PROGRAMACIÓN CONCURRENTE

TRABAJO PRÁCTICO N°8

Tema: Executorservice

Fecha de Entrega: Jueves 24/10/2024

Grupo 11 - 2024 - Facultad de Ingeniería

Integrantes:

Apellido y Nombres	Carrera	LU	DNI	E-mail
Ajalla Daniel Ernesto	Ing. Informática	8253	42748428	ernestoajalla2021@gmail.com
Quinta Samuel Humberto	Ing. Informática	7719	41902623	quintasamuel28@gmail.com
Echenique Juan Facundo	Ing. Informática	9518	44197218	juanfacu11@gmail.com
Llampa Hector Wilfredo	Ing. Informática	7128	36591175	wilyhc11@gmail.com
Farfán Maximiliano Gabriel	Ing. Informática	9182	45172134	mgf040903@gmail.com
Reinoso Flavio Ignacio	Ing. Informática	7996	41844711	41844711@fi.unju.edu.ar

2) Se desea construir una clase que posea 2 métodos, el primero (Tarea1) obtendrá la fecha y hora actual del sistema (HH:mm:ss:S) y la almacenará en una lista. El segundo método (Tarea2), trabajará con el último número agregado de la lista, con este dato obtendrá el valor de milisegundos de la fecha/hora obtenido, y si el valor es un número primo lo agregará a un archivo llamado Primos.txt, si el valor de milisegundos no es primo, lo agregará a un archivo llamado NoPrimos.txt. Dichas tareas correrán con una pausa inicial de 2" y de forma ininterrumpida cada 2".

```java
public class Inciso02 {
    private static final List<String> listaFechas = new ArrayList<>();
    private static final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm:ss:S");
    private static final ScheduledExecutorService servicio = Executors.newScheduledThreadPool(2);
    public static void main(String[] args) {
        Runnable tarea1 = new Runnable() {
            @Override
            public void run() {
                try {
                    String fechaActual = LocalDateTime.now().format(formatter);
                    synchronized (listaFechas) {
                        listaFechas.add(fechaActual);
                    }
                    System.out.println("Tarea1: Fecha y hora almacenada: " + fechaActual);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
        Runnable tarea2 = new Runnable() {
            @Override
            public void run() {
                try {
                    String ultimaFecha;
                    synchronized (listaFechas) {
                        if (listaFechas.isEmpty()) {
                            return;
                        }
                        ultimaFecha = listaFechas.get(listaFechas.size() - 1);
                    }
                    int milisegundos = Integer.parseInt(ultimaFecha.split(":")[2]);
                    if (esPrimo(milisegundos)) {
                        escribirArchivo("Primos.txt", milisegundos);
                    } else {
                        escribirArchivo("NoPrimos.txt", milisegundos);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
        servicio.scheduleAtFixedRate(tarea1, 2, 2, TimeUnit.SECONDS);
        servicio.scheduleAtFixedRate(tarea2, 2, 2, TimeUnit.SECONDS);
    }
    private static boolean esPrimo(int numero) {
        if (numero <= 1) return false;
        for (int i = 2; i <= Math.sqrt(numero); i++) {
            if (numero % i == 0) return false;
        }
        return true;
    }
    private static void escribirArchivo(String nombreArchivo, int milisegundos) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(nombreArchivo, true))) {
            writer.write(milisegundos + "\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

EN LA IMAGEN ANTERIOR SE MOSTRÓ LA IMPLEMENTACIÓN DEL EJERCICIO SOLICITADO (EJERCICIO 2), A CONTINUACIÓN EXPLICAREMOS SU FUNCIONAMIENTO CON MÁS DETALLE:

VARIABLES ESTÁTICAS Y FINALES

```
    private static final List<String> listaFechas = new ArrayList<>();
    private static final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm:ss:S");
    private static final ScheduledExecutorService servicio = Executors.newScheduledThreadPool(2);
```

1. listaFechas: Una lista de cadenas (String) para almacenar fechas en formato "HH:mm:ss:S". Se utiliza ArrayList como implementación de la interfaz List.
2. formatter: Un objeto DateTimeFormatter con el patrón "HH:mm:ss:S" para formatear fechas y horas.
3. Crea un objeto ScheduledExecutorService con un tamaño de piscina de 2 hilos.
4. Programa la tarea tarea1 para ejecutarse cada 2 segundos utilizando scheduleAtFixedRate().
5. Programa la tarea tarea2 para ejecutarse cada 2 segundos utilizando scheduleAtFixedRate().

Tarea 1: Almacenar hora actual

