BITTIGER

JAVA MULTITHREADING_1

Outline

- Thread vs Process
- Concurrency vs Parallelism
- Runnable vs Thread
- Life Cycle
- Synchronized
- Lock

Thread VS Process

- Process

- A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

- Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes.

- the Java virtual machine run as a single process.

Thread VS Process

- Threads
- Threads are sometimes called lightweight processes. Creating a new thread requires fewer resources than creating a new process.
- Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files.

Conclusion

- Process contains threads. Thread is lightweight.
- Processes are independent of each other. Threads share common resources
- e.g.: JVM——> Process Main Method ——> Thread

Concurrency vs Parallelism

- Concurrency: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

- Parallelism: A condition that arises when at least two threads are executing simultaneously.

How to Create a thread in Java

- Instantiate a subclass which extends the `Thread` class.
  - Extend the `java.lang.Thread` class.
- `new` the Thread constructor using a subclass which implements the `Runnable` interface.
  - Implement the `java.lang.Runnable` interface.

```
2  class MyThread extends Thread{
3    @Override
4    public void run(){
5
6    }
7
8  }
9
10 class MyRunnable implements Runnable{
11   @Override
12   public void run(){
13
14   }
15 }
16
17 public class Solution{
18   public static void main(String[] args){
19     MyThread thread1 = new MyThread();
20     Thread thread2 = new Thread(new MyRunnable());
21     thread1.start();
22     thread2.start();
23   }
24 }
```

line 2 : extends Thread

line 10 : implements Runnable

line 19 : new MyThread Object
line 20 : new Thread Object

line 21-22 : start thread.

run() vs start()

- `public void run()`: no new thread will be created.
  - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
- `public void start()`: a new thread will be created.
  - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread implicitly.
  - The result is that two threads are running concurrently:
    - the current thread – which returns from the call to the start method.
    - and the new created thread – which executes its run method.

Runnable vs Thread

- `java.lang.Runnable` - interface
  - `run()`
  - The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must (override) define a method of no arguments called run.
- This interface is designed to provide a common protocol for objects that wish to execute code while they are active.

Runnable vs Thread

- `java.lang.Runnable` - interface
- `java.lang.Object` - class
  - `java.lang.Thread` - class
    - Fields:
      - `static int MIN_PRIORITY/NORM_PRIORITY/MAX_PRIORITY`
    - Constructors: allocates a new Thread object.
      - `public Thread() / Thread(Runnable target) / Thread(Runnable target, String name)`
    - Methods:
      - `run(), start(), sleep(), yield(), join(), isAlive(), wait(), notify(), notifyAll(), setPriority(), getPriority(), setName(), getName(), interrupt().`

Runnable vs Thread

- Similarities:
  - The subclass should override the `run()` method.
  - The functionality that expected to be executed concurrently should be implemented in the `run()` method.

Runnable vs Thread

- Differences:
  - Extend the `Thread` class will make your class unable to extend other classes.
  - Implement the `Runnable` interface provides better object-oriented design and consistency and also avoid the single inheritance problems.
  - In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods.

