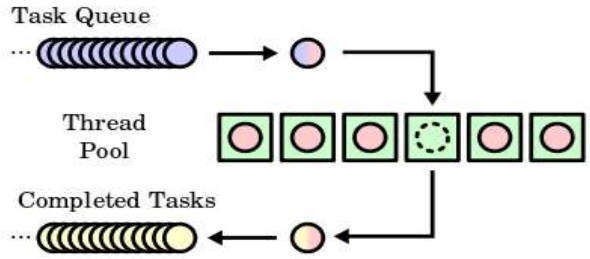
Advanced Java Session 1 and 2

Threads

1) Threads vs Proecesses
2) Thread Lifecycle
3) Creating and Managing Threads (Thread class , Runnable & Callable)
4) Synchrinozation
5) Inter Thread Communication
------ Session 2----------
6) Thread pools and Executors
7) Handling Interrupts and Exceptions
8) Pitfalls with threads
   a) Deadlock
   b) Race Condition
   c) Dirty Read
9) Real life usage of Threads Example (producer – consumer)
10) Additional Topics
    a) Thread Schedulling

6) Thread Pool and Executors

A **thread pool is a collection of pre-initialized threads**. Generally, the collection size is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using the same threads. If there are more tasks than threads, then tasks need to wait in a queue like structure (FIFO – First in first out).
When any thread completes its execution, it can pickup a new task from the queue and execute it. When all tasks are completed, the threads remain active and wait for more tasks in the thread pool.



Thread Pool

A watcher keeps watching the queue (usually BlockingQueue) for any new tasks. As soon as tasks come, threads start picking up tasks and executing them again.

Callable

Java provides two approaches for creating threads one by implementing the Runnable interface and the other by inheriting the Thread class. However, one important feature missing with the implementation of the **Runnable interface is that it is not possible for a thread to return something when it completes its execution, i.e., when the run() method execution is over. In order to support this feature, the Java Callable interface is used**.

Future:

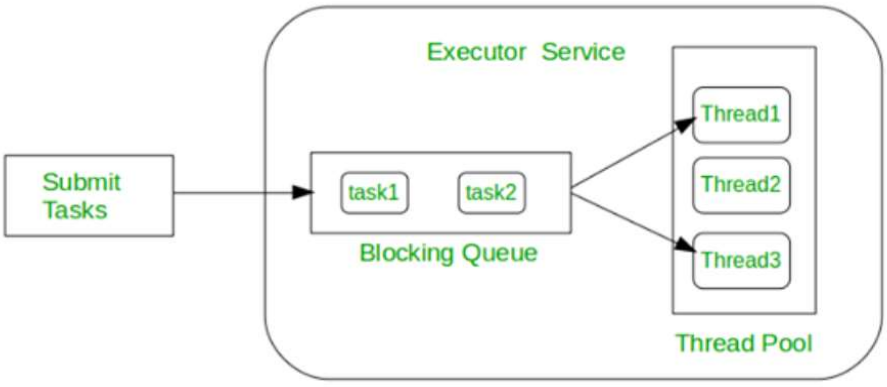
A Future interface provides methods to check if the computation is complete, to wait for its completion and to retrieve the results of the computation. The result is retrieved using Future's get() method when the computation has completed, and it blocks until it is completed.
Future and FutureTask both are available in java.util.concurrent package from Java 1.5.

FutureTask:

1.FutureTask is a concrete implementation of the Future, Runnable, and RunnableFuture interfaces and therefore can be submitted to an ExecutorService instance for execution.
2.When calling ExecutorService.submit() on a Callable or Runnable instance, the ExecutorService returns a Future representing the task. and one can create it manually also.
3.FutureTask acts similar to a CountDownLatch when calling get() in that it waits for the task to complete or error out.
4.Behaviour of the parameterless get() method depends on the state of the task. If tasks are not completed, get() method blocks until the task is completed. Once the task complete, it returns the result or throws an ExecutionException.
5.An overloaded variant of get() allows passing a timeout parameter to limit the amount of time the thread waits for a result.

Blocking Queue
The BlockingQueue interface in Java is added in Java 1.5 along with various other concurrent Utility classes .
BlockingQueue interface supports flow control (in addition to queue) by introducing blocking if either BlockingQueue is full or empty. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more elements or clearing the queue completely. Similarly, it blocks a thread trying to delete from an empty queue until some other threads insert an item. BlockingQueue does not accept a null value. If we try to enqueue the null item, then it throws NullPointerException.

