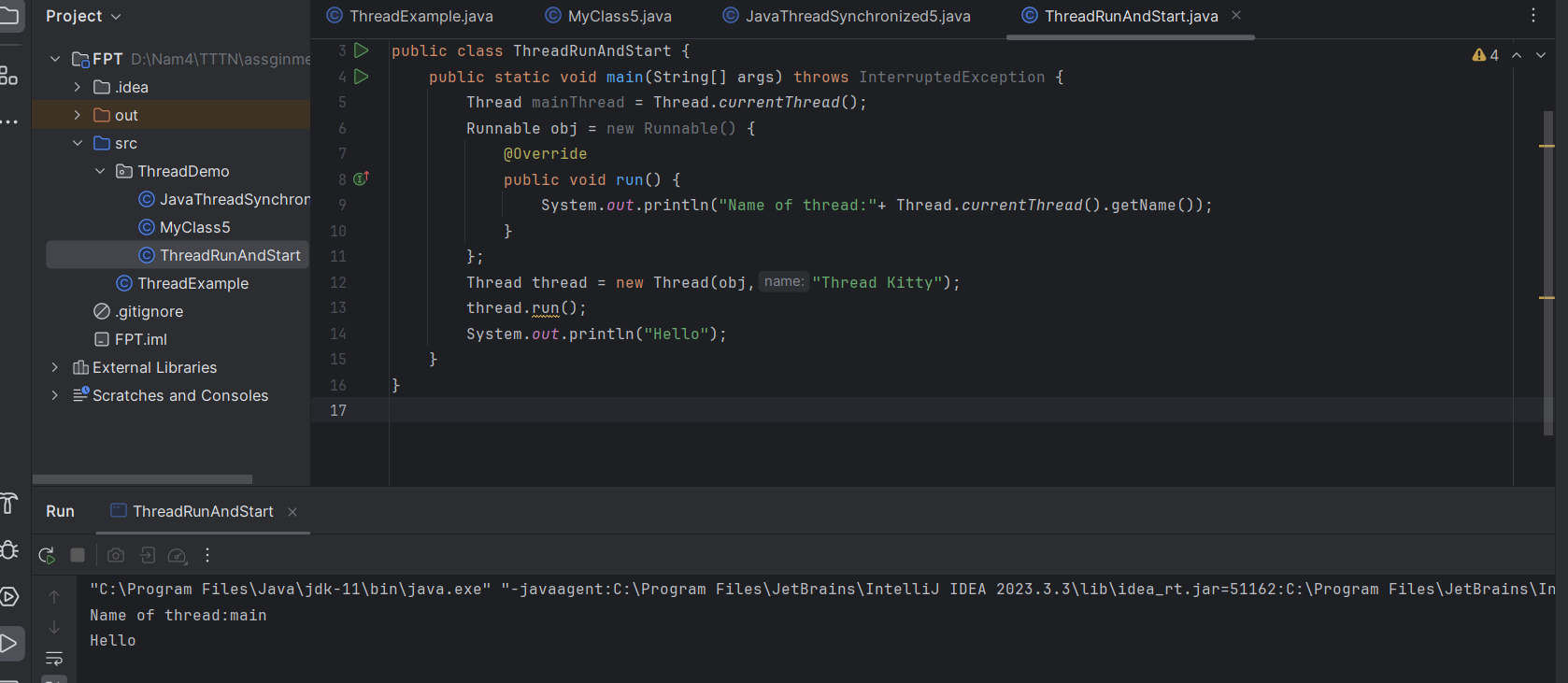
1. Tạo 1 class implemantation Runnable interface