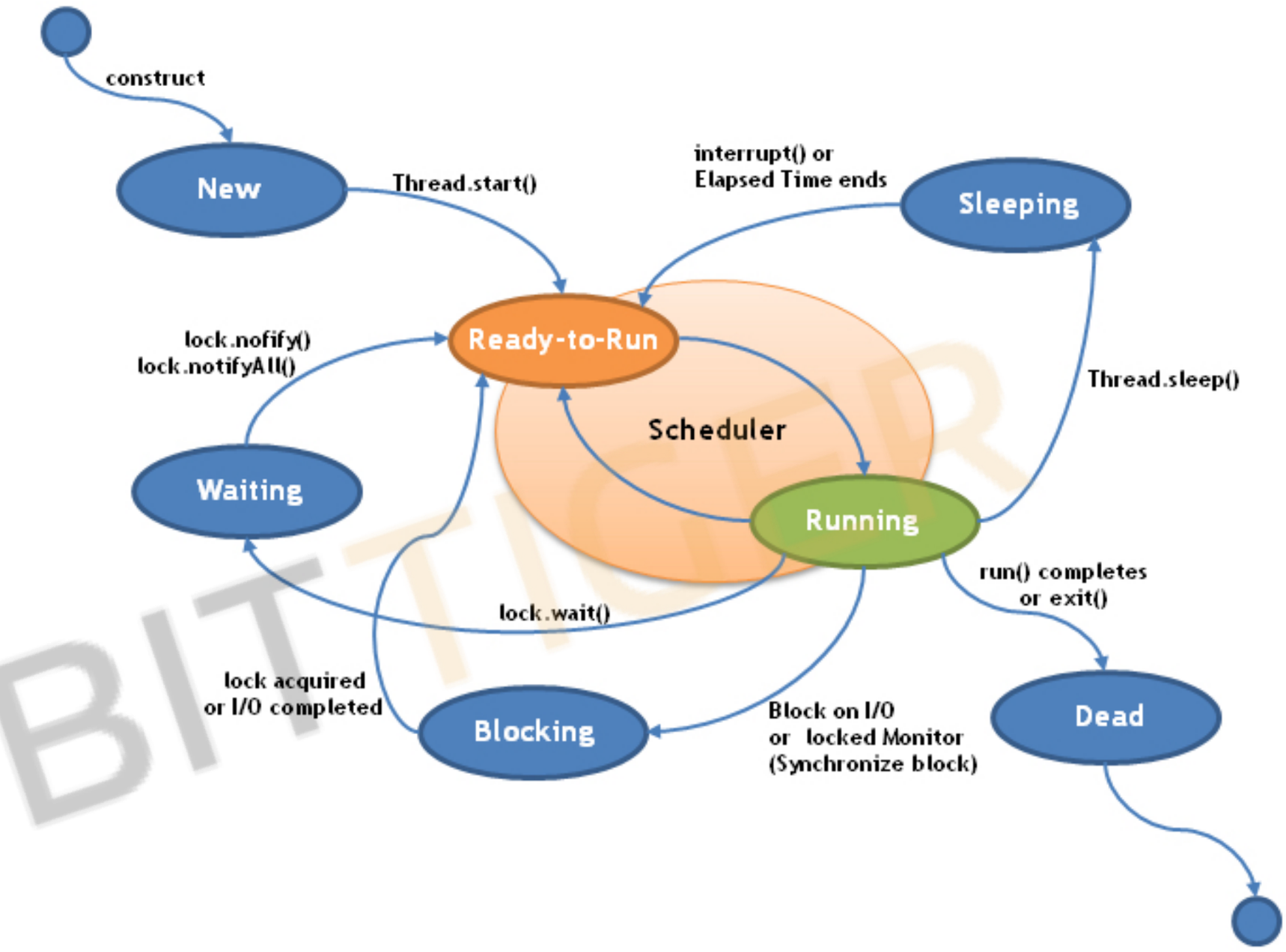
Thread priority

- Thread priorities are the integers which decide how one thread should be treated with respect to other threads.
- Thread priority decides when to switch from one running thread to another, this process is called context switching.
- When a thread is created, it inherits its priority from the thread that created it.

Thread priority

- A thread can voluntarily release control, then the highest priority thread that is ready to run is given to the CPU.
  - `yield()`: used to give other threads, which has the same priority as the invoked thread, a chance to execute, and the current running thread changes to runnable state.
- A thread's priority can be modified at any time after its creation using the `setPriority()` method.
  - Thread priorities are integers ranging between `MIN_PRIORITY (1)` and `MAX_PRIORITY (10)`. Normally the thread priority is `NORM_PRIORITY (5)`.

thread Life cycle



join() & sleep()

- `public final void join(long millis, int nanos) throws InterruptedException`
  - The `join()` method is used when we want the parent thread waiting until the children threads which invoke `join()` end.

- `public static void sleep(long millis, int nanos) throws InterruptedException`
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

```
26  class MyThread extends Thread{
27    @Override
28    public void run() {
29      for(int i = 0; i < 10; i++){
30        System.out.println(i);
31        try{
32          Thread.sleep(200);
33        } catch (InterruptedException e){
34          e.printStackTrace();
35        }
36      }
37    }
38  }
```

line 32 : Thread.sleep()

```java
public class Solution {
    public static void main(String[] args){
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        thread1.start();

        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        thread2.start();

    }
}
```

thread1.join()

```
/Library/Java/JavaVir
0
1
2
3
4
5
6
7
8
9
0
1
2
3
4
5
6
7
8
9
```

```
/Library/Java/Java
0
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
```

join():
Parent thread: main thread
child thread: thread1, thread2

no join() method.

wait(), notify(), notifyAll()

- `public final void notify()` wakes up the first thread that invoked `wait()` on the same object and changes the thread to the ready state.
- `public final void notifyAll()` wakes up ALL the threads that invoked `wait()` on the same object and changes the thread to the ready state. The highest priority thread will run first.
- public final void wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Deprecated

