Instituto Tecnológico de Estudios Superiores de Monterrey

Lenguajes de Programación

Reporte de Proyecto Final

Valeria Guerra de la O | A01705316

Fernando Vargas Alvarez | A01066270

Victor Omar Molina Camarena | A01423485

Guillermo Carlos Espino Mateos | A01704354

Emilio Padilla Miranda | A01704889

Equipo | Rosetta Stone

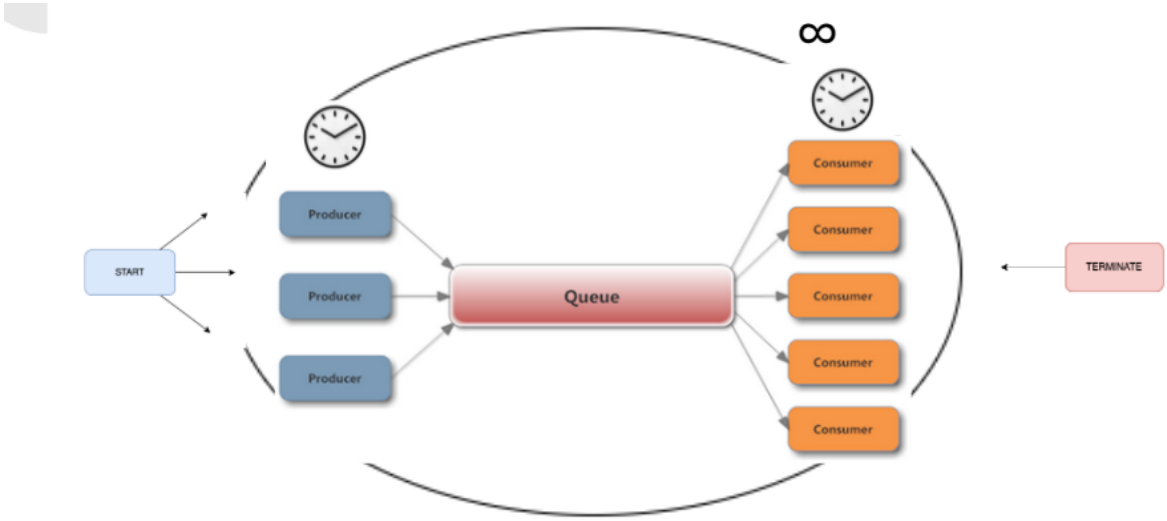
31/01/22

Descripción del proyecto:

El proyecto consiste en una interfaz gráfica que recibe un número determinado de productores, un número determinado de consumidores, un tiempo de intervalo entre productores, un tiempo de intervalo entre consumidores, el tamaño del buffer que servirá de puente entre los consumidores y productores, y los números del rango para generar los productos. Los productos se tratan de operaciones aleatorias con formato de Scheme, que pueden visualizarse en una de las dos tablas de la interfaz gráfica.

El mecanismo cuenta con un botón de inicio que ejecuta la creación de productores y consumidores, los cuales corren en cuanto son creados. La particularidad de estos productores y consumidores es que a pesar de ser hilos de ejecución individuales, todos comparten el mismo buffer. La ejecución puede detenerse si se hace clic en el botón de terminar. La descripción de cada uno de los mecanismos individuales se detallará en la contribución de cada integrante. Abajo un esquema de funcionamiento de nuestro proyecto a rasgos generales.

Esquema de funcionamiento:



Fernando:

Dentro del proyecto, además de estar apoyando a mis