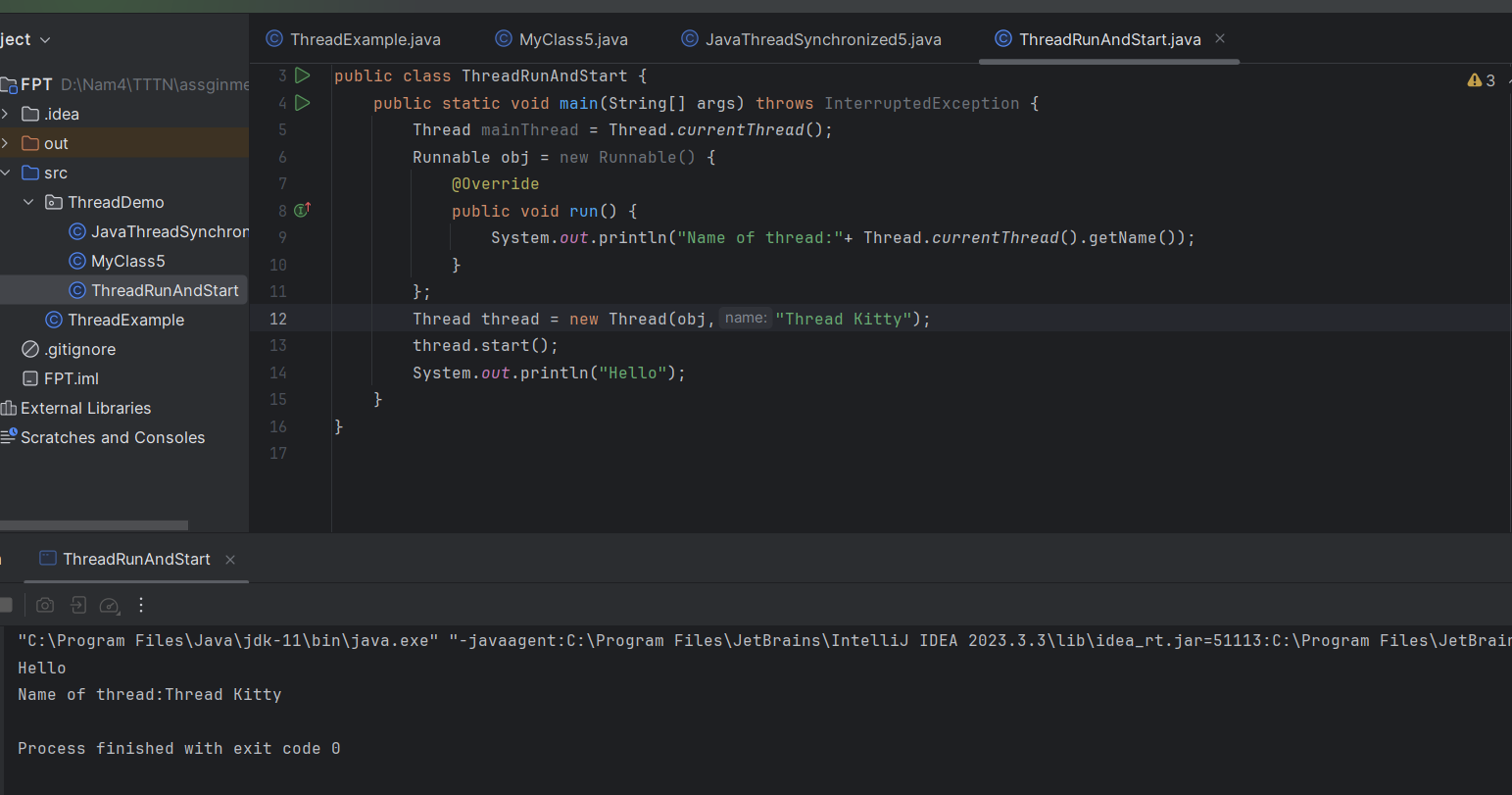


1. Tạo 1 thể hiện của class MyRunable và truyền vào constructor của Thread



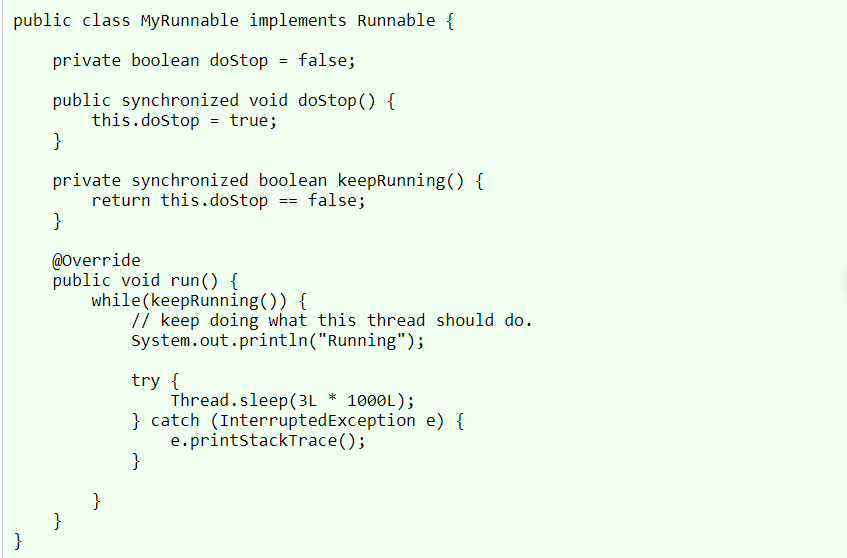
* 1 Lỗi phổ biến khi tạo và thực thi thread, đó là thay vì gọi start() để thực thi thì sẽ gọi run()

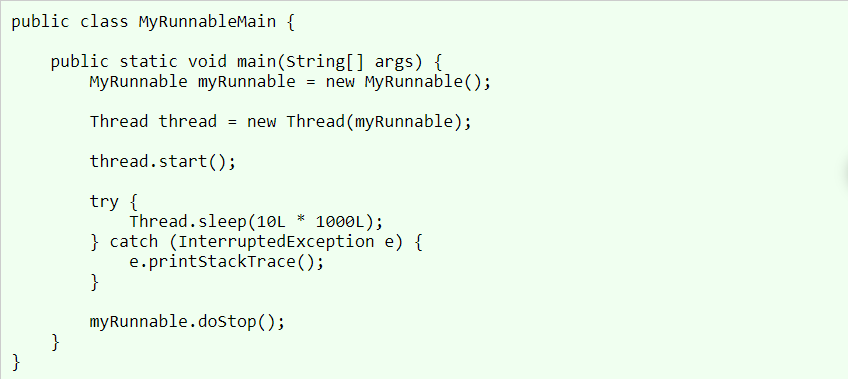


* Nếu ko gọi start() , mà gọi run() , nó sẽ không tạo ra luồng mới, có nghĩa là nó sẽ thực thi ngay trong main như một phương thức bình thường => cho ra kết quả như hình trên.
* Còn khi gọi start(), nó sẽ tạo ra 1 thread mới, và in ra kết quả như sau :
* 

Dừng 1 thread

Để dừng 1 thread, trước kia sử dụng phương thức stop(), nhưng hiện giờ nó không còn được triển khai nữa. Thay vì gọi stop() , sẽ sử dụng flag để triển khai code để dừng 1 thread.