通常要配合一个boolean 标志位使用来避免pre-notify

Synchronized

- When two or more threads need access to the same resource at same time, there should be some way that the resource is only available for one thread at one time. Synchronization helps to achieve this goal.
- used for method and object

```
58  class Demo{
59    public static synchronized void getClassValue() {
60      // Class Lock
61    }
62    public synchronized void setNewValue() {
63      // Object Lock
64    }
65    public synchronized void getNewValue() {
66      // Object Lock
67    }
68    public void getNormalValue() {
69      // None Lock
70    }
71  }
```

line 59 : static synchronized method (类锁)

line 62,65 : synchronized method (对象锁)

line 68 : normal method

Deadlock

- Whenever there are multithreads content for exclusive access to multiple locks, there is a possibility of deadlock.
- A set of threads is said to be deadlocked when EACH thread is waiting for an action that can ONLY performed by other threads.
- It describes a situation where two or more threads are blocked forever, and waiting for each other.
- Deadlock occurs when multiple threads need the same locks but obtain them in different order.

```
2  public class Solution {
3      static Object lock1 = new Object();
4      static Object lock2 = new Object();
5
6      static class ThreadDemo1 extends Thread {
7          public void run() {
8              // Synchronized Block
9              synchronized (lock1) {
10                 try{
11                     Thread.sleep(100);
12                 } catch (InterruptedException e){
13                 }
14                 synchronized (lock2){
15                     System.out.println("ThreadDemo1 !");
16                 }
17             }
18         }
19     }
20     static class ThreadDemo2 extends Thread {
21         public void run() {
22             synchronized (lock2) {
23                 try {
24                     Thread.sleep(100);
25                 } catch (InterruptedException e){
26                 }
27                 synchronized (lock1){
28                     System.out.println("ThreadDemo2 !");
29                 }
30             }
31
32         }
33     }
```

line 3-4 : Lock

line 9: synchronized Lock1

line 14 : synchronized Lock2

line 22 : synchronized Lock2

line 27 : synchronized Lock1

```
35      public static void main(String[] args) {
36          ThreadDemo1 t1 = new ThreadDemo1();
37          ThreadDemo2 t2 = new ThreadDemo2();
38          t1.start();
39          t2.start();
40      }
```

line 38-39 : start threads and dead lock happens

How to avoid deadlock

- Keep synchronized blocks short.
  - As short as possible while still protecting the integrity of shared data operations.

- Don’t invoke methods of other objects while holding a lock.(no nested lock)

Producer and consumer

- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.
- The producer's job is to generate data, put it into the buffer, and start again.
- the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

```
42  class Token {
43      private int val;
44      private String name;
45      public Token(int val, String name) {
46          this.val = val;
47          this.name = name;
48      }
49      @Override
50      public String toString(){
51          return this.name + "  " + this.val;
52      }
53  }
```

line 42: define Token class

```
54  class Bucket{
55      private BlockingQueue<Token> que;
56      private int rate;
57      public Bucket(int size, int rate){
58          this.que = new ArrayBlockingQueue<Token>(size);
59          this.rate = rate;
60      }
61      public void putToken(Token token){
62          try{
63              Thread.sleep(this.rate);
64              que.put(token);
65          } catch (InterruptedException e){
66              e.printStackTrace();
67          }
68      }
69      public Token getToken(){
70          try{
71              Thread.sleep(100);
72              return que.take();
73          } catch (InterruptedException e){
74              e.printStackTrace();
75              return null;
76          }
77      }
78  }
```

line 55: BlockingQueue

line 61: put method

line 64: que.put()

line 69: get method

line 72: que.take()

```
81  class PutExcuter implements Runnable{
82      private Bucket bucket;
83      public PutExcuter(Bucket bucket){
84          this.bucket = bucket;
85      }
86      @Override
87      public void run(){
88          int counter = 0;
89          while(counter < 10){
90              bucket.putToken(new Token(counter, Thread.currentThread().getName()));
91              System.out.println("PUT:  " + Thread.currentThread().getName() + " " + counter);
92              counter++;
93          }
94      }
95  }
96  class GetExecuter implements Runnable{
97      private Bucket bucket;
98      private int num;
99      public GetExecuter(Bucket bucket, int num){
100         this.bucket = bucket;
101         this.num = num;
102     }
103     @Override
104     public void run(){
105         int counter = 0;
106         while(counter < this.num){
107             System.out.println("GET:  " + Thread.currentThread().getName() + " From " + bucket.getToken());
108             counter++;
109         }
110     }
111 }
```

```java
public static void main(String[] args){
    Bucket bucket = new Bucket(20, 200);
    PutExecuter putExcuter = new PutExecuter(bucket);
    GetExecuter getExecuter = new GetExecuter(bucket, 6);

    List<Thread> putThread = new ArrayList<>();
    for(int i = 0; i < 2; i++){
        putThread.add(new Thread(putExcuter));
    }

    List<Thread> getThread = new ArrayList<>();
    for(int i = 0; i < 3; i++){
        getThread.add(new Thread(getExecuter));
    }

    for(Thread thread : putThread){
        thread.start();
    }

    for(Thread thread : getThread) {
        thread.start();
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/C
GET:  Thread-4 From Thread-1  0
PUT:  Thread-0 0
GET:  Thread-2 From Thread-0  0
PUT:  Thread-1 0
PUT:  Thread-0 1
GET:  Thread-3 From Thread-0  1
PUT:  Thread-1 1
GET:  Thread-4 From Thread-1  1
PUT:  Thread-0 2
GET:  Thread-3 From Thread-1  2
GET:  Thread-2 From Thread-0  2
PUT:  Thread-1 2
PUT:  Thread-1 3
PUT:  Thread-0 3
GET:  Thread-2 From Thread-0  3
GET:  Thread-4 From Thread-1  3
PUT:  Thread-1 4
PUT:  Thread-0 4
GET:  Thread-3 From Thread-1  4
GET:  Thread-2 From Thread-0  4
PUT:  Thread-1 5
GET:  Thread-4 From Thread-1  5
PUT:  Thread-0 5
GET:  Thread-3 From Thread-0  5
PUT:  Thread-1 6
GET:  Thread-2 From Thread-1  6
PUT:  Thread-0 6
GET:  Thread-4 From Thread-0  6
PUT:  Thread-1 7
GET:  Thread-3 From Thread-1  7
PUT:  Thread-0 7
GET:  Thread-2 From Thread-0  7
PUT:  Thread-1 8
GET:  Thread-4 From Thread-1  8
PUT:  Thread-0 8
GET:  Thread-3 From Thread-0  8
PUT:  Thread-1 9
PUT:  Thread-0 9
```

