Bharati Vidyapeeth (Deemed to be University)
Department of Engineering and Technology
Kharghar, Navi Mumbai

Department of Artificial Intelligence Machine Learning

Experiment No. 03

Name: Kamran Khan	Roll No: 11
Subject: High Performance Parallel Computing	Class/Batch: CSE-AIML / B1
Date of Performance: __ / __ / 2024	Date of Submission: __ / __ / 2024

AIM

To understand the concept of Thread.sleep() using Java, and implement sorting with concurrent threads and thread sleep function

Theory/Procedure/Algorithm

Concept of `Thread.sleep()` in Java

In Java, the `Thread` class, located in the `java.lang` package, represents a thread of execution within a program. One of the key methods of this class is the static method `Thread.sleep()`, which allows a thread to pause its execution for a specified period of time. After the specified duration, the thread resumes execution unless it is interrupted.

Variations of `Thread.sleep()` Method

There are two primary overloads of the `Thread.sleep()` method:

1. `sleep(long millis)` : Pauses the current thread for the given number of milliseconds.
2. `sleep(long millis, int nanos)` : Pauses the current thread for the specified number of milliseconds and nanoseconds (where nanos ranges from 0 to 999,999).

Method Signatures:

1. `public static void sleep(long millis) throws InterruptedException`
2. `public static void sleep(long millis, int nanos) throws InterruptedException`

Both versions of the method throw an `InterruptedException` if another thread interrupts the current thread during its sleep period. These methods do not return any value and must be used in a `try-catch` block or declared with `throws` to handle the exception.

Parameters:

- **millis**: The number of milliseconds the thread should sleep.
- **nanos**: (Optional) An additional sleep time in nanoseconds (between 0 and 999,999).

If invalid values for `millis` or `nanos` are passed (such as negative values or out-of-bound nanosecond values), an `IllegalArgumentException` will be thrown.

Key Points:

- **Thread Control**: The method pauses the execution of the current thread without affecting the execution of other threads.
- **System Load Impact**: While `Thread.sleep()` aims to pause the thread for a specific amount of time, the actual sleep time may vary depending on system load and scheduling. In modern systems, this variance is usually small.
- **Exception Handling**: The `InterruptedException` is a checked exception, which means it must either be caught in a `try-catch` block or declared in the method signature using `throws InterruptedException`.

Example:

In [ ]:
```
try {
    Thread.sleep(1000); // Pauses execution for 1 second
} catch (InterruptedException e) {
    // Handle the exception if the thread is interrupted
    System.out.println("Thread was interrupted");
}
```

Usage in Concurrent Sorting:

In a multi-threaded sorting algorithm, such as merge sort or quick sort, `Thread.sleep()` can be used to simulate delays in sorting operations. Each thread may sort a portion of the array, and `Thread.sleep()` can help manage thread timing, synchronization, or simulate real-world processing delays. This allows for better control over when and how threads perform their tasks.

For example, when implementing concurrent sorting:

- One thread may sort a portion of the array.
- After completing its portion, the thread can call `Thread.sleep()` to allow another thread to perform its operation or simulate a processing delay.

This ensures that multiple threads don't overwhelm the system and helps manage the timing of each sorting task.

Task 1: Develop a Java code for implementing parallel sort for 1-Dimentional array of size 100. Code must create 04 parallel threads with equal division of input data.