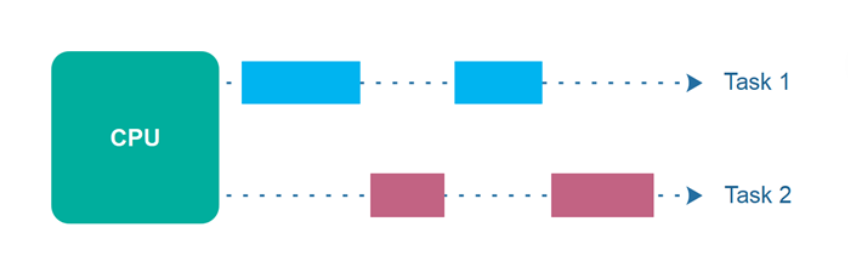




Ta tạo ra 2 phương thức và sử dụng flag doStop để đánh dấu, doStop = true => thread dừng lại, doStop = false => thread tiếp tục thực hiện

1. Concurrency and Parallel
   1. Concurrent execution

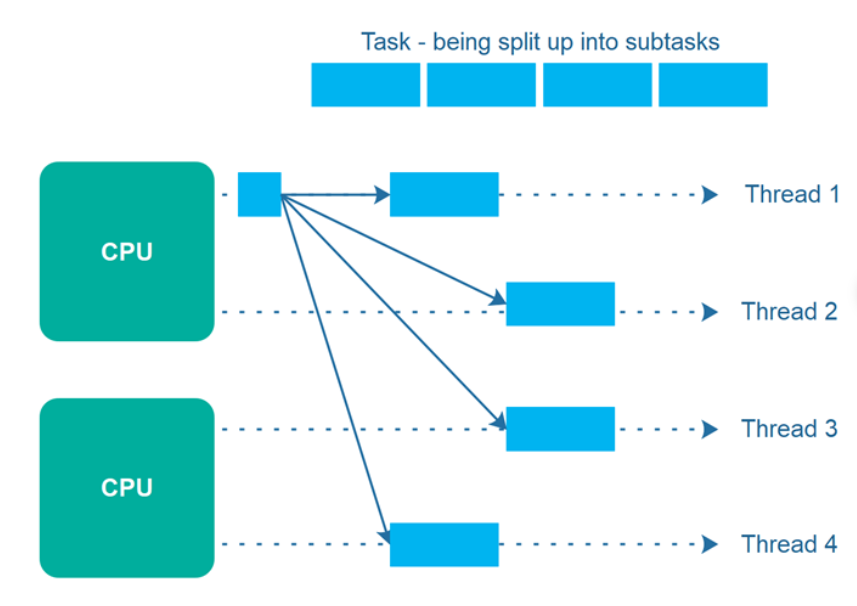
Concurrency có nghĩa là 1 chương trình có thể thực hiện một hoặc nhiều task tại cùng một thời điểm. Để thực hiện điều này, CPU sẽ chuyển đổi qua lại giữa các task trong suốt quá trình thực hiện. Điều này có nghĩa là các thread sẽ chạy xen kẽ nhau.



Giả sử ông A cần hoàn thành công việc trong ngày bao gồm đi lấy hộ chiếu và chuẩn bị slide cho buổi họp hôm sau. Ông A đi tới văn phòng hộ chiếu mất 5 tiếng, và phải chờ thủ tục mất 2 tiếng. Trong thời gian chờ, ông A sẽ lấy máy ra và chuẩn bị slide và chờ cho tới khi được gọi để lấy hộ chiếu. sau đó ông A về nhà chỉ cần hoàn thành nốt phần slide còn lại mất 1 tiếng là hoàn thiện.Mọi việc vẫn dc hoàn thiện đồng thời.

3.2.Parallel execution

Parallel có nghĩa là 1 chương trình chia các task của nó thành các task nhỏ hơn ( subtask). Và mỗi subtask sẽ được thực hiện song song, mỗi subtask được thực hiện trên 1 thread riêng biệt.



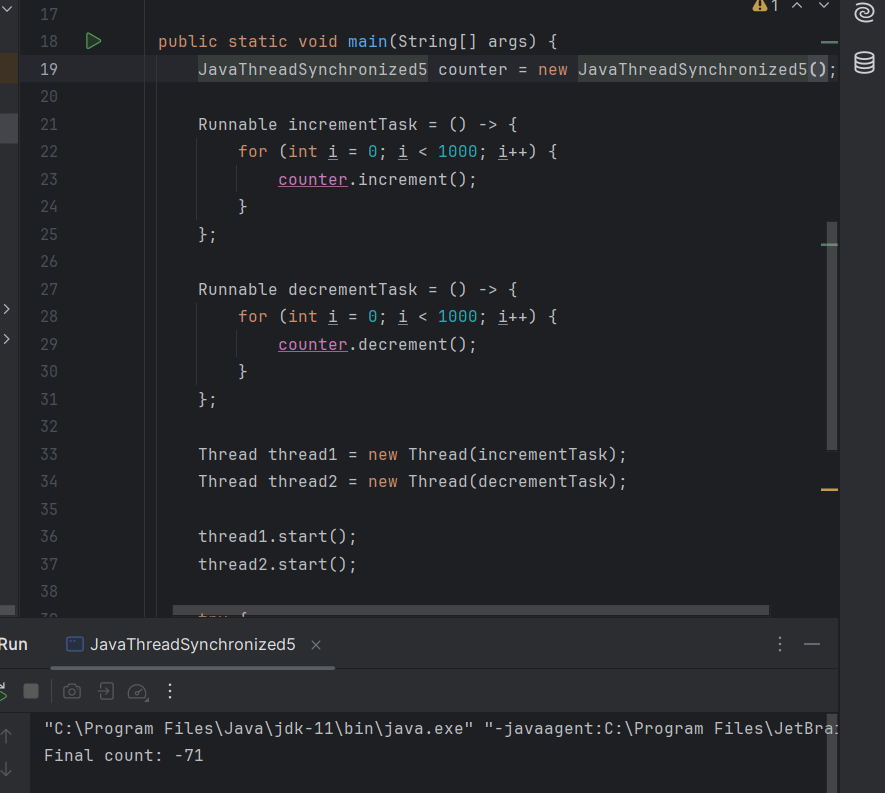
Theo ví dụ trên, công việc của ông A gồm có lấy hộ chiếu và làm slide, nếu ô A làm sếp và có 1 nhân viên đáng tin cậy, ông A giao cho anh ta nhiệm vụ làm slide, anh ta có thể tự mình thực hiện việc làm slide một mình, không cần làm phiền tới ông A, vì thế ông A có thể tập trung vào việc lấy hộ chiếu mà không cần tự làm slide nữa. Sau khi lấy xong hộ chiếu, ông A về nhà sẽ nhận dc mail kết quả chứa slide. Mọi việc đều được diễn ra một cách song song mà không phải thực hiện chuyển đổi như concurent execution.