line 115: putThread List　　line 129: putThread start

line 116: getThread List　　line 133: getThread start

Lock and Condition Interface

- Lock Interface : ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

- A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular Lock instance use its newCondition() method.

```
2  static class Bucket{
3          private Lock lock1 = new ReentrantLock();
4          private Condition condition1 = lock1.newCondition();
5          private Condition condition2 = lock1.newCondition();
6
7          private Deque<String> bucket;
8          private int size;
9          private int rate;
10
11         public Bucket(int size, int rate){
12             bucket = new ArrayDeque<>();
13             this.size = size;
14             this.rate = rate;
15         }
```

line 3: define Lock as ReentrantLock

line 4 - 5: get corresponding condition from Lock

```
public void set(String val){
        lock1.lock();
        try{
            while(bucket.size() == this.size){
                condition1.await();
            }
            this.bucket.addLast(val);
            System.out.println("PUT:  " + val);
            Thread.sleep(this.rate);
            condition2.signalAll();
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock1.unlock();
        }

    }
    public void get(){
        lock1.lock();
        try{
            while(bucket.isEmpty()){
                condition2.await();
            }
            Thread.sleep(200);
            System.out.println("GET:  " + this.bucket.removeFirst());
            condition1.signalAll();
        }catch(InterruptedException e){
            e.printStackTrace();
        }finally {
            lock1.unlock();
        }
    }
}
```

```
185 class Producer implements Runnable{
186     private Bucket bucket;
187     public Producer(Bucket bucket){
188         this.bucket = bucket;
189     }
190     @Override
191     public void run(){
192         for(int i = 0; i < 10; i++){
193             this.bucket.set("Token----   " + i);
194         }
195     }
196 }
197 class Consumer implements Runnable{
198     private Bucket bucket = null;
199     public Consumer(Bucket bucket){
200         this.bucket = bucket;
201     }
202     @Override
203     public void run(){
204         for(int i = 0; i < 10; i++){
205             this.bucket.get();
206         }
207     }
208 }
```

```
210  public static void main(String[] args){
211      Bucket bucket = new Bucket(5, 150);
212      Producer pro = new Producer(bucket);
213      Consumer con = new Consumer(bucket);
214
215      new Thread(pro).start();
216      new Thread(pro).start();
217      new Thread(con).start();
218      new Thread(con).start();
219  }
```

```
/Library/Java/JavaVirtualMachines/
PUT:  Token-----  0
PUT:  Token-----  1
PUT:  Token-----  2
PUT:  Token-----  3
PUT:  Token-----  4
GET:  Token-----  4
GET:  Token-----  3
GET:  Token-----  2
GET:  Token-----  1
GET:  Token-----  0
PUT:  Token-----  5
PUT:  Token-----  6
PUT:  Token-----  7
PUT:  Token-----  8
PUT:  Token-----  9
GET:  Token-----  9
GET:  Token-----  8
GET:  Token-----  7
GET:  Token-----  6
GET:  Token-----  5
PUT:  Token-----  0
PUT:  Token-----  1
PUT:  Token-----  2
GET:  Token-----  2
GET:  Token-----  1
GET:  Token-----  0
PUT:  Token-----  3
```

line 212-215: define producer thread and start

line 213-217: define consumer thread and start

Q&A