Java Thread

* Thread class v/s Runnable v/s Callable interface
  1. Using Thread will lose on multiple inheritance
  2. Thread makes sense you are adding new behaviour, extending/adding functionality
  3. Runnable is always prime choice if you are not doing pt#2
  4. Runnable represent a Task which can be executed by Thread/Executor/other means
  5. Runnable is suitable for situation where we aren't looking for result
  6. Callable can be executed only by Executor
  7. Callable's call method can returns Future. Future has get(), isDone() etc methods
* ExecutorService
  1. ThreadPoolExecutor
  2. ScheduledExecutorService
     + executorService.execute(runnableTask)
     + Future<String> future = executorService.submit(callableTask)
     + shutdown() -> void
     + shutdownNow() -> List<Runnable>
* ForkJoinPool
  1. RecursiveTask -> return result
  2. RecursiveAction -> No return
* CyclicBarrier
* CountDownLatch
* JAVA Thread Scheduling
  1. Features :
     + The JVM schedules using a preemptive , priority based scheduling algorithm.
     + All Java threads have a priority and the thread with the highest priority is scheduled to run by the JVM.
     + In case two threads have the same priority a FIFO ordering is followed.
* Uncaught exception handling
  1. Thread.setUncaughtExceptionHandler()
* ThreadGroup
  1. Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call

ThreadGroup tg1 = **new** ThreadGroup("Group A");

Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");

Semaphore in Java

[**3.8**](https://www.geeksforgeeks.org/medium/)

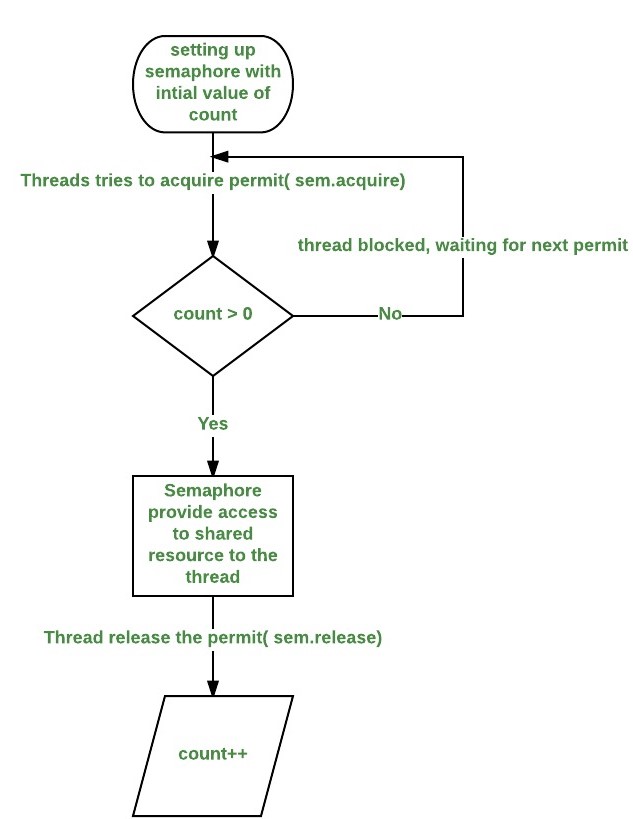
A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

**Working of semaphore**

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

* If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.
* Otherwise, the thread will be blocked until a permit can be acquired.
* When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.
* If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide **Semaphore** class in *java.util.concurrent* package that implements this mechanism, so you don’t have to implement your own semaphores.

**Flow Diagram :**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/d2.jpeg)

**Constructors in Semaphore class :** There are two constructors in Semaphore class.

Semaphore(int num)

Semaphore(int num, boolean how)

Here, *num* specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting *how* to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

**Using Semaphores as Locks(**[**preventing race condition**](https://practice.geeksforgeeks.org/problems/what-is-race-condition)**)**

We can use a semaphore to lock access to a resource, each thread that wants to use that resource must first call *acquire( )* before accessing the resource to acquire the lock. When the thread is done with the resource, it must call *release( )* to release lock. Here is an example that demonstrate this:

|  |
| --- |
| // java program to demonstrate  // use of semaphores Locks  import java.util.concurrent.\*;    //A shared resource/class.  class Shared  {      static int count = 0;  }    class MyThread extends Thread  {      Semaphore sem;      String threadName;      public MyThread(Semaphore sem, String threadName)      {          super(threadName);          this.sem = sem;          this.threadName = threadName;      }        @Override      public void run() {            // run by thread A          if(this.getName().equals("A"))          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count++;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread B to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                    // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }            // run by thread B          else          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count--;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread A to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                  // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }      }  }    // Driver class  public class SemaphoreDemo  {      public static void main(String args[]) throws InterruptedException      {          // creating a Semaphore object          // with number of permits 1          Semaphore sem = new Semaphore(1);            // creating two threads with name A and B          // Note that thread A will increment the count          // and thread B will decrement the count          MyThread mt1 = new MyThread(sem, "A");          MyThread mt2 = new MyThread(sem, "B");            // stating threads A and B          mt1.start();          mt2.start();            // waiting for threads A and B          mt1.join();          mt2.join();            // count will always remain 0 after          // both threads will complete their execution          System.out.println("count: " + Shared.count);      }  } |

Run on IDE

Output:

Starting A

Starting B

B is waiting for a permit.