```
Runnable tarea1 = new Runnable() {
    @Override
    public void run() {
        try {
            String fechaActual = LocalDateTime.now().format(formatter);
            synchronized (listaFechas) {
                listaFechas.add(fechaActual);
            }
            System.out.println("Tarea1: Fecha y hora almacenada: " + fechaActual);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
};
```

1. Crea un objeto Runnable llamado tarea1.
2. En el método run(), obtiene la hora actual utilizando LocalDateTime.now() y la formatea como una cadena con el patrón "HH:mm:ss:S" (horas, minutos, segundos y milisegundos).
3. Almacena la hora actual en una lista estática llamada listaFechas utilizando synchronized para garantizar el acceso seguro en un entorno multihilo.
4. Imprime la hora almacenada en la consola.

Tarea 2: Verificar número primo y escribir en archivo

```
Runnable tarea2 = new Runnable() {
    @Override
    public void run() {
        try {
            String ultimaFecha;
            synchronized (listaFechas) {
                if (listaFechas.isEmpty()) {
                    return;
                }
                ultimaFecha = listaFechas.get(listaFechas.size() - 1);
            }
            int milisegundos = Integer.parseInt(ultimaFecha.split(":")[2]);
            if (esPrimo(milisegundos)) {
                escribirArchivo("Primos.txt", milisegundos);
            } else {
                escribirArchivo("NoPrimos.txt", milisegundos);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
};
```

1. Crea un objeto Runnable llamado tarea2.
2. En el método run(), verifica si la lista listaFechas está vacía. Si lo está, sale del método.
3. Obtiene la última hora almacenada en la lista y extrae los milisegundos de la cadena.
4. Verifica si los milisegundos son un número primo utilizando el método esPrimo().
5. Si es primo, escribe los milisegundos en un archivo llamado "Primos.txt". De lo contrario, escribe los milisegundos en un archivo llamado "NoPrimos.txt".

Método esPrimo

```
private static boolean esPrimo(int numero) {
    if (numero <= 1) return false;
    for (int i = 2; i <= Math.sqrt(numero); i++) {
        if (numero % i == 0) return false;
    }
    return true;
}
```

1. Recibe un número entero como parámetro.
2. Verifica si el número es menor o igual a 1. Si es así, devuelve false (no es primo).
3. Itera desde 2 hasta la raíz cuadrada del número (inclusive) utilizando un bucle for.
4. Dentro del bucle, verifica si el número es divisible por el valor actual del iterador (i). Si es así, devuelve false (no es primo).
5. Si el bucle se completa sin encontrar divisores, devuelve true (es primo).

Método escribirArchivo

```
private static void escribirArchivo(String nombreArchivo, int milisegundos) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(nombreArchivo, true))) {
        writer.write(milisegundos + "\n");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

1. Recibe el nombre del archivo y el número entero a escribir como parámetros.
2. Crea un objeto BufferedWriter para escribir en el archivo.
3. Abre el archivo en modo append (agregar al final del archivo existente) utilizando FileWriter.
4. Escribe el número entero en el archivo seguido de un salto de línea (\n) utilizando writer.write().
5. Cierra el flujo de escritura automáticamente utilizando try-with-resources.

scheduleAtFixedRate:

```
servicio.scheduleAtFixedRate(tarea1, 2, 2, TimeUnit.SECONDS);
servicio.scheduleAtFixedRate(tarea2, 2, 2, TimeUnit.SECONDS);
```

La tarea se ejecuta por primera vez después del retraso inicial (2 segundos). Después de la primera ejecución, la tarea se ejecuta nuevamente después del período especificado (2 segundos). El servicio sigue ejecutando la tarea cada período (2 segundos) hasta que se cancele o se detenga.

Finalmente un ejemplo de ejecución

```
Tarea1: Fecha y hora almacenada: 14:24:41:1
Tarea1: Fecha y hora almacenada: 14:24:43:1
Tarea1: Fecha y hora almacenada: 14:24:45:1
Tarea1: Fecha y hora almacenada: 14:24:47:1
Tarea1: Fecha y hora almacenada: 14:24:49:1
Tarea1: Fecha y hora almacenada: 14:24:51:1
Tarea1: Fecha y hora almacenada: 14:24:53:1
Tarea1: Fecha y hora almacenada: 14:24:55:1
Tarea1: Fecha y hora almacenada: 14:24:57:1
Tarea1: Fecha y hora almacenada: 14:24:59:1
Tarea1: Fecha y hora almacenada: 14:25:01:1
Tarea1: Fecha y hora almacenada: 14:25:03:1
Tarea1: Fecha y hora almacenada: 14:25:05:1
Tarea1: Fecha y hora almacenada: 14:25:07:1
Tarea1: Fecha y hora almacenada: 14:25:09:1
Tarea1: Fecha y hora almacenada: 14:25:11:1
```

