

MULTITHREADING

Trình bày bởi:
Phạm Thanh Phúc

Ngày trình bày:
15/7/2024

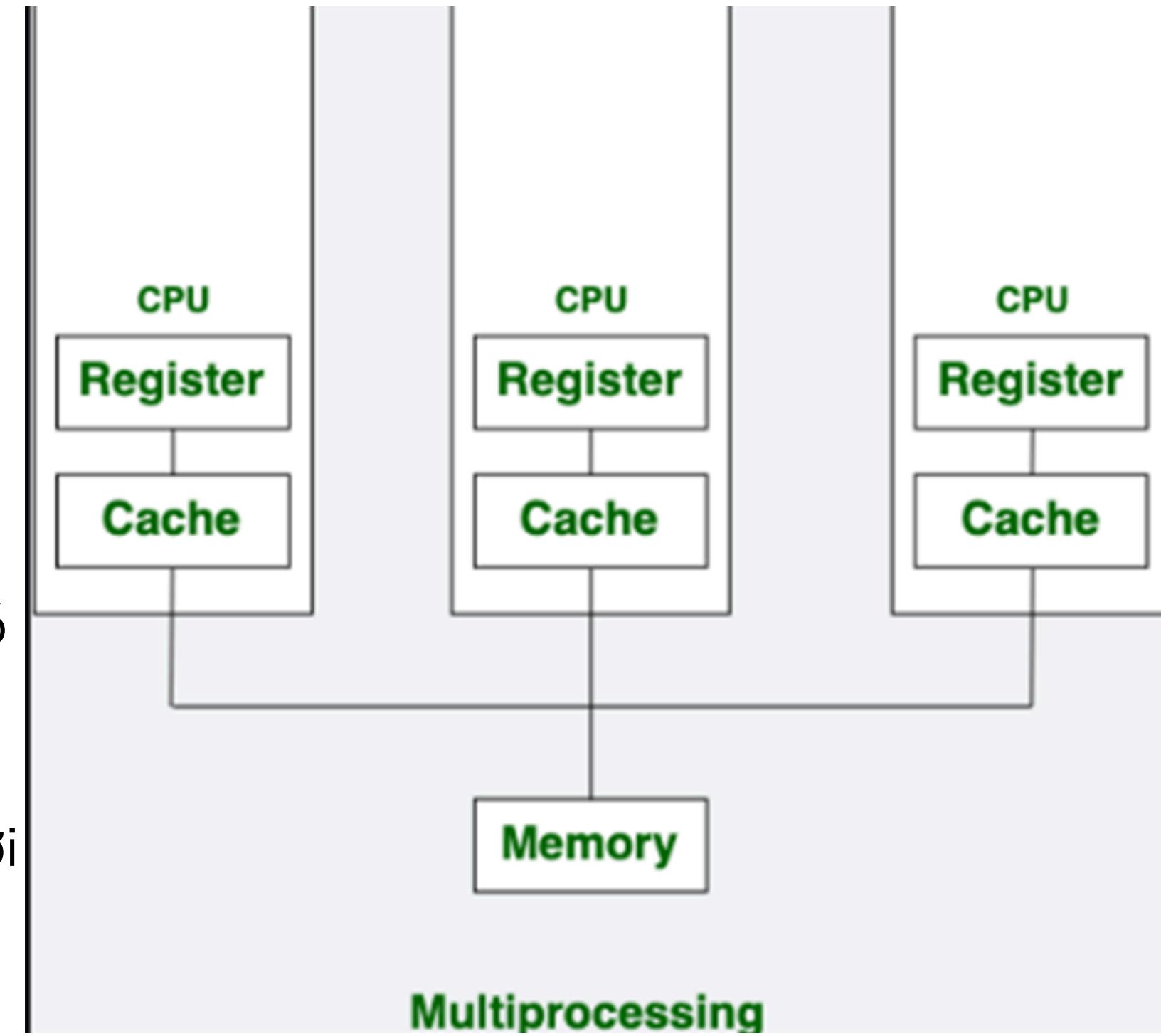
Tổng quan chủ đề

- I. Multitasking, Multiprocessing, Multithreading
- II. Thread Details
- III. Introduce concurrency and parallel
- IV. Race Condition
- V. Synchronized keyword
- VI. DeadLock
- VII. Starvation
- VIII. Thread Signal (`wait()`, `notify()`, `notifyAll()`)
- IX. ThreadPool

I.Multitasking, Multiprocessing,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

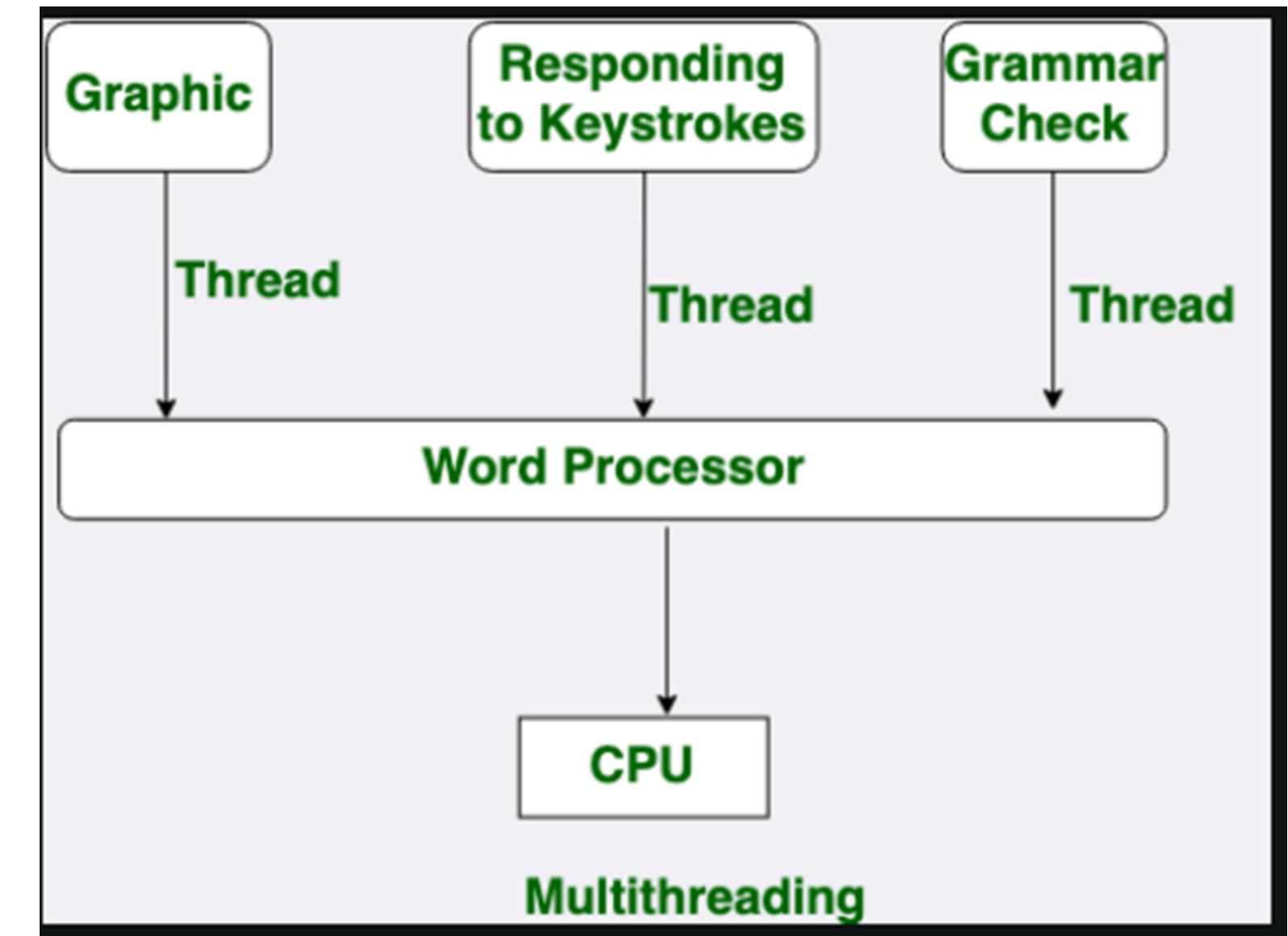


Vd : Khi click vào inteliji và skype để thực hiện công việc code và liên lạc => 2 process dc tạo ra, cta có thể trao đổi qua lại giữa 2 process để thực hiện công việc , nhưng dữ liệu của 2 bên không ảnh hưởng tới nhau.