B gets a permit.

A is waiting for a permit.

B: -1

B: -2

B: -3

B: -4

B: -5

B releases the permit.

A gets a permit.

A: -4

A: -3

A: -2

A: -1

A: 0

A releases the permit.

count: 0

**Note :**The output can be different in different executions of above program, but final value of *count* variable will always remain 0.

**Explanation of above program :**

* The program uses a semaphore to control access to the *count* variable, which is a static variable within the Shared class. *Shared.count* is incremented five times by thread A and decremented five times by thread B.To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.
* Notice the call to sleep( ) within run( ) method inside MyThread class. It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by thread A and then decremented five times by thread B. The increments and decrements are not intermixed at assembly code.
* Without the use of the semaphore, accesses to *Shared.count* by both threads would have occurred simultaneously, and the increments and decrements would be intermixed.To confirm this, try commenting out the calls to *acquire( )* and *release( )*. When you run the program, you will see that access to Shared.count is no longer synchronized, thus you will not always get *count* value 0.

CountDownLatch in Java

[**2**](https://www.geeksforgeeks.org/easy/)

CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.

**Working of CountDownLatch:**  
When we create an object of CountDownLatch, we specify the number if threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

**Example of CountDownLatch in JAVA:**

|  |
| --- |
| /\* Java Program to demonstrate how to use CountDownLatch,     Its used when a thread needs to wait for other threads     before starting its work. \*/  import java.util.concurrent.CountDownLatch;    public class CountDownLatchDemo  {      public static void main(String args[]) throws InterruptedException      {          // Let us create task that is going to wait for four          // threads before it starts          CountDownLatch latch = new CountDownLatch(4);            // Let us create four worker threads and start them.          Worker first = new Worker(1000, latch, "WORKER-1");          Worker second = new Worker(2000, latch, "WORKER-2");          Worker third = new Worker(3000, latch, "WORKER-3");          Worker fourth = new Worker(4000, latch, "WORKER-4");          first.start();          second.start();          third.start();          fourth.start();            // The main task waits for four threads          latch.await();            // Main thread has started          System.out.println(Thread.currentThread().getName() +                             " has finished");      }  }    // A class to represent threads for which the main thread  // waits.  class Worker extends Thread  {      private int delay;      private CountDownLatch latch;        public Worker(int delay, CountDownLatch latch,                                       String name)      {          super(name);          this.delay = delay;          this.latch = latch;      }        @Override      public void run()      {          try          {              Thread.sleep(delay);              latch.countDown();              System.out.println(Thread.currentThread().getName()                                 + " finished");          }          catch (InterruptedException e)          {              e.printStackTrace();          }      }  } |

Run on IDE

**Output:**

WORKER-1 finished

WORKER-2 finished

WORKER-3 finished

WORKER-4 finished

main has finished

**Facts about CountDownLatch:**

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once countdown reaches to zero.
3. countDown() method decrements the count and await() method blocks until count == 0

Java.util.concurrent.CyclicBarrier in Java

[**4.4**](https://www.geeksforgeeks.org/hard/)

CyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

**Working of CyclicBarrier**

CyclicBarriers are defined in java.util.concurrent package. First a new instance of a CyclicBarriers is created specifying the number of threads that the barriers should wait upon.

**CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads);**

Each and every thread does some computation and after completing it’s execution, calls await() methods as shown:

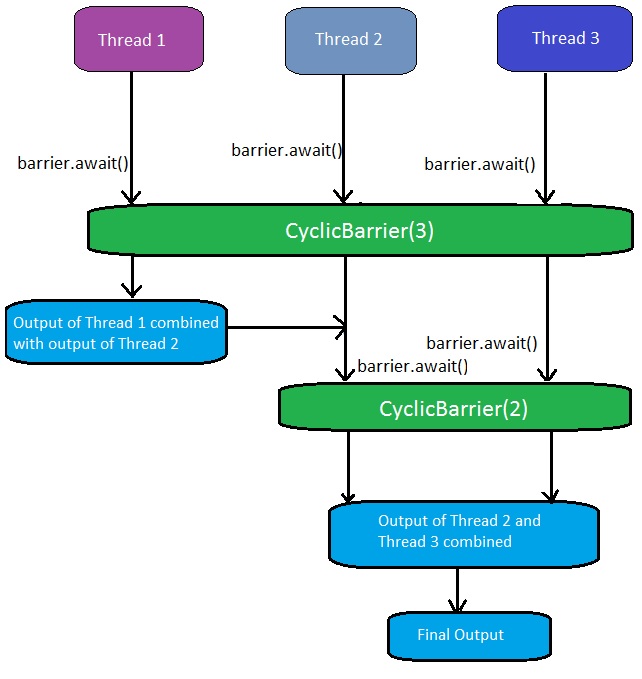
public void run()

{

// thread does the computation

newBarrier.await();

}

**Image to represent the working of CyclicBarrier:**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/barrier.jpg)  
Once the number of threads that called await() equals **numberOfThreads**, the barrier then gives a way for the waiting threads. The CyclicBarrier can also be initialized with some action that is performed once all the threads have reached the barrier. This action can combine/utilize the result of computation of individual thread waiting in the barrier.

Runnable action = ...

//action to be performed when all threads reach the barrier;

CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads, action);

**Important Methods of CyclicBarrier:**

1. **getParties:** Returns the number of parties required to trip this barrier.  
   **Syntax:**

public int getParties()

**Returns:**  
the number of parties required to trip this barrier

1. **reset:** Resets the barrier to its initial state.  
   **Syntax:**

public void reset()

**Returns:**  
void but resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException.

1. **isBroken:** Queries if this barrier is in a broken state.  
   **Syntax:**

public boolean isBroken()

**Returns:**  
true if one or more parties broke out of this barrier due to interruption or timeout since construction or the last reset, or a barrier action failed due to an exception; false otherwise.

1. **getNumberWaiting:** Returns the number of parties currently waiting at the barrier.  
   **Syntax:**

public int getNumberWaiting()

**Returns:**  
the number of parties currently blocked in await()

1. **await:** Waits until all parties have invoked await on this barrier.  
   **Syntax:**

public int await() throws InterruptedException, BrokenBarrierException

**Returns:**  
the arrival index of the current thread, where index getParties() – 1 indicates the first to arrive and zero indicates the last to arrive.

1. **await:** Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.  
   **Syntax:**
2. public int await(long timeout, TimeUnit unit)
3. throws InterruptedException,

BrokenBarrierException, TimeoutException

**Returns:**  
the arrival index of the current thread, where index getParties() – 1 indicates the first to arrive and zero indicates the last to arrive

|  |
| --- |
| //JAVA program to demonstrate execution on Cyclic Barrier    import java.util.concurrent.TimeUnit;  import java.util.concurrent.TimeoutException;  import java.util.concurrent.BrokenBarrierException;  import java.util.concurrent.CyclicBarrier;    class Computation1 implements Runnable  {      public static int product = 0;      public void run()      {          product = 2 \* 3;          try          {              Tester.newBarrier.await();          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }      }  }    class Computation2 implements Runnable  {      public static int sum = 0;      public void run()      {          // check if newBarrier is broken or not          System.out.println("Is the barrier broken? - " + Tester.newBarrier.isBroken());          sum = 10 + 20;          try          {              Tester.newBarrier.await(3000, TimeUnit.MILLISECONDS);                // number of parties waiting at the barrier              System.out.println("Number of parties waiting at the barrier "+              "at this point = " + Tester.newBarrier.getNumberWaiting());          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }          catch (TimeoutException e)          {              e.printStackTrace();          }      }  }      public class Tester implements Runnable  {      public static CyclicBarrier newBarrier = new CyclicBarrier(3);        public static void main(String[] args)      {          // parent thread          Tester test = new Tester();            Thread t1 = new Thread(test);          t1.start();      }      public void run()      {          System.out.println("Number of parties required to trip the barrier = "+          newBarrier.getParties());          System.out.println("Sum of product and sum = " + (Computation1.product +          Computation2.sum));            // objects on which the child thread has to run          Computation1 comp1 = new Computation1();          Computation2 comp2 = new Computation2();            // creation of child thread          Thread t1 = new Thread(comp1);          Thread t2 = new Thread(comp2);            // moving child thread to runnable state          t1.start();          t2.start();            try          {              Tester.newBarrier.await();          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }            // barrier breaks as the number of thread waiting for the barrier          // at this point = 3          System.out.println("Sum of product and sum = " + (Computation1.product +          Computation2.sum));            // Resetting the newBarrier          newBarrier.reset();          System.out.println("Barrier reset successful");      }  } |

**Explanation:** The value of (sum + product) = 0 is printed on the console because the child thread hasn't yet ran to set the values of sum and product variable. Following this, (sum + product) = 36 is printed on the console because the child threads ran setting the values of sum and product. Furthermore, the number of waiting thread on the barrier reached 3, due to which the barrier then allowed all thread to pass and finally 36 was printed. The value of “Number of parties waiting at the barrier at this point” = 0 because all the three threads had already called await() method and hence, the barrier is no longer active. In the end, newBarrier is reset and can be used again.