In [ ]:
```
import java.util.Arrays;

public class ParallelSort {
  private static final int ARRAY_SIZE = 100;
  private static final int NUM_THREADS = 4;

  public static void main(String[] args) {
    int[] array = createArray();
    System.out.println("Before sorting: " + Arrays.toString(array));

    Thread[] threads = new Thread[NUM_THREADS];
    int segmentSize = ARRAY_SIZE / NUM_THREADS;

    for (int i = 0; i < NUM_THREADS; i++) {
      int startIndex = i * segmentSize;
      int endIndex = (i == NUM_THREADS - 1) ? ARRAY_SIZE - 1
```

```
                                        : (startIndex + segmentSize - 1);
    threads[i] = new Thread(new SortTask(array, startIndex, endIndex));
    threads[i].start();
  }

  for (Thread thread : threads) {
    try {
      thread.join(); // Ensure all threads complete before moving forward
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  // Perform final merge sort to combine the sorted segments
  mergeSort(array, 0, ARRAY_SIZE - 1);

  System.out.println("After sorting: " + Arrays.toString(array));
}

// Function to create a random array
private static int[] createArray() {
  int[] array = new int[ARRAY_SIZE];
  for (int i = 0; i < ARRAY_SIZE; i++) {
    array[i] =
        (int) (Math.random() * 400); // Random numbers between 0 and 400
  }
  return array;
}

// Merge Sort function
private static void mergeSort(int[] array, int left, int right) {
  if (left < right) {
    int mid = (left + right) / 2;
    mergeSort(array, left, mid);
    mergeSort(array, mid + 1, right);
    merge(array, left, mid, right);
  }
}

// Merge function to combine two sorted halves
private static void merge(int[] array, int left, int mid, int right) {
  int[] temp = new int[right - left + 1];
  int i = left, j = mid + 1, k = 0;

  while (i <= mid && j <= right) {
    if (array[i] <= array[j]) {
      temp[k++] = array[i++];
    } else {
      temp[k++] = array[j++];
    }
  }

  while (i <= mid) {
    temp[k++] = array[i++];
  }

  while (j <= right) {
    temp[k++] = array[j++];
  }

  System.arraycopy(temp, 0, array, left, temp.length);
}

// Task class for sorting each segment
static class SortTask implements Runnable {
```

```
    private int[] array;
    private int startIndex;
    private int endIndex;

    public SortTask(int[] array, int startIndex, int endIndex) {
      this.array = array;
      this.startIndex = startIndex;
      this.endIndex = endIndex;
    }

    @Override
    public void run() {
      Arrays.sort(array, startIndex, endIndex + 1);
    }
  }
}
```

In [1]:
```
from IPython.display import HTML

HTML('<img src="./Task%201%20Output.png" style="height:150px;"/>')
```

Out[1]:
```
Before sorting: [150, 340, 154, 258, 298, 339, 358, 197, 281, 256, 210, 184, 214, 258, 164, 222, 318, 119, 15, 267, 179, 42, 371, 363, 275, 60, 325,
235, 160, 375, 158, 290, 194, 2, 87, 224, 91, 142, 381, 196, 277, 159, 182, 145, 279, 289, 107, 51, 100, 314, 348, 184, 183, 95, 340, 16, 336, 182, 1
45, 341, 266, 224, 264, 20, 139, 50, 118, 251, 116, 190, 197, 156, 207, 214, 237, 203, 101, 397, 397, 80, 96, 192, 356, 393, 380, 216, 106, 93, 387,
32, 384, 319, 327, 23, 174, 271, 71, 110, 302, 374]
After sorting: [2, 15, 16, 20, 23, 32, 42, 50, 51, 60, 71, 80, 87, 91, 93, 95, 96, 100, 101, 106, 107, 110, 116, 118, 119, 139, 142, 145, 145, 150, 1
54, 156, 158, 159, 160, 164, 174, 179, 182, 182, 183, 184, 184, 190, 192, 194, 196, 197, 197, 203, 207, 210, 214, 214, 216, 222, 224, 224, 235, 237,
251, 256, 258, 258, 264, 266, 267, 271, 275, 277, 279, 281, 289, 290, 298, 302, 314, 318, 319, 325, 327, 336, 339, 340, 340, 341, 348, 356, 358, 363,
 371, 374, 375, 380, 381, 384, 387, 393, 397, 397]
```

Task 2: Use Thread Sleep function on the above code and observe the output..