Multithreading

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



So sánh Thread and Process

Thread	Process
Là 1 thành phần trong process	Process chứa 1 or nhiều thread
Có thể trao đổi dữ liệu giữa các process	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 phải sử dụng Multiple thread?

01

Viết một vấn đề
cụ thể của công ty
ở đây

Giải thích ngắn gọn vấn đề
được nêu.

02

Viết một vấn đề
cụ thể của công ty
ở đây

Giải thích ngắn gọn vấn đề
được nêu.

03

Viết một vấn đề
cụ thể của công ty
ở đây

Giải thích ngắn gọn vấn đề
được nêu.

Tại sao phải sử dụng multiple thread?

1. Sử dụng CPU tốt hơn

```
5 seconds reading file A  
2 seconds processing file A  
5 seconds reading file B  
2 seconds processing file B  
-----  
14 seconds total
```

Sử lý các tác vụ tuần tự

← Các tác vụ thực hiện xong thì tác vụ tiếp theo ms dc tiến hành

Vậy khi dùng đa luồng thì sao??

```
5 seconds reading file A  
5 seconds reading file B + 2 seconds processing file A  
2 seconds processing file B  
-----  
12 seconds total
```

2. 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.

_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



Một số 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

2. 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 switching) có thể gây chi phí cao.
- **Context switching** 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

Một 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.

1. Thread scheduler

Một thành phần của Java quyết định luồng nào sẽ chạy hoặc thực thi và luồng nào sẽ chờ. Nó dựa vào Priority và Time of arrival

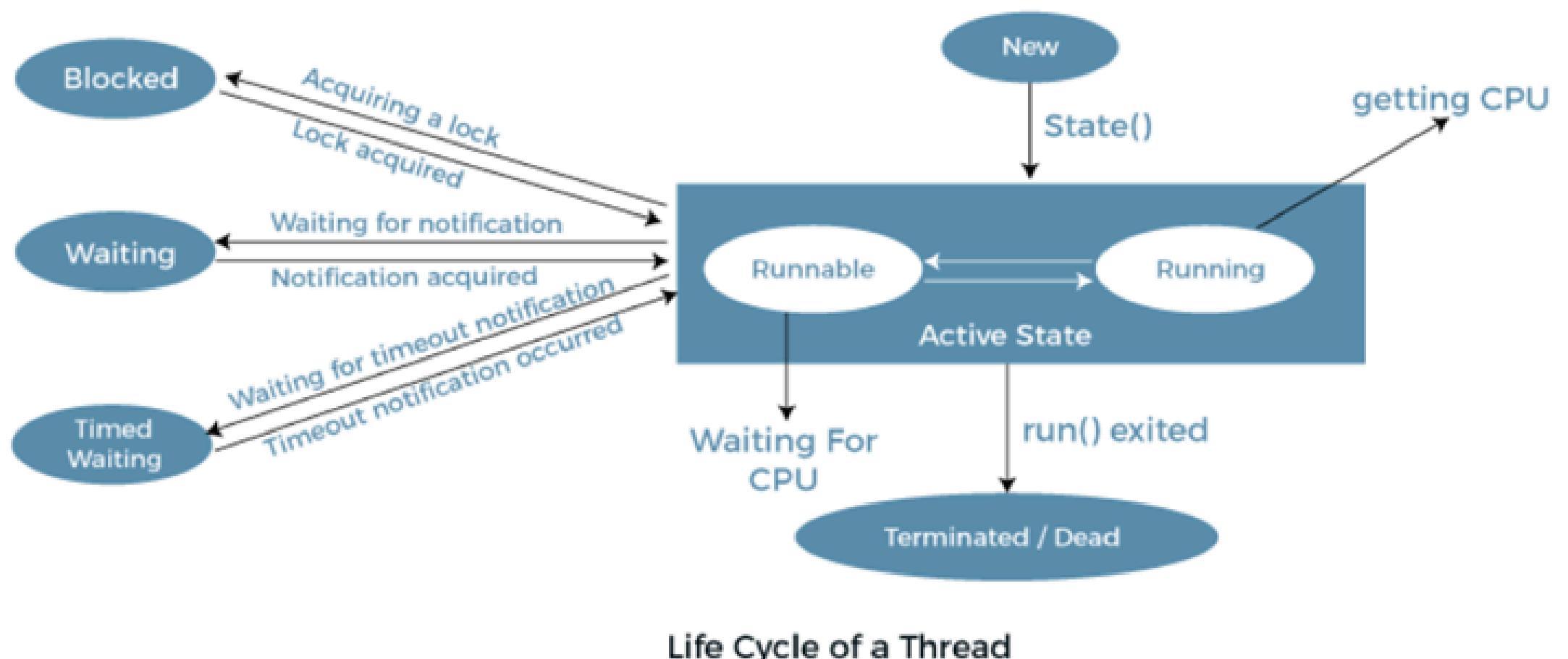
2. Priority

Mức độ ưu tiên của mỗi luồng nằm trong khoảng từ 1 đến 10. Nếu một luồng có mức độ ưu tiên cao hơn, điều đó có nghĩa là luồng đó có cơ hội được bộ lập lịch luồng chọn tốt hơn

3. Time of Arrival

Khi mà 2 thread có cùng độ ưu tiên, thì lúc này ko thể dựa vào độ ưu tiên để chọn thread mà sẽ dựa vào xem luồng nào tới trước sẽ được chọn trước

Life cycle of Thread



1. New state
2. Active state
3. Blocked or Waiting state
4. Timed Waiting
5. Terminated

ĐÁNH GIÁ NĂM VỪA QUA



1. New state

Khi 1 thread được tạo, nó ở trạng thái new. Các đoạn code chưa được thực thi khi ở trạng thái này



2. Active state

Khi 1 thread gọi phương thức start(). Sẽ chuyển từ trạng thái new => active. Trạng thái active bao gồm 2 trạng thái là Runnable và Running.
+ Runnable: Một thread sẵn sàng để chạy thì sẽ chuyển vào Runnable state
+ Running: Khi thread đạt dc CPU, nó chuyển từ trạng thái runnable sang running



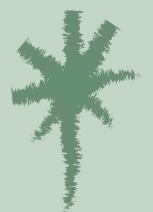
3. Blocked or Waiting state

Thread ,khi nó không hoạt động nó sẽ ở trạng thái blocked (nghĩa là khi 1 thread truy cập vào 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). Waiting là trạng thái thread này chờ thread khác và không có thời gian cụ thể.