1. Các vấn đề thường gặp khi thao tác với multithreading
   1. Race Condition

Định nghĩa : critical section là đoạn code được thực thi bởi nhiều luồng và sự thực thi của mỗi thread gây ra sự khác nhau về kết quả hiện tại.

Race condition xảy ra khi hai hay nhiều luồng cùng thực hiện đọc và ghi cùng 1 biến. Chia làm 2 loại :

1.Read – modify-write: 2 hay nhiều luồng trước hết đọc biến, sau đó sửa giá trị của nó và viết lại vào biến.

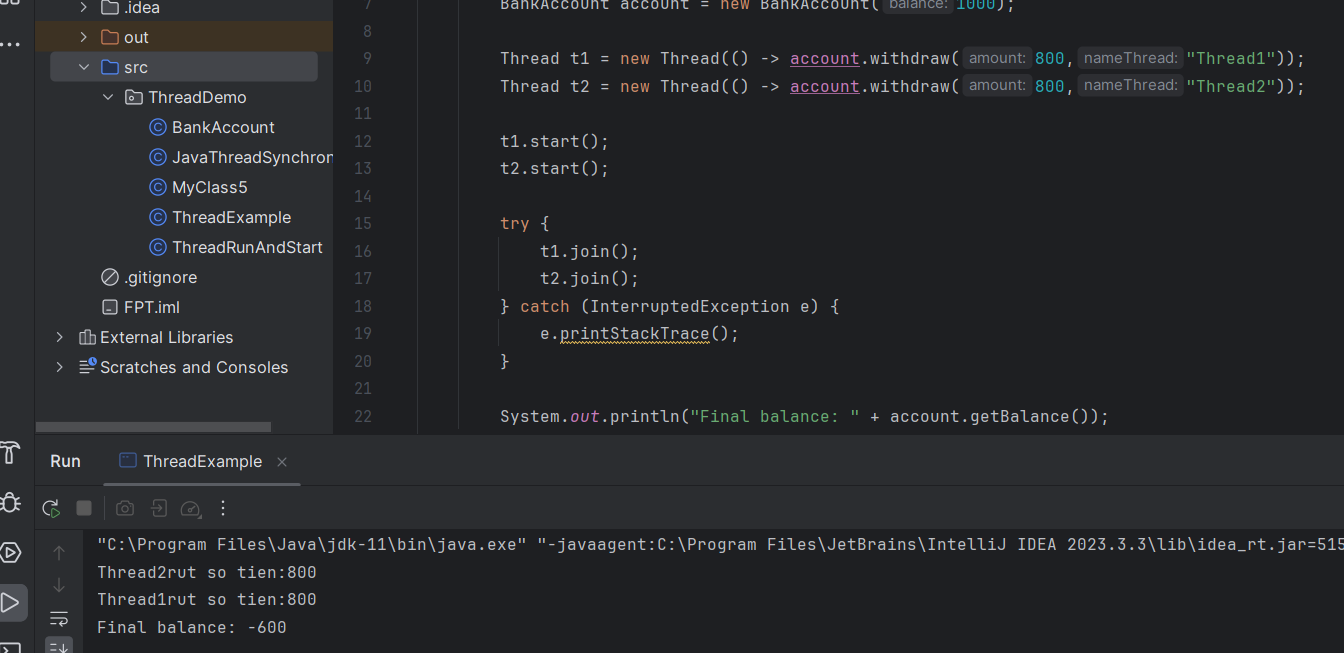


2.Check-then-act

2 luồng cùng check cùng 1 điều kiện, sau đó thực thi dựa vào điều kiện. Điều này cũng gây ra race condition



Tập trung vào việc kiểm tra một điều kiện trước khi thực hiện một hành động. Thường được sử dụng trong các trường hợp cần xác nhận điều kiện trước khi thay đổi trạng thái hoặc tài nguyên.



1. Java synchonized

Synchronized là chỉ cho phép 1 thread duy nhất tại 1 thời điểm cụ thể hoàn thành 1 task nhất định. Mang lại tính nhất quán cho chương trình.

Sử dụng synchronized với các trường hợp sau:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods
   1. Synchronized instance methods

Khi một phương thức instance được khai báo với từ khóa synchronized, khóa (lock) được sử dụng là đối tượng hiện tại (this).

 **Instance Method:** Khóa là đối tượng instance hiện tại. Điều này có nghĩa là nếu bạn có nhiều đối tượng instance của cùng một class, mỗi đối tượng có khóa riêng của nó. Các luồng khác nhau có thể truy cập đồng thời vào các phương thức synchronized của các đối tượng khác nhau.