Executor Framework



Executor framework (since Java 1.5) solved this problem. The framework consists of three main interfaces (and lots of child interfaces):

1. ***Executor,***

2. ***ExecutorService***
3. ***ThreadPoolExecutor***

Executor Service

```
//Executes only one thread ExecutorService es =
Executors.newSingleThreadExecutor();
//Internally manages thread pool of 2 threads ExecutorService es =
Executors.newFixedThreadPool(2);
//Internally manages thread pool of 10 threads to run scheduled tasks
ExecutorService es = Executors.newScheduledThreadPool(10);
```

Executing *Runnable* Tasks

We can execute runnables using the following methods :
void execute(Runnable task) – executes the given command at some time in the future.

Future submit(Runnable task) – submits a runnable task for execution and returns a `Future` representing that task. The Future's `get()` method will return `null` upon successful completion

Future submit(Runnable task, T result) – Submits a runnable task for execution and returns a `Future` representing that task. The Future's `get()` method will return the given `result` upon successful completion.

Interrupt

interrupt() method of thread class is used to interrupt the thread. If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked) then using the interrupt() method, we can interrupt the thread execution by throwing InterruptedException.

If the thread is not in the sleeping or waiting state then calling the interrupt() method performs a normal behavior and doesn't interrupt the thread but sets the interrupt flag to true.

Syntax
public void interrupt()

Exception
SecurityException: This exception throws if the current thread cannot modify the thread.
Catch InterruptedException to handle interrupt .

Pitfalls In Threads

1) Deadlock

A **lock** occurs when multiple processes try to access the same resource at the same time.
One process loses out and must wait for the other to finish.

A **deadlock** occurs when the waiting process is still holding on to another resource that the first needs before it can finish.

So, an example:

Resource A and resource B are used by process X and process Y

- X starts to use A.
- X and Y try to start using B
- Y 'wins' and gets B first
- now Y needs to use A
- A is locked by X, which is waiting for Y

The best way to avoid deadlocks is to avoid having processes cross over in this way. Reduce the need to lock anything as much as you can.
In databases avoid making lots of changes to different tables in a single transaction, avoid triggers
Note : Livelock

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

2) Race Condition

A condition in which the critical section (a part of the program where shared memory is accessed) is concurrently executed by two or more threads. It leads to incorrect behavior of a program.
In layman terms, a **race condition** can be defined as, a condition in which two or more threads compete together to get certain shared resources.

For example, if thread A is reading data from the linked list and another thread B is trying to delete the same data. This process leads to a race condition that may result in run time error.

https://www.javatpoint.com/race-condition-in-java

3) Dirty Read

- In typical database transactions, one transaction reads changes the value while the other reads the value before committing or rolling back by the first transaction. This reading process is called as 'dirty read'.

Because there is always a chance that the first transaction might rollback the change which causes the second transaction reads an invalid value.

- It takes no notice of any other lock taken by another process.

- It is a problem if the uncommitted transaction fails or is rolled back.

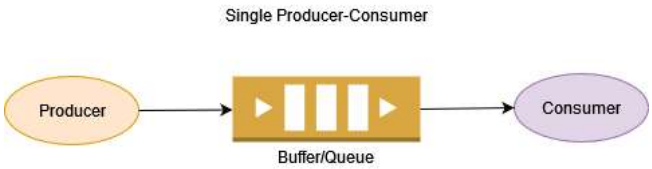
Producer – Consumer Problem

Java Concurrency article.

2. Producer-Consumer Problem

Producer and Consumer are two separate processes. Both processes share a common buffer or queue. The producer continuously produces certain data and pushes it onto the buffer, whereas the consumer consumes those data from the buffer.