4.Timed waiting

Nó tương tự ở trạng thái waiting nhưng sẽ được gọi lại để chạy sau một khoảng thời gian. có thể bằng cách sử dụng sleep()



5.Terminated

Thread kết thúc khi hoàn thành công việc của nó, hoặc gặp phải exception.

Tạo thread

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

1. Tạo thread sử dụng extend Thread Class

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

```
public class MyThread extends Thread {  
  
    public void run(){  
        System.out.println("MyThread running");  
    }  
}
```

```
MyThread myThread = new MyThread();  
myThread.start();
```

Tạo thread

2. Tạo thread bằng cách implementation Runnable interface

Tạo 1 class implemantation Runnable interface

```
public class MyRunnable implements Runnable {  
    public void run(){  
        System.out.println("MyRunnable running");  
    }  
}
```

Tạo 1 thểm hiện của class MyRunnable và truyền vào constructor của Thread

```
Runnable runnable = new MyRunnable();
```

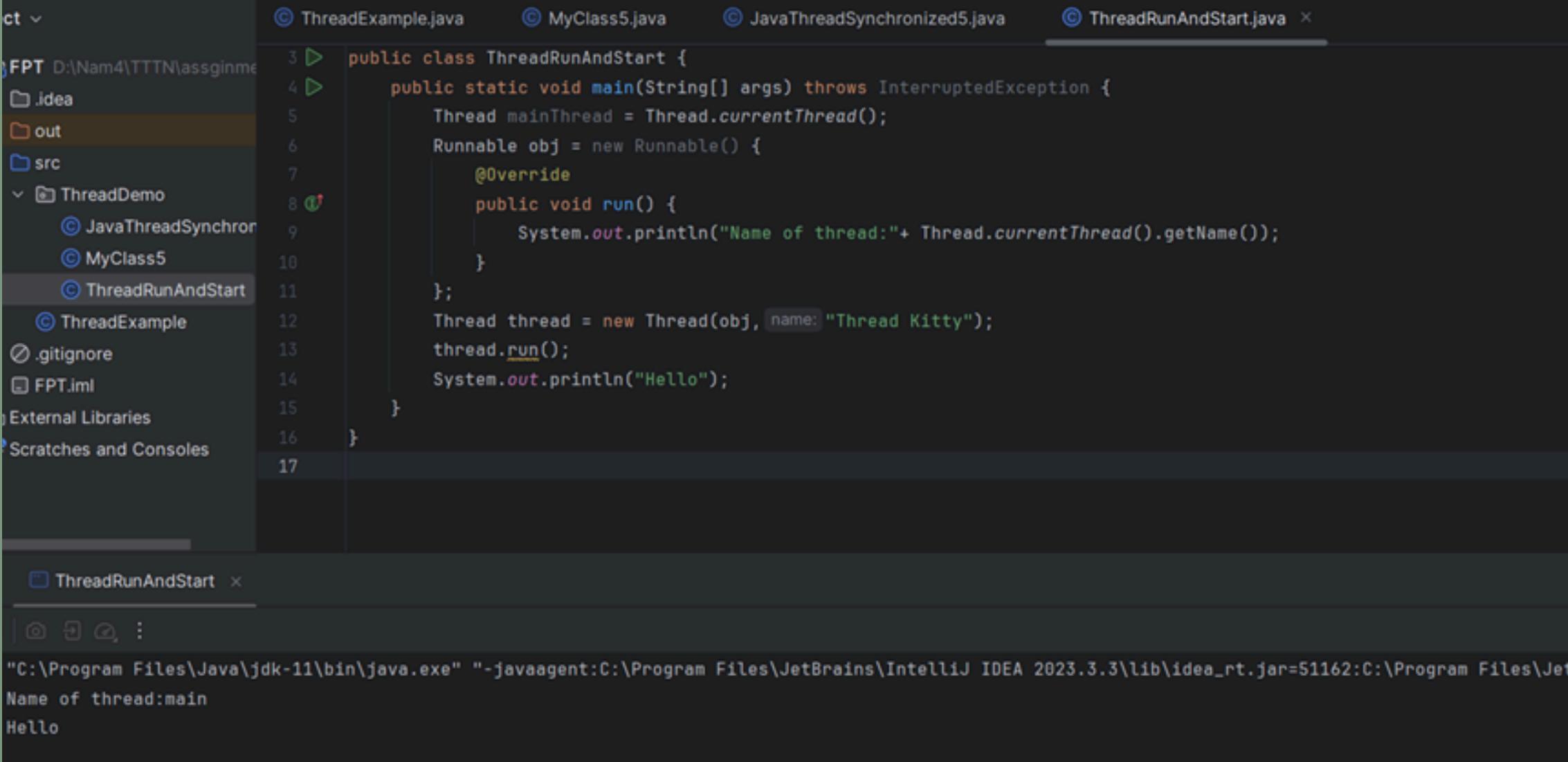
```
Thread thread = new Thread(runnable);  
thread.start();
```

Cũng có thể tạo bằng lambda
expression



```
Thread threadLambda = new Thread(  
    () -> {  
        System.out.println("TestLambda to create thread");  
    }  
);  
threadLambda.start();
```

Chú ý khi tạo thread

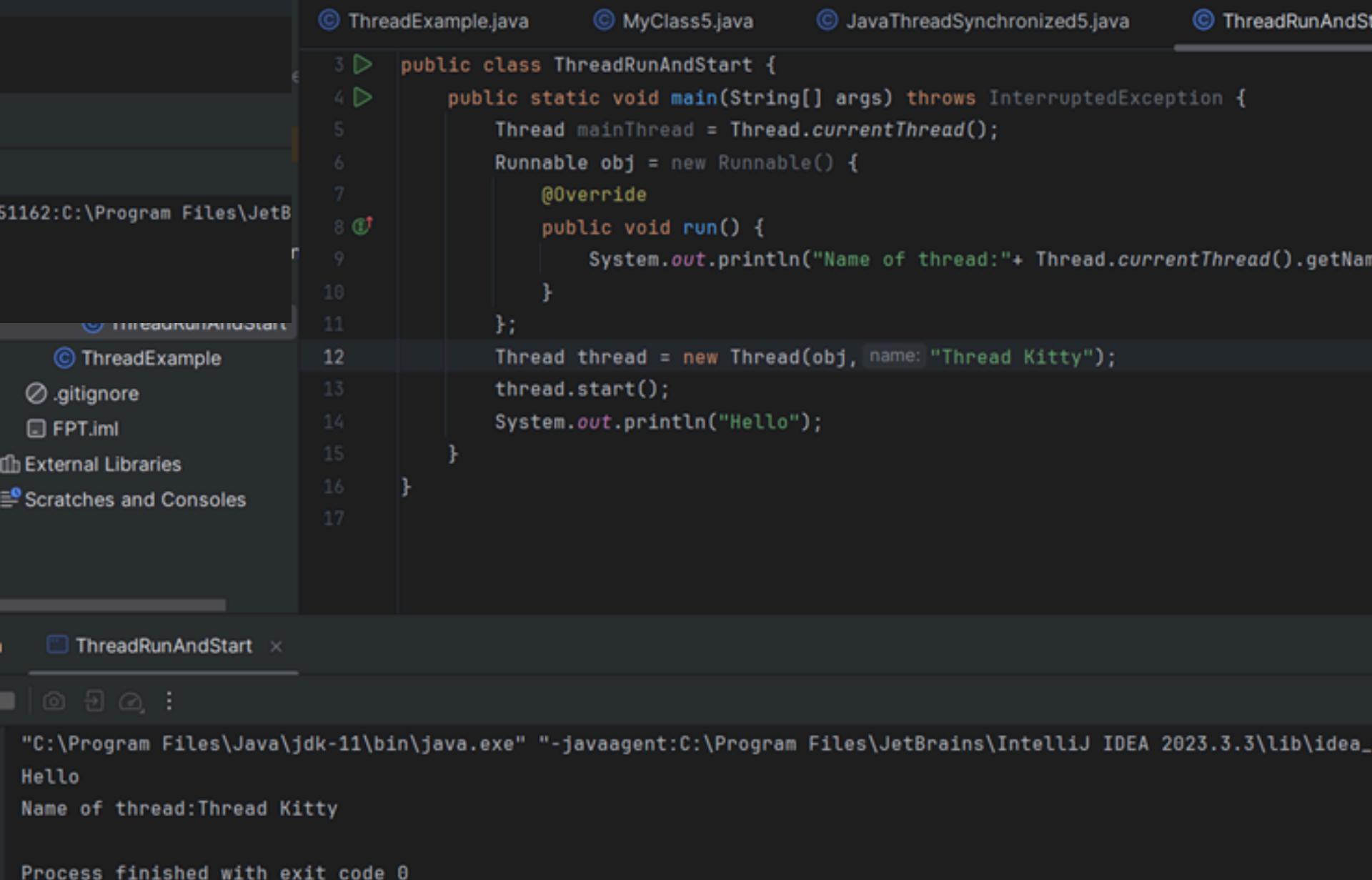


```
public class ThreadRunAndStart {
    public static void main(String[] args) throws InterruptedException {
        Thread mainThread = Thread.currentThread();
        Runnable obj = new Runnable() {
            @Override
            public void run() {
                System.out.println("Name of thread:" + Thread.currentThread().getName());
            }
        };
        Thread thread = new Thread(obj, name: "Thread Kitty");
        thread.run();
        System.out.println("Hello");
    }
}
```