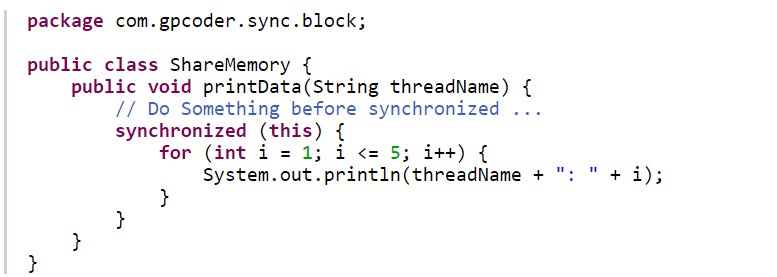
 **Static Method:** Khóa là class object. Điều này có nghĩa là tất cả các phương thức synchronized static của cùng một class chia sẻ cùng một khóa. Chỉ một luồng có thể truy cập bất kỳ phương thức synchronized static nào của class đó tại một thời điểm.

Ví dụ về synchronized trong static method : đếm số lần đăng nhập vào hệ thống, cần 1 biến đếm cố định static method countLogin;

* 1. Synchronized block

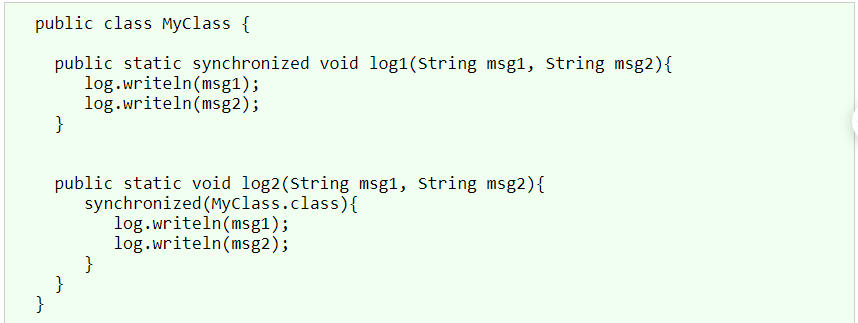
Khi sử dụng Synchronized Block, chúng ta cần phải cho biết khóa của đối tượng nào bạn muốn sử dụng làm khóa trong đoạn code cần đồng bộ. Bằng cách đó, chúng ta cũng có thể sử dụng đối tượng khác làm khóa.

Ví dụ với synchronized block đối với instance method

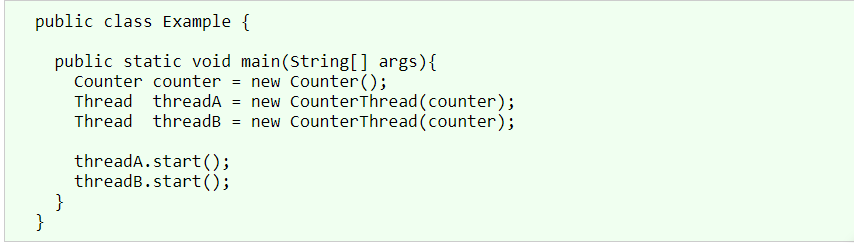


Khi mà không muốn đồng bộ cả phương thức, mà chỉ muốn đồng bộ 1 phần, nghĩa là có những tác vụ trong phương thức có thể cho phép truy cập từ nhiều nguồn. Ví dụ khi thực hiện mua sản phẩm, bao gồm xem thông tin sản phẩm và mua sản phẩm, thì việc xem thông tin sản phẩm không cần phải đồng bộ hóa, các luồng có thể truy cập để xem thông tin sản phẩm, còn việc mua phải dc đồng bộ để đảm bảo số lượng của sản phẩm

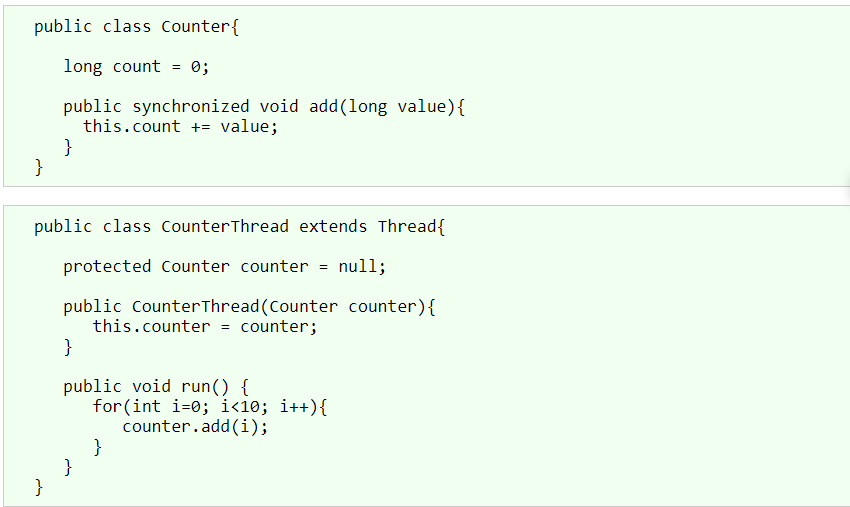
Tương tự, ta cx có thể tạo synchronized block với static method:



* Notice:

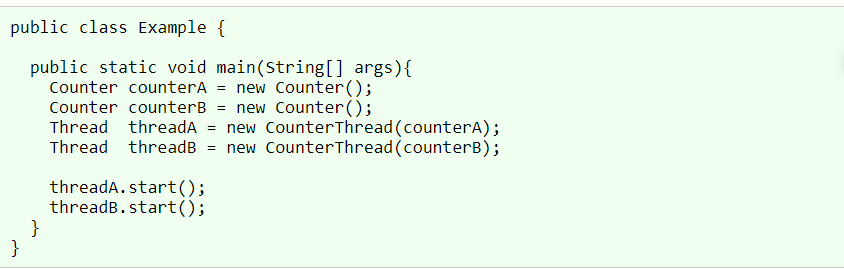


=>> ở đây, là 2 thread cùng truy cập vào cùng 1 instance là counter



Do đó, khi truy cập vào method add, nó sẽ tăng giá trị count trong object counter của class Counter., vào mỗi thời điểm chỉ có 1 thread truy cập vào phương thức add=> kết quả sẽ là ( 0 + 1+… + 10 ) + ( 0 + 1 + … + 10)