In [ ]:
```
import java.time.LocalTime;
import java.util.Arrays;

public class ParallelSortWithThreadSleep {
  private static final int ARRAY_SIZE = 100;
  private static final int NUM_THREADS = 4;

  public static void main(String[] args) {
    int[] array = createArray();
    System.out.println("Before sorting: " + Arrays.toString(array));

    Thread[] threads = new Thread[NUM_THREADS];
    int segmentSize = ARRAY_SIZE / NUM_THREADS;

    for (int i = 0; i < NUM_THREADS; i++) {
      int startIndex = i * segmentSize;
      int endIndex = (i == NUM_THREADS - 1) ? ARRAY_SIZE - 1
                                            : (startIndex + segmentSize - 1);
      threads[i] = new Thread(new SortTask(array, startIndex, endIndex));
      threads[i].start();
    }

    for (Thread thread : threads) {
      try {
        thread.join(); // Ensure all threads complete before moving forward
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    // Perform final merge sort to combine the sorted segments
```

```
    mergeSort(array, 0, ARRAY_SIZE - 1);

    System.out.println("After sorting: " + Arrays.toString(array));
  }

  // Function to create a random array
  private static int[] createArray() {
    int[] array = new int[ARRAY_SIZE];
    for (int i = 0; i < ARRAY_SIZE; i++) {
      array[i] =
          (int) (Math.random() * 400); // Random numbers between 0 and 400
    }
    return array;
  }

  // Merge Sort function
  private static void mergeSort(int[] array, int left, int right) {
    if (left < right) {
      int mid = (left + right) / 2;
      mergeSort(array, left, mid);
      mergeSort(array, mid + 1, right);
      merge(array, left, mid, right);
    }
  }

  // Merge function to combine two sorted halves
  private static void merge(int[] array, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
      if (array[i] <= array[j]) {
        temp[k++] = array[i++];
      } else {
        temp[k++] = array[j++];
      }
    }

    while (i <= mid) {
      temp[k++] = array[i++];
    }

    while (j <= right) {
      temp[k++] = array[j++];
    }

    System.arraycopy(temp, 0, array, left, temp.length);
  }

  // Task class for sorting each segment
  static class SortTask implements Runnable {
    private int[] array;
    private int startIndex;
    private int endIndex;

    public SortTask(int[] array, int startIndex, int endIndex) {
      this.array = array;
      this.startIndex = startIndex;
      this.endIndex = endIndex;
    }

    @Override
    public void run() {
      try {
        // Add sleep to simulate delay
```

```
        System.out.println("[" + LocalTime.now()
            + "] Sorting segment: " + startIndex + " to " + endIndex);
        Thread.sleep(2000); // Sleep for 2 seconds before sorting
        Arrays.sort(array, startIndex, endIndex + 1);
        System.out.println("[" + LocalTime.now()
            + "] Sorted segment: " + startIndex + " to " + endIndex);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

In [3]:
```
from IPython.display import HTML

HTML('<img src="./Task%202%20Output.png" style="height:300px;"/>')
```

Out[3]:
```
Before sorting: [97, 81, 327, 324, 48, 235, 8, 130, 173, 80, 23, 243, 310, 206, 7, 176, 44, 18, 374, 360, 188, 24, 168, 198, 187, 8
3, 121, 59, 244, 38, 257, 32, 220, 332, 276, 253, 107, 259, 89, 161, 21, 282, 250, 33, 182, 150, 138, 368, 346, 137, 345, 255, 111,
 48, 107, 132, 287, 268, 187, 327, 185, 67, 246, 274, 200, 27, 360, 102, 384, 31, 340, 209, 59, 317, 212, 104, 242, 103, 195, 82, 5
5, 3, 84, 231, 24, 233, 307, 379, 1, 380, 339, 330, 208, 313, 360, 69, 154, 388, 47, 209]
[23:57:01.394439100] Sorting segment: 50 to 74
[23:57:01.394439100] Sorting segment: 0 to 24
[23:57:01.394439100] Sorting segment: 25 to 49
[23:57:01.394439100] Sorting segment: 75 to 99
[23:57:03.417693400] Sorted segment: 75 to 99
[23:57:03.417693400] Sorted segment: 25 to 49
[23:57:03.417693400] Sorted segment: 50 to 74
[23:57:03.417693400] Sorted segment: 0 to 24
After sorting: [1, 3, 7, 8, 18, 21, 23, 24, 24, 27, 31, 32, 33, 38, 44, 47, 48, 48, 55, 59, 59, 67, 69, 80, 81, 82, 83, 84, 89, 97,
 102, 103, 104, 107, 107, 111, 121, 130, 132, 137, 138, 150, 154, 161, 168, 173, 176, 182, 185, 187, 187, 188, 195, 198, 200, 206,
208, 209, 209, 212, 220, 231, 233, 235, 242, 243, 244, 246, 250, 253, 255, 257, 259, 268, 274, 276, 282, 287, 307, 310, 313, 317, 3
24, 327, 327, 330, 332, 339, 340, 345, 346, 360, 360, 360, 368, 374, 379, 380, 384, 388]
```

Reference materials: https://www.w3resource.com/java-exercises/thread/java-thread-exercise-3.php

CONCLUSION

The experiment demonstrated the use of the `Thread.sleep()` method in Java to control the execution timing of concurrent threads during a parallel sorting process. By introducing delays, we observed how thread timing can be managed without affecting the overall sorting outcome. This reinforces the concept of thread management and synchronization in multi-threaded applications.

ASSESSMENT

Timely Submission (7)	Presentation (06)	Understanding (12)	Total (25)	Sign