"C:\Program Files\Java\jdk-11\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.3\lib\idea_rt.jar=51162:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.3\bin" -Dfile.encoding=UTF-8

Name of thread:main
Hello

Còn khi gọi start(), nó sẽ tạo ra 1 thread mới



```
public class ThreadRunAndStart {
    public static void main(String[] args) throws InterruptedException {
        Thread mainThread = Thread.currentThread();
        Runnable obj = new Runnable() {
            @Override
            public void run() {
                System.out.println("Name of thread:" + Thread.currentThread().getName());
            }
        };
        Thread thread = new Thread(obj, name: "Thread Kitty");
        thread.start();
        System.out.println("Hello");
    }
}
```

"C:\Program Files\Java\jdk-11\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.3\lib\idea_rt.jar=51162:C:\Program Files\JetBrains\IntelliJ IDEA 2023.3.3\bin" -Dfile.encoding=UTF-8

Hello
Name of thread:Thread Kitty

Process finished with exit code 0

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

Dừng 1 thread

Để dừng 1 thread, sẽ sử dụng flag để triển khai code để dừng 1 thread.

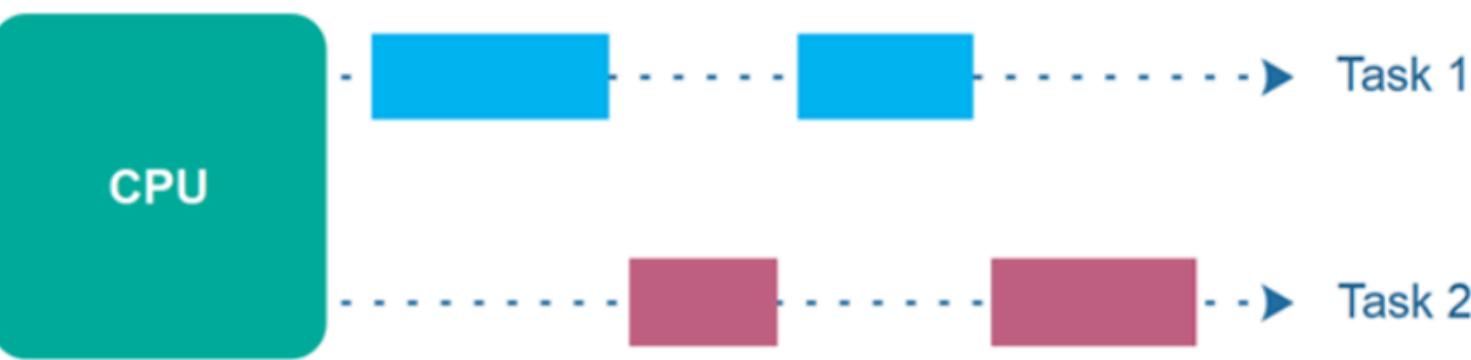
```
public class MyRunnable implements Runnable {  
    private boolean doStop = false;  
    public synchronized void doStop() {  
        this.doStop = true;  
    }  
    private synchronized boolean keepRunning() {  
        return this.doStop == false;  
    }  
    @Override  
    public void run() {  
        while(keepRunning()) {  
            // keep doing what this thread should do.  
            System.out.println("Running");  
            try {  
                Thread.sleep(3L * 1000L);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
public class MyRunnableMain {  
    public static void main(String[] args) {  
        MyRunnable myRunnable = new MyRunnable();  
        Thread thread = new Thread(myRunnable);  
        thread.start();  
        try {  
            Thread.sleep(10L * 1000L);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        myRunnable.doStop();  
    }  
}
```

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

III. Introduce concurrency and parallel

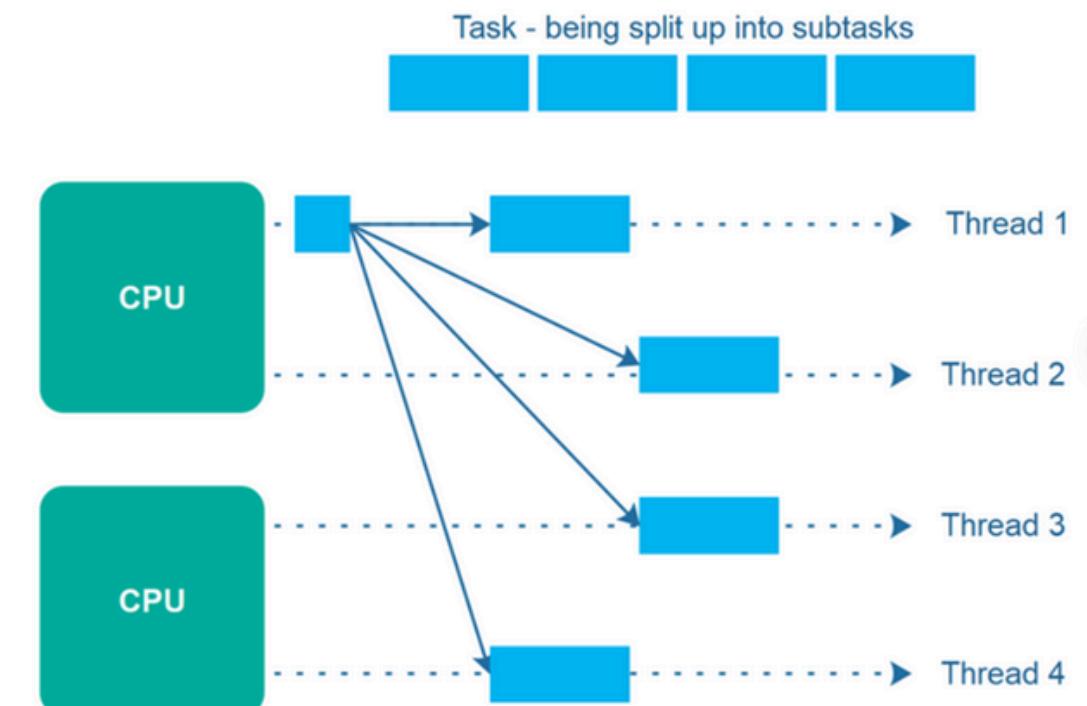
3.1. Concurrent execution



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.