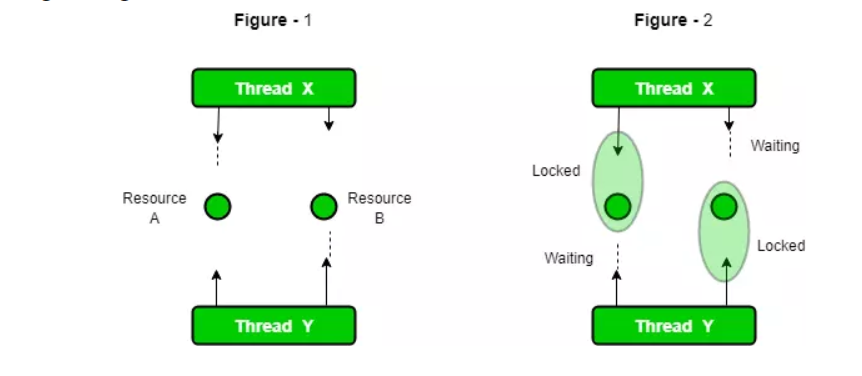
Tiếp đó là khi tạo ra 2 instance của class Counter:



2 luồng này không cùng tham chiếu tới cùng 1 instance, do đó, việc gọi thread A sẽ không ảnh hưởng tới threadB, tức là 2 luồng không phải đợi luồng kia thực hiện xong r mới thực hiện. Mà nó truy cập vào intance của riêng nó, và kết quả của 2 luồng ( count của counterA sẽ bằng ( 1 + .. + 10), count của counterB sẽ = ( 1 + .. + 10).

2 . Một synchronized block phải được đồng bộ hóa trên 1 object để đảm bảo chỉ có 1 luồng có thể thực hiện đoạn mã tại một thời điểm trên object cụ thể đó. Nhưng không khuyến khích sử dụng String object or primitive wrapper object – k cần cho vào slide