4) Programe la ejecución de una tarea repetitiva, que cada 5 segundos chequee si en un directorio (carpeta/folder) a elección se ha creado un nuevo archivo. Para ello deberá iniciar la lectura y carga de los nombres de archivos para luego verificar si ha agregado uno nuevo, en caso de que un nuevo archivo sea creado se deberá mostrar por pantalla "Nuevo archivo [nombre_archivo], con tamaño [tamaño_archivo]". Deberá copiar un nuevo archivo a dicha carpeta para comprobar el proceso.

```java
public class Inciso04 {
    private static final String DIRECTORIO = "C:/Users/danie/OneDrive/Escritorio/PC/Archivo";
    private static Set<String> archivosExistentes = new HashSet<>();

    public static void main(String[] args) {
        ScheduledExecutorService servicio = Executors.newScheduledThreadPool(1);
        Runnable tarea = new Runnable() {
            @Override
            public void run() {
                chequearNuevosArchivos();
            }
        };

        cargarArchivosExistentes();
        servicio.scheduleAtFixedRate(tarea, 0, 5, TimeUnit.SECONDS);
    }

    private static void cargarArchivosExistentes() {
        File directorio = new File(DIRECTORIO);
        if (directorio.exists() && directorio.isDirectory()) {

            File[] archivos = directorio.listFiles();

            System.out.println("Contenido de la carpeta:");
            for (File archivo : archivos) {
                if (archivo.isFile()) {
                    System.out.println("Archivo: " + archivo.getName());
                } else if (archivo.isDirectory()) {
                    System.out.println("Carpeta: " + archivo.getName());
                }
                archivosExistentes.add(archivo.getName());
            }
        }
    }

    private static void chequearNuevosArchivos() {
        File directorio = new File(DIRECTORIO);
        if (directorio.exists() && directorio.isDirectory()) {
            for (File archivo : directorio.listFiles()) {
                if (!archivosExistentes.contains(archivo.getName())) {
                    archivosExistentes.add(archivo.getName());
                    System.out.println("Nuevo archivo [" + archivo.getName() + "], con tamaño [" + archivo.length() + " bytes]");
                }
            }
        }
    }
}
```

Clase Inciso04

1. La clase tiene un campo estático DIRECTORIO que especifica la ruta del directorio a monitorear.
2. Un campo estático archivosExistentes de tipo Set<String> almacena los nombres de los archivos que ya existen en el directorio.

Método main

1. Crea un servicio de ejecución programada (ScheduledExecutorService) con un solo hilo (newScheduledThreadPool(1)).
2. Define una tarea (Runnable) que se ejecutará periódicamente.
3. La tarea se llama chequearNuevosArchivos.
4. Se llama al método cargarArchivosExistentes para inicializar el conjunto de archivos existentes.
5. Programa la tarea para ejecutarse cada 5 segundos (scheduleAtFixedRate).

Método cargarArchivosExistentes

```
private static void cargarArchivosExistentes() {
    File directorio = new File(DIRECTORIO);
    if (directorio.exists() && directorio.isDirectory()) {

        File[] archivos = directorio.listFiles();

        System.out.println("Contenido de la carpeta:");
        for (File archivo : archivos) {
            if (archivo.isFile()) {
                System.out.println("Archivo: " + archivo.getName());
            } else if (archivo.isDirectory()) {
                System.out.println("Carpeta: " + archivo.getName());
            }
            archivosExistentes.add(archivo.getName());
        }
    }
}
```

1. Verifica si el directorio existe y si es un directorio.
2. Obtiene la lista de archivos y directorios en el directorio (listFiles()).
3. Recorre la lista e imprime el contenido del directorio.
4. Agrega los nombres de los archivos existentes al conjunto archivosExistentes.

Método chequearNuevosArchivos

```
private static void chequearNuevosArchivos() {
    File directorio = new File(DIRECTORIO);
    if (directorio.exists() && directorio.isDirectory()) {
        for (File archivo : directorio.listFiles()) {
            if (!archivosExistentes.contains(archivo.getName())) {
                archivosExistentes.add(archivo.getName());
                System.out.println("Nuevo archivo [" + archivo.getName() + "], "
                        + "con tamaño [" + archivo.length() + " bytes]");
            }
        }
    }
}
```

1. Verifica si el directorio existe y si es un directorio.
2. Recorre la lista de archivos y directorios en el directorio (listFiles()).
3. Verifica si cada archivo no está en el conjunto archivosExistentes.
4. Si es un archivo nuevo, lo agrega al conjunto e imprime un mensaje indicando el nombre del archivo y su tamaño en bytes.