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.

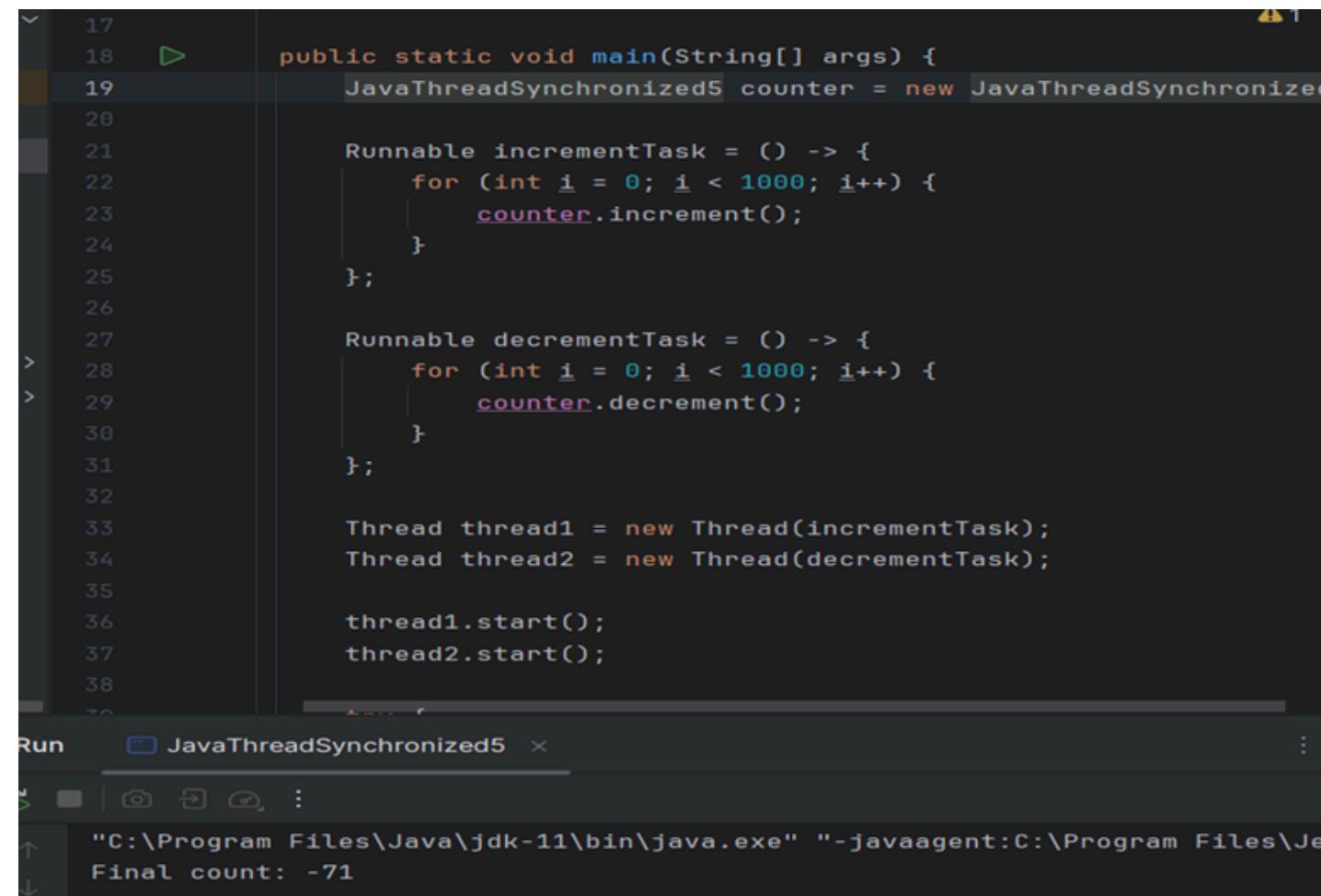


Race Condition

Khái niệm: **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. Check then act

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



The screenshot shows a Java code editor with a dark theme. The code is a Java program named 'JavaThreadSynchronized5'. It creates two threads, 'thread1' and 'thread2', each running a task to increment or decrement a shared counter. The output window at the bottom shows the final count as '-71', indicating a race condition where the final value is incorrect due to concurrent modifications.

```
17
18  public static void main(String[] args) {
19      JavaThreadSynchronized5 counter = new JavaThreadSynchronized5();
20
21      Runnable incrementTask = () -> {
22          for (int i = 0; i < 1000; i++) {
23              counter.increment();
24          }
25      };
26
27      Runnable decrementTask = () -> {
28          for (int i = 0; i < 1000; i++) {
29              counter.decrement();
30          }
31      };
32
33      Thread thread1 = new Thread(incrementTask);
34      Thread thread2 = new Thread(decrementTask);
35
36      thread1.start();
37      thread2.start();
38  }
```

Run JavaThreadSynchronized5

"C:\Program Files\Java\jdk-11\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\lib\javassist-agent.jar" -Dfile.encoding=UTF-8