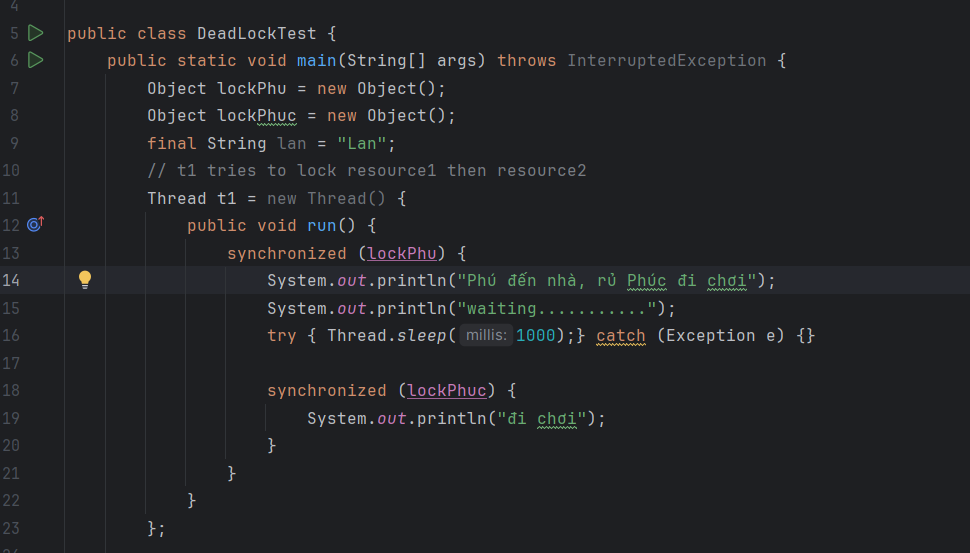
1. DeadLock
   1. DeadLock

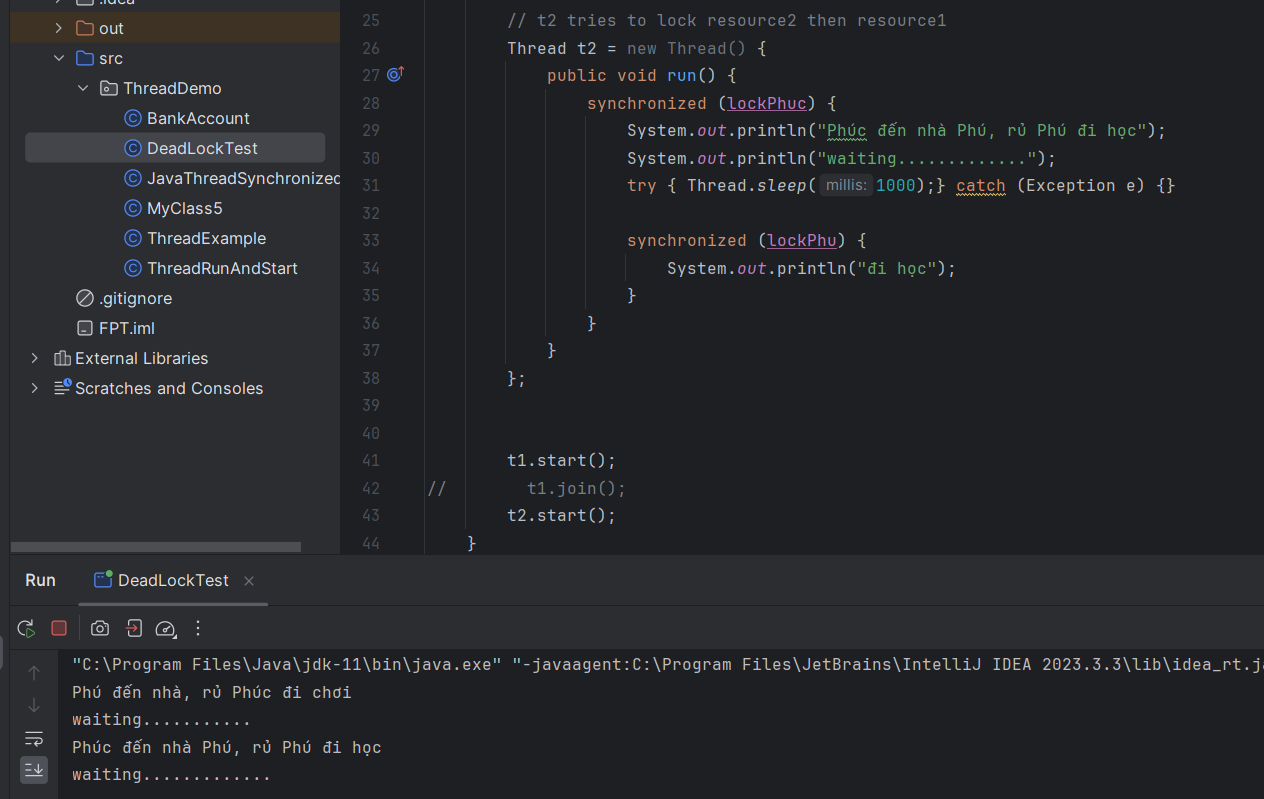


Cả 2 luồng hoạt động đồng thời, threadX chiếm A, threadY chiếm B, sau đó threadX cần làm gì đó và cần tài nguyên ở B, cần chiếm khóa B nhưng lúc này khóa B đang bị block bởi threadY. Đồng thời, threadY cũng cần tài nguyên ở A, và cần chiếm khóa A. Có thể thấy, cả 2 thread đều cần khóa của mỗi bên, nhưng không ai nhả ra, dẫn đến tình trạng “kẹt”, chờ vĩnh viễn.

**Deadlock xảy ra khi mà 2 luồng giữ khóa của nhau và đều đang chờ để có thể lấy đc khóa của đối phương**

Xét ví dụ sau :





1. Starvation

Có thể hiểu là khi 1 thread không được thực hiện trong thời gian dài or không dc thực hiện bởi vì các threads khác được ưu tiên thực thi.

+Có thể xảy ra khi 1 thread có độ ưu tiên thấp hơn so vs các luồng còn lại ( 10 luồng độ ưu tiên 8, 1 luồng ưu tiên 3 => luồng ưu tiên 3 sẽ phải đợi các luồng khác truy cập )

+ Có thể xảy ra khi 1 luồng có “nguy cơ” bị chặn vĩnh viễn khi các luồng khác thay phiên nhau lấy dc khóa và truy cập vào tài nguyên

+ Khi nhiều luồng gọi wait() trên 1 đối tượng, và khi gọi notify() trên đối tượng đó để đánh thức 1 thread, thì sẽ có nguy cơ 1 thread sẽ có nguy cơ không bao giờ dc đánh thức.

Assume there are 10,000 threads running. Among them, one thread is having priority=1 (lowest priority), and the remaining all threads are having a priority greater than 1 (priority > 1). In this case, the thread having priority=1 has to wait till the completion of all remaining threads having high priority.

class MyThread extends Thread {

public void run() {

String threadName = Thread.currentThread().getName();

System.out.println(threadName + " Started");

synchronized(MyThread.class) { **// lock**

**// doing some useful work**

try {

Thread.sleep(2000); // 2 sec

} catch (InterruptedException ie){}

}

System.out.println(threadName + " End");

}

}

public class Test {

public static void main(String[] args) {

System.out.println("Start of Main thread");

MyThread mt[] = new MyThread[10];

for (int i=0; i<mt.length; i++) {

mt[i] = new MyThread(); // create thread

mt[i].start();