Let's review a diagram showing this simple scenario:

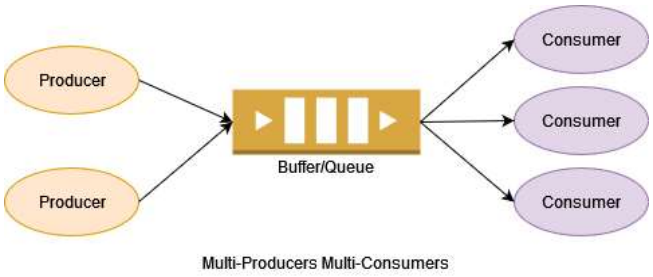


Inherently, this problem has certain complexities to deal with:

1. Both producer and consumer may try to update the queue at the same time. This could lead to data loss or inconsistencies.
2. Producers might be slower than consumers. In such cases, the consumer would process elements fast and wait.
3. In some cases, the consumer can be slower than a producer. This situation leads to a queue overflow issue.

4. In real scenarios, we may have multiple producers, multiple consumers, or both. This may cause the same message to be processed by different consumers.

The diagram below depicts a case with multiple producers and multiple consumers:



Multi-Producers Multi-Consumers

We need to handle resource sharing and synchronization to solve a few complexities:

Synchronization on queue while adding and removing data

On queue empty, the consumer has to wait until the producer adds new data to the queue

When the queue is full, the producer has to wait until the consumer consumes data and the queue has some empty buffer

Example in Code

Java Thread schedulling algorithems

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.

Thread Scheduler Algorithms

On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

First Come First Serve Scheduling:

In this scheduling algorithm, the scheduler picks the threads thar arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

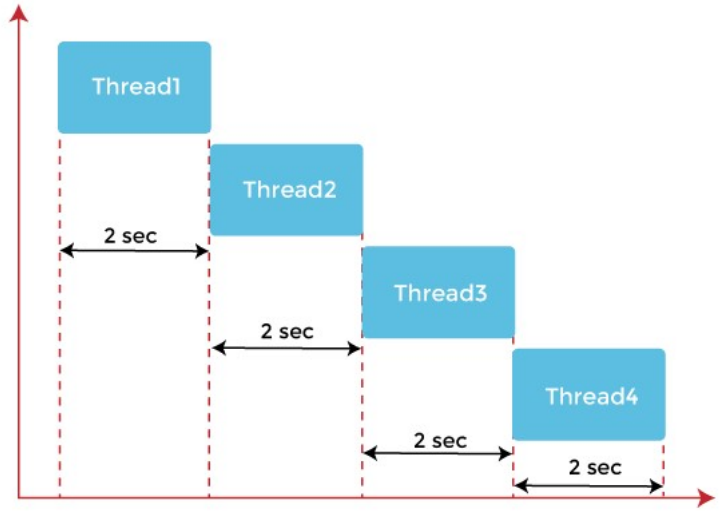
In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

First Come First Serve Scheduling

Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

Time-slicing scheduling:

Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

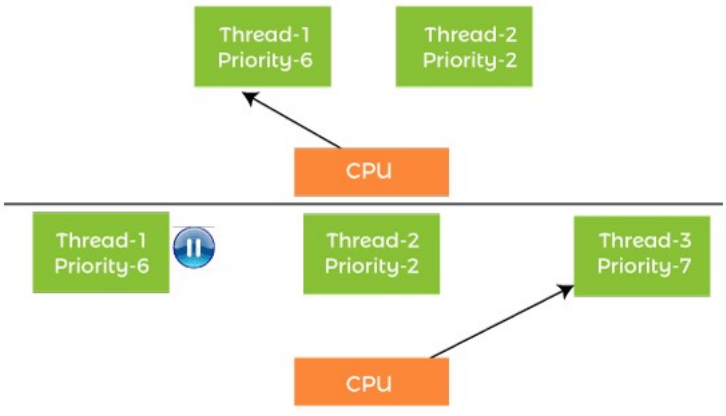


Time slicing scheduling

In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

Preemptive-Priority Scheduling:

The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Preemptive-Priority Scheduling

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

Important Points

1) Once you start a thread you cannot make it as a daemon thread , and daemon threadsd are controlled by JVM , once you make a thread as daemon you will have no control over its lifecycle .

2) Garbedge Collections

We can also request JVM to run Garbage Collector. There are two ways to do it :

1. Using *System.gc()* method: System class contain static method *gc()* for requesting JVM to run Garbage Collector.
2. Using *Runtime.getRuntime().gc()* method: **Runtime class** allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.
3. There is no guarantee that any of the above two methods will run Garbage Collector.
4. The call *System.gc()* is effectively equivalent to the call : *Runtime.getRuntime().gc()*