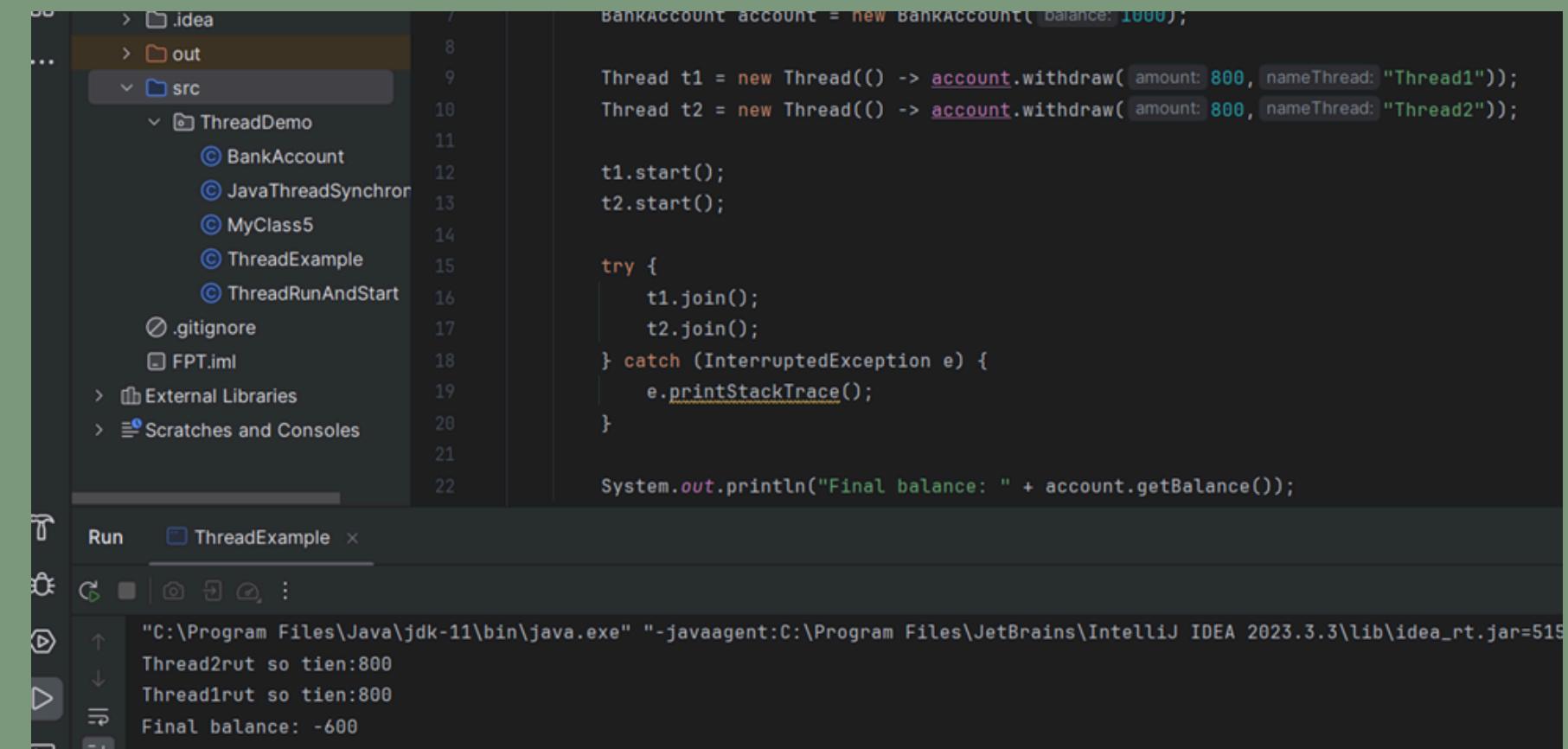
Final count: -71

Race Condition

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

```
9
10    2 usages
11
12    public void withdraw(int amount, String nameThread) {
13
14        if (balance >= amount) {
15
16            try {
17
18                Thread.sleep( millis: 1000 );
19
20            } catch (InterruptedException e) {
21
22                e.printStackTrace();
23
24            }
25
26            balance -= amount;
27
28            System.out.println(nameThread + " rut so tien:" + amount);
29
30        } else{
31
32            System.out.println("Balance doesn't enough");
33
34        }
35
36    }
```



The screenshot shows the IntelliJ IDEA interface. On the left, the project structure is visible with a file named 'ThreadExample.java' selected. The code in 'ThreadExample.java' demonstrates a race condition with two threads. It initializes a bank account with a balance of 1000, creates two threads (t1 and t2), and has each thread withdraw 800 units. Both threads print their withdrawal amounts to the console. Finally, it prints the final balance. The terminal window at the bottom shows the output: 'Thread2rut so tien:800', 'Thread1rut so tien:800', and 'Final balance: -600'. This indicates a race condition where both threads accessed and modified the shared balance variable simultaneously.

```
8
9
10    BankAccount account = new BankAccount( balance: 1000 );
11
12    Thread t1 = new Thread(() -> account.withdraw( amount: 800, nameThread: "Thread1"));
13    Thread t2 = new Thread(() -> account.withdraw( amount: 800, nameThread: "Thread2"));
14
15    t1.start();
16    t2.start();
17
18    try {
19        t1.join();
20        t2.join();
21    } catch (InterruptedException e) {
22        e.printStackTrace();
23
24    }
25
26    System.out.println("Final balance: " + account.getBalance());
```

Với example đồng thời rút tiền, 2 người đều check điều kiện và thỏa mãn điều kiện $balance > 800$, 2 luồng xảy ra đồng thời, và như thế khi người 1 rút tiền xong, thì $balance$ sẽ còn 200, nhưng điều kiện của người 2 vẫn thỏa mãn và vẫn sẽ rút tiếp và kết quả không mong muốn là -600

III. Java synchronized

Synchronized 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 method

```
public class MyCounter {  
    private int count = 0;  
  
    public synchronized void add(int value){  
        this.count += value;  
    }  
}
```

Một synchronized instance method trong Java được đồng bộ hóa trên 1 instance (object) sở hữu phương thức đó. Có nghĩa là, với mỗi object, phương thức được đồng bộ hóa trong object đó, tại một thời điểm chỉ có 1 thread execute.

2. Synchronized Static method

```
public static MyStaticCounter{  
    private static int count = 0;  
  
    public static synchronized void add(int value){  
        count += value;  
    }  
  
    public static synchronized void subtract(int value){  
        count -= value;  
    }  
}
```

Một synchronized static method trong Java được đồng bộ hóa trên 1 class sở hữu phương thức đó. Có nghĩa là, với mỗi **class**, phương thức được đồng bộ hóa trong **class** đó, tại một thời điểm chỉ có 1 thread dc excute.

3. Synchronized block

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 bên. 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

```
package com.gpcoder.sync.block;

public class ShareMemory {
    public void printData(String threadName) {
        // Do Something before synchronized ...
        synchronized (this) {
            for (int i = 1; i <= 5; i++) {
                System.out.println(threadName + ": " + i);
            }
        }
    }
}
```

3. Synchronized block with instance method

```
package com.gpcoder.sync.block;

public class ShareMemory {
    public void printData(String threadName) {
        // Do Something before synchronized ...
        synchronized (this) {
            for (int i = 1; i <= 5; i++) {
                System.out.println(threadName + ": " + i);
            }
        }
    }
}
```

Synchronized block sử dụng monitor object để đồng bộ hóa. Các đối tượng sử dụng trong ngoặc của khối đồng bộ được gọi là monitor object , ở đây sử dụng từ khóa this. Và chỉ có 1 luồng có thể thực hiện trên 1 object monitor.

Monitor object có thể là bất cứ object nào, không nhất thiết sử dụng this để làm monitor object

4. Synchronized block with static method

```
public class MyClass {  
  
    public static synchronized void log1(String msg1, String msg2){  
        log.writeln(msg1);  
        log.writeln(msg2);  
    }  
  
    public static void log2(String msg1, String msg2){  
        synchronized(MyClass.class){  
            log.writeln(msg1);  
            log.writeln(msg2);  
        }  
    }  
}
```

Tương tự như với instance method, synchronized block với static method thay vì sử dụng object, nó sẽ sử dụng class làm monitor

Xét ví dụ :

```
public class Counter{  
    long count = 0;  
  
    public synchronized void add(long value){  
        this.count += value;  
    }  
}
```

```
public class CounterThread extends Thread{  
    protected Counter counter = null;  
  
    public CounterThread(Counter counter){  
        this.counter = counter;  
    }  
  
    public void run() {  
        for(int i=0; i<10; i++){  
            counter.add(i);  
        }  
    }  
}
```

```
public class Example {  
  
    public static void main(String[] args){  
        Counter counter = new Counter();  
        Thread threadA = new CounterThread(counter);  
        Thread threadB = new CounterThread(counter);  
  
        threadA.start();  
        threadB.start();  
    }  
}
```

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 tạo ra 2 instance của class Counter:

```
public class Example {  
  
    public static void main(String[] args){  
        Counter counterA = new Counter();  
        Counter counterB = new Counter();  
        Thread threadA = new CounterThread(counterA);  
        Thread threadB = new CounterThread(counterB);  
  
        threadA.start();  
        threadB.start();  
    }  
}
```

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 instance của riêng nó, và kết quả của 2 luồng (count của counterA sẽ bằng (1 + .. + 10), count của counterB sẽ = (1 + .. + 10)).