**BrokenBarrierException**

A barrier breaks when any of the waiting thread leaves the barrier. This happens when one or more waiting thread is interrupted or when the waiting time is completed because the thread called the await() methods with a timeout as follows:

newBarrier.await(1000, TimeUnit.MILLISECONDS);

// thread calling this await()

// methods waits for only 1000 milliseconds.

When the barrier breaks due to one of more participating threads, the await() methods of all the other threads throws a BrokenThreadException. Whereas, the threads that are already waiting in the barriers have their await() call terminated.

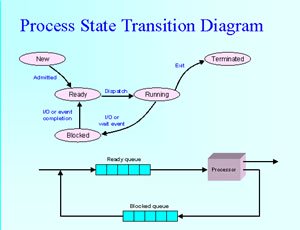
**Difference between a CyclicBarrier and a CountDownLatch**

* A CountDownLatch can be used only once in a program(until it’s count reaches 0).
* A CyclicBarrier can be used again and again once all the threads in a barriers is released.

Difference between Process and Thread

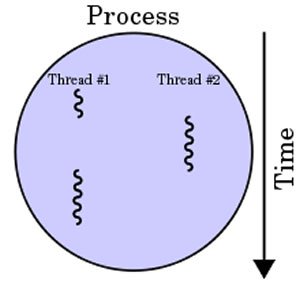
Key difference: Thread and Process are two closely related terms in multi-threading. The main difference between the two terms is that the threads are a part of a process, i.e. a process may contain one or more threads, but a thread cannot contain a process.

In programming, there are two basic units of execution: processes and threads. They both execute a series of instructions. Both are initiated by a program or the operating system. This article helps to differentiate between the two units.



A process is an instance of a program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently. A program is a collection of instructions; a process is the actual execution of those instructions.

A process has a self-contained execution environment. It has a complete set of private basic run-time resources; in particular, each process has its own memory space. Processes are often considered similar to other programs or applications. However, the running of a single application may in fact be a set of cooperating processes. To facilitate communication between the processes, most operating systems use Inter Process Communication (IPC) resources, such as pipes and sockets. The IPC resources can also be used for communication between processes on different systems. Most applications in a virtual machine run as a single process. However, it can create additional processes using a process builder object.



In computers, a thread can execute even the smallest sequence of programmed instructions that can be managed independently by an operating system. The applications of threads and processes differ from one operating system to another. However, the threads are made of and exist within a process; every process has at least one. Multiple threads can also exist in a process and share resources, which helps in efficient communication between threads.

On a single processor, multitasking takes place as the processor switches between different threads; it is known as multithreading. The switching happens so frequently that the threads or tasks are perceived to be running at the same time. Threads can truly be concurrent on a multiprocessor or multi-core system, with every processor or core executing the separate threads simultaneously.

In summary, threads may be considered lightweight processes, as they contain simple sets of instructions and can run within a larger process. Computers can run multiple threads and processes at the same time.

Comparison between Process and Thread:

|  |  |  |
| --- | --- | --- |
|  | Process | Thread |
| Definition | An executing instance of a program is called a process. | A thread is a subset of the process. |
| Process | It has its own copy of the data segment of the parent process. | It has direct access to the data segment of its process. |
| Communication | Processes must use inter-process communication to communicate with sibling processes. | Threads can directly communicate with other threads of its process. |
| Overheads | Processes have considerable overhead. | Threads have almost no overhead. |
| Creation | New processes require duplication of the parent process. | New threads are easily created. |
| Control | Processes can only exercise control over child processes. | Threads can exercise considerable control over threads of the same process. |
| Changes | Any change in the parent process does not affect child processes. | Any change in the main thread may affect the behavior of the other threads of the process. |
| Memory | Run in separate memory spaces. | Run in shared memory spaces. |
| File descriptors | Most file descriptors are not shared. | It shares file descriptors. |
| File system | There is no sharing of file system context. | It shares file system context. |
| Signal | It does not share signal handling. | It shares signal handling. |
| Controlled by | Process is controlled by the operating system. | Threads are controlled by programmer in a program. |
| Dependence | Processes are independent. | Threads are dependent. |