Funcionamiento

1. Al iniciar el programa, se carga la lista de archivos existentes en el directorio.
2. Cada 5 segundos, se ejecuta la tarea chequearNuevosArchivos.
3. La tarea verifica si hay nuevos archivos en el directorio y los agrega al conjunto archivosExistentes.
4. Si se encuentra un nuevo archivo, se imprime un mensaje con su nombre y tamaño.

```
Contenido de la carpeta:
Archivo: NoPrimos.txt
Archivo: Nuevo Documento de texto.txt
Archivo: Primos.txt
```

```
Contenido de la carpeta:
Archivo: NoPrimos.txt
Archivo: Nuevo Documento de texto.txt
Archivo: Primos.txt
Nuevo archivo [Nuevo Documento de texto (2).txt], con tamaño [0
Nuevo archivo [UltimaPrueba.txt], con tamaño [0 bytes]
```

5) Se dispone del siguiente arreglo:

```
long[] vector = new long[] { 100477L, 105477L, 112986L,100078L,
165987L, 142578L };
```

Se desea hacer el siguiente cálculo de cada número con la siguiente función que demo
varios segundos:

```
static BigInteger M = new BigInteger("1999");
```

```
private static BigInteger compute(long n) {
  String s = "";
  for (long i = 0; i < n; i++) {
      s = s + n;
  }
  return new BigInteger(s.toString()).mod(M);
```

Para ello deberá definir un Pool de ejecución de tamaño 2 para ejecutar dichos cálculos.

```java
public class Inciso05 {
    static BigInteger M = new BigInteger("1999");
    public static void main(String[] args) {
        long[] vector = new long[]{100477L, 105477L, 112986L, 100078L, 165987L, 142578L};
        ExecutorService executor = Executors.newFixedThreadPool(2);
        for(int i=0; i<vector.length; i++) {
            Future<BigInteger> resultado = executor.submit(new TareaCalculo(vector[i]));
            try {
               System.out.println("Resultado: " + resultado.get());
            } catch (InterruptedException | ExecutionException e) {
               e.printStackTrace();
            }
        }
        executor.shutdown();
    }

    private static class TareaCalculo implements Callable<BigInteger> {
        private final long n;
        public TareaCalculo(long n) {
           this.n = n;
        }
        @Override
        public BigInteger call() {
            return compute(n);
        }
    }

    private static BigInteger compute(long n) {
        String s = "";
        for (long i = 0; i < n; i++) {
            s = s + n;
        }
        return new BigInteger(s.toString()).mod(M);
    }
}
```

Clase Inciso05

1. La clase tiene un campo estático M de tipo BigInteger que representa el divisor (1999).
2. El método main crea un arreglo de números enteros largos (long[]) y un pool de ejecución fijo de 2 hilos (ExecutorService).

Método main

```java
public static void main(String[] args) {
    long[] vector = new long[]{100477L, 105477L, 112986L, 100078L, 165987L, 142578L};
    ExecutorService executor = Executors.newFixedThreadPool(2);
    for(int i=0; i<vector.length; i++) {
        Future<BigInteger> resultado = executor.submit(new TareaCalculo(vector[i]));
        try {
           System.out.println("Resultado: " + resultado.get());
        } catch (InterruptedException | ExecutionException e) {
           e.printStackTrace();
        }
    }
    executor.shutdown();
}
```

1. Crea un arreglo de números enteros largos (vector).
2. Crea un pool de ejecución fijo de 2 hilos (ExecutorService).

3. Recorre el arreglo y para cada número:

- Crea una tarea de cálculo (TareaCalculo) con el número actual.
- Envía la tarea al pool de ejecución y obtiene un objeto Future que representa el resultado.
- Espera por el resultado y lo imprime.

4. Cierra el pool de ejecución.

Clase TareaCalculo

```
private static class TareaCalculo implements Callable<BigInteger> {
    private final long n;
    public TareaCalculo(long n) {
        this.n = n;
    }
    @Override
    public BigInteger call() {
        return compute(n);
    }
}
```

- Implementa la interfaz Callable, que permite devolver un resultado.
- Tiene un campo privado n que representa el número a procesar.
- El constructor inicializa el campo n.

Método call de TareaCalculo

- Llama al método compute con el número n.

Método compute

1. Crea una cadena vacía (s).
2. Repite n veces:
   - Concatena el número n a la cadena s.
3. Crea un BigInteger a partir de la cadena s.
4. Calcula el resto de dividir el BigInteger por M (1999) utilizando el método mod.

Funcionamiento

1. El programa crea un pool de ejecución con 2 hilos.
2. Divide el cálculo en tareas independientes (TareaCalculo) para cada número en el arreglo.
3. Cada tarea calcula el resto de dividir un número grande por 1999.
4. Los resultados se imprimen a medida que están disponibles.

```
Resultado: 988
Resultado: 765
Resultado: 932
Resultado: 1828
Resultado: 186
Resultado: 771
```