VI. DeadLock

Figure - 1

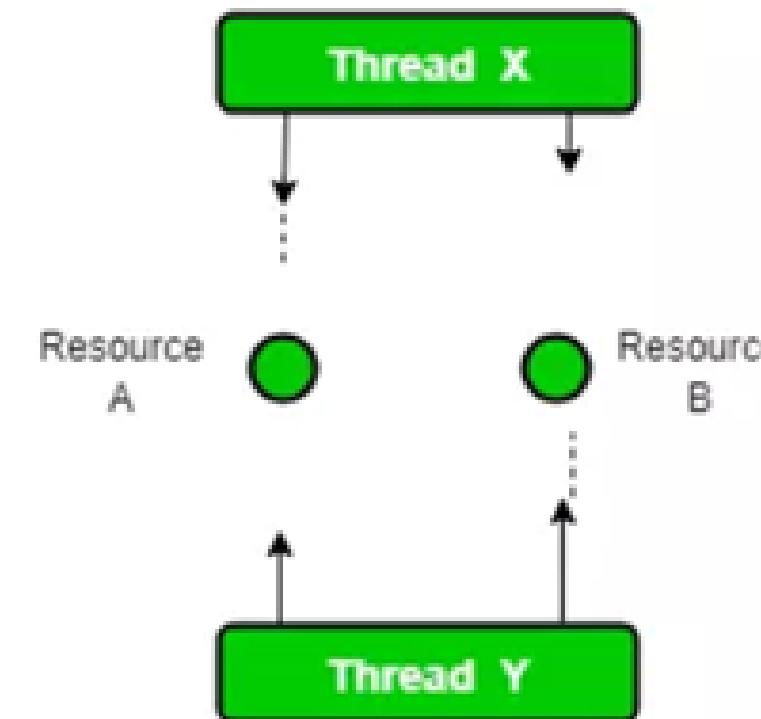
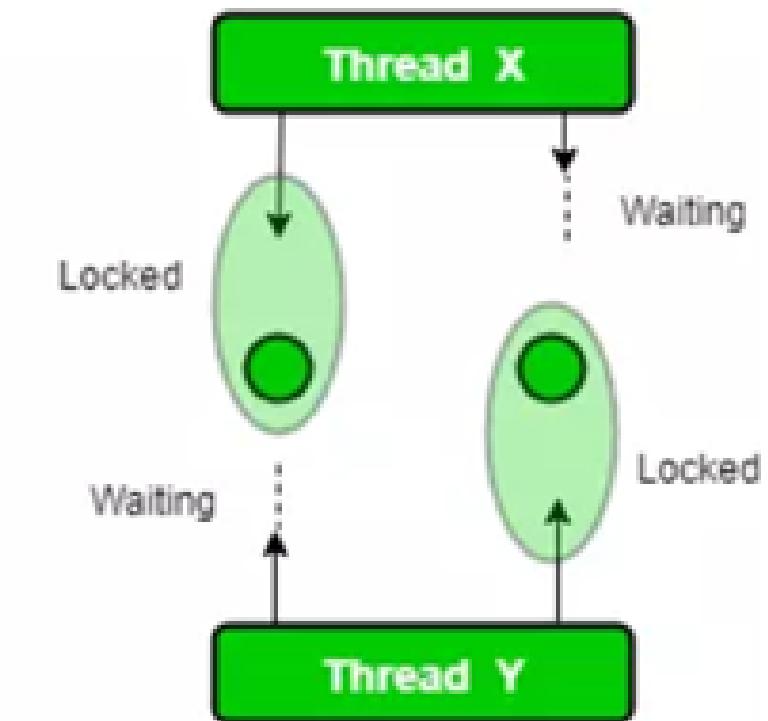


Figure - 2



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 dc khóa của đối phương**

```

5 ▶ public class DeadLockTest {
6 ▶   public static void main(String[] args) throws InterruptedException {
7     Object lockPhu = new Object();
8     Object lockPhuc = new Object();
9     final String lan = "Lan";
10    // t1 tries to lock resource1 then resource2
11    Thread t1 = new Thread() {
12      @Override
13      public void run() {
14        synchronized (lockPhu) {
15          System.out.println("Phú đến nhà, rủ Phúc đi chơi");
16          System.out.println("waiting.....");
17          try { Thread.sleep( millis: 1000); } catch (Exception e) {}
18
19          synchronized (lockPhuc) {
20            System.out.println("đi chơi");
21          }
22        }
23      };

```

Luồng 1 chiếm `lockPhu` thực hiện hành động, nghỉ 1s, sau đó nó muốn chiếm `lockPhuc` để chạy tiếp.

Đồng thời, Luồng 2 chiếm `lockPhuc`, thực hiện, nghỉ 1s, sau đó muốn chiếm `lockPhu` nhưng lúc này 2 luồng đều chưa nhả khóa, vì thế đều đợi nhau

The screenshot shows the IntelliJ IDEA interface with the code editor and a terminal window. The code editor displays the `DeadLockTest` class with its implementation. The terminal window shows the execution results of the program.

```

25
26
27 @Override
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
// t2 tries to lock resource2 then resource1
Thread t2 = new Thread() {
  public void run() {
    synchronized (lockPhuc) {
      System.out.println("Phúc đến nhà Phú, rủ Phú đi học");
      System.out.println("waiting.....");
      try { Thread.sleep( millis: 1000); } catch (Exception e) {}

      synchronized (lockPhu) {
        System.out.println("đi học");
      }
    }
  }
};

t1.start();
// t1.join();
t2.start();
}


```

The terminal output shows:

```

Phú đến nhà, rủ Phúc đi chơi
waiting.....
Phúc đến nhà Phú, rủ Phú đi học
waiting.....