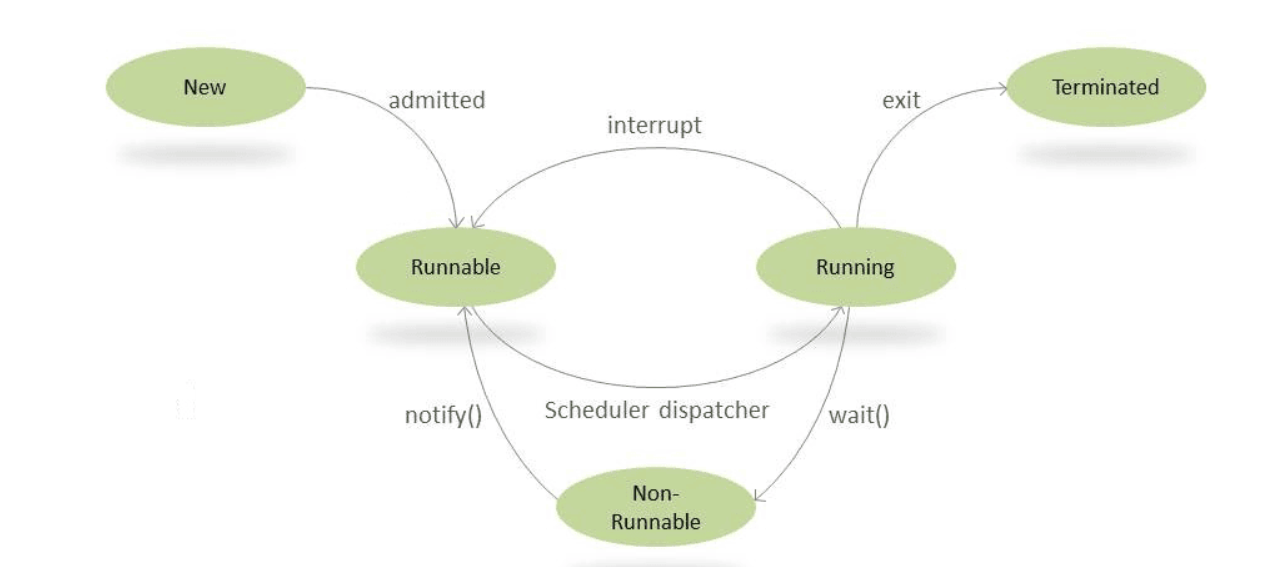
}

System.out.println("End of Main thread");

}

}

VI. Thread Signal wait(), notify(), notifyAll()



Đầu tiên, khi tạo 1 thread, nó ở trạng thái new(), sau khi start() thread sẽ vào trạng thái runable. thread sẽ không thực sự chạy ngay lập tức mà sẽ được đặt vào hàng chờ để lên lịch chạy bởi bộ lập lịch của hệ điều hành. Scheduler sẽ xác định xem luồng nào sẽ dc chạy và cấp phát CPU, sau đó thread sẽ chuyển sang trạng thái running. Khi gọi wait(), thread sẽ thoát khỏi monitor object hiện tai vào trạng thái waiting- chờ vĩnh viễn cho đến khi có 1 thread khác truy cập vào trên monitor object đó và call notify().hoặc nếu gọi wait(time) thì sẽ vào trạng thái time\_waiting. Sau khi gọi notify(),

Java có cơ chế cho phép luồng trở nên không hoạt động trong khi chờ một tín hiệu từ 1 luồng khác. Luồng gọi wait(), notify(), or notifyAll() phải có được lock tại đối tượng đó, nói cách khác việt gọi các hàm này phải dc tiến hành trong khối synchronized.

Wait() Nó yêu cầu luồng gọi từ bỏ lock và chuyển sang chế độ chờ cho đến khi một số luồng khác vào cùng một monitor object và gọi thông báo(). Phương thức wait() giải phóng khóa trước khi chờ và lấy lại khóa trước khi quay lại từ phương thức wait().

Notify() Nó đánh thức 1 luồng đang chờ tại cùng 1 monitor object. Nhưng khi nó gọi notify(), nó chưa lập tức giải phóng lock trên monitor object để các luồng đang chờ có thể truy cập vào mà phải cho đến khi nào cả khối đồng bộ hoàn thành công việc của nó thì nó ms nhả lock



Khi luồng đầu tiên gọi doWait() nó sẽ truy cập vào khối đồng bộ và sau đó gọi wait(), sau khi gọi wait() luồng đang gọi sẽ giải phóng lock và vào trạng thái chờ

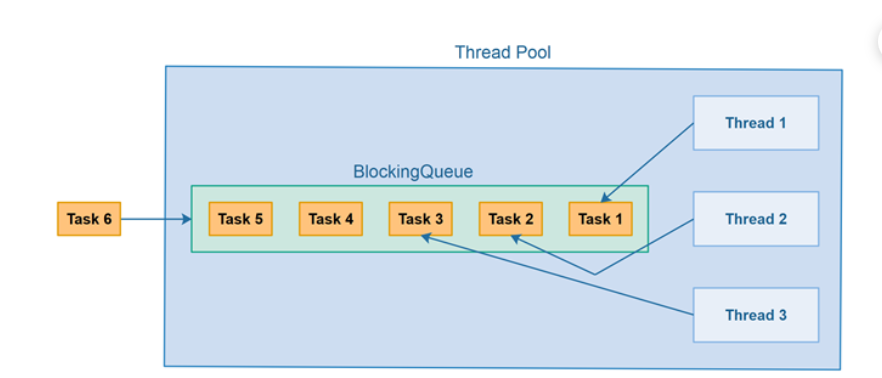
Khi luồng thứ 2 gọi doNotify() nó truy cập vào khối đồng bộ, trong đó nó sẽ gọi notify(), nó sẽ đánh thức luồng bị chờ do gọi wait() trên cùng 1 monitor object

1. ThreadPool

8.1 Khái niệm 1 threadpool là 1 nơi chứa các thread có thể tái sử dụng để thực hiện tác vụ, vì thế mỗi thread có thể thực hiện nhiều hơn 1 tác vụ. Thread pool là giải pháp thay thế cho việc tạo mới 1 thead cho mỗi tác vụ cần thực thi

? Tại sao cần sử dụng threadpool ? Việc tạo thêm mới thead đi kèm với nó là chi phí hiệu năng cao hơn so với việc sử dụng lại 1 luồng đã dc tạo. Bên cạnh đó là việc sử dụng threadpool làm cho việc kiểm soát số lượng các thread trở lên dễ dàng hơn. Nếu cứ tạo mới thread, mà mỗi thread sẽ tiêu thụ 1 lượng CPU nhất định để thực thi, việc thực thi nhiều thread sẽ dẫn tới chết máy.

8.2 ThreadPool hoạt động ntn?



Thay vì việc tạo mới 1 luồng cho mỗi tác vụ, các tác vụ có thể dc truyền vào 1 “BlockingQueue” . Và ngay khi Pool có 1 thread nào rảnh thì sẽ thực thi task đó

1. Một số hàm sử dụng vs thread

Sleep

Join()

Yield

setPriority

currrentThread()

getName()/setName()