```

VII. 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

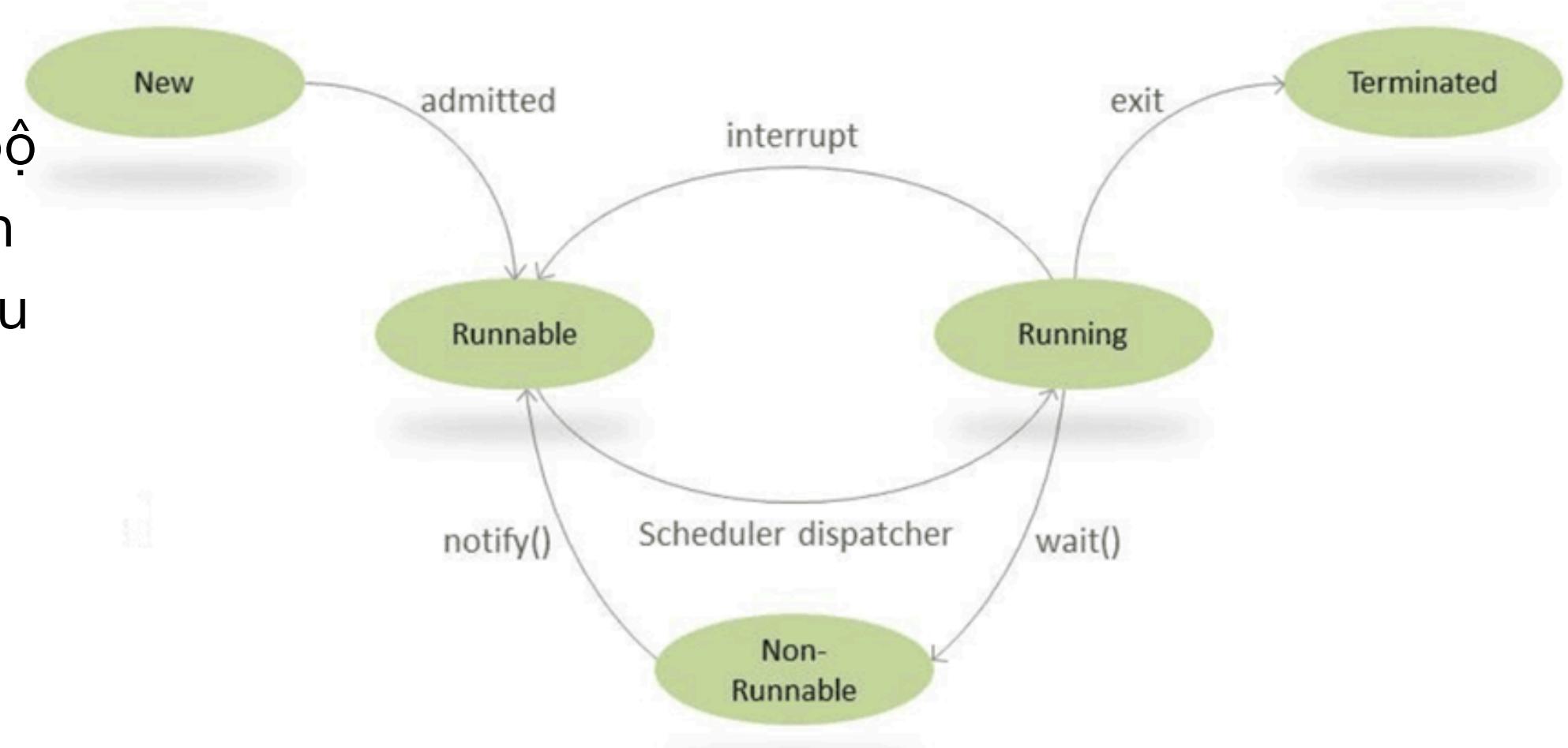
Có thể xảy ra khi 1 luồng có “nguy cơ” bị chặn vĩnh viễn khi các luồng khác thay phiên nhau lấy dc khóa và truy cập vào tài nguyên

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.

VIII. Thread Signal -wait(), notify(), notifyAll()

Đầu tiên, khi tạo 1 thread, nó ở trạng thái new(), sau khi start() thread sẽ vào trạng thái runnable. 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 tại vào trạng thái waiting- chờ vĩnh viễn cho đến khi có 1 thread khác truy cập vào trên monitor object đó và call notify().



Ví dụ

```
public class MonitorObject{  
}  
  
public class MyWaitNotify{  
  
    MonitorObject myMonitorObject = new MonitorObject();  
  
    public void doWait(){  
        synchronized(myMonitorObject){  
            try{  
                myMonitorObject.wait();  
            } catch(InterruptedException e){...}  
        }  
    }  
  
    public void doNotify(){  
        synchronized(myMonitorObject){  
            myMonitorObject.notify();  
        }  
    }  
}
```

- 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

*Lưu ý rằng, khi gọi notify(), luồng không lập tức thoát khỏi monitor object, mà tất cả các hành động trong khối block được thực hiện xong thì nó mới thoát khỏi để cho luồng khác truy cập

IX.ThreadPool

8.1 Khái niệm

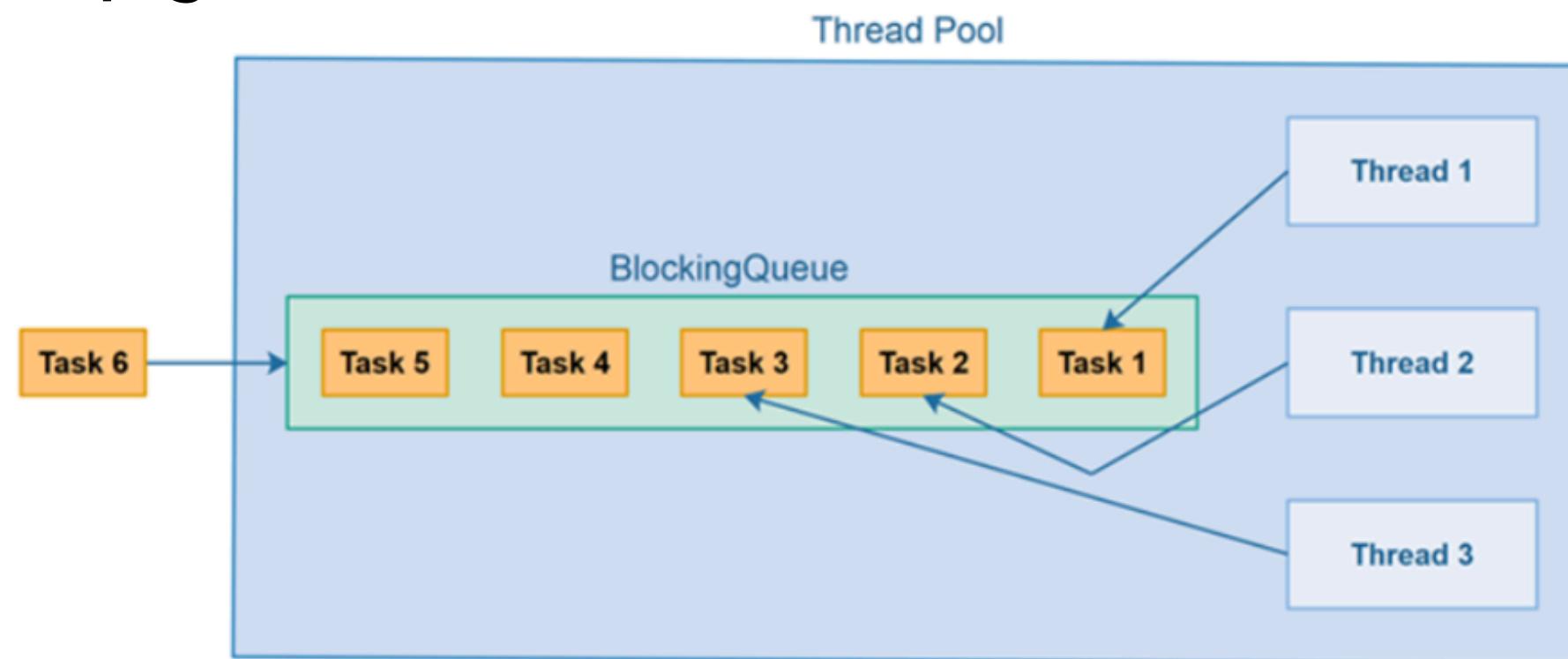
1 threadpool là 1 nơi chứa các thread có thể tái sử dụng để thực hiện tác vụ, vì thế mỗi thread có thể thực hiện nhiều hơn 1 tác vụ. Thread pool là giải pháp thay thế cho việc tạo mới 1 thread cho mỗi tác vụ cần thực thi

Tại sao cần sử dụng threadpool ?

Việc tạo thêm mới thread 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.

IX.ThreadPool

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. Trong threadpool, các thread sau khi hoàn thành nhiệm vụ nó sẽ không “chết”, mà ở vào trạng thái không hoạt động, và chờ đợi để được lựa chọn thực hiện một task khác

**THANKS
FOR
WATCHING**

Question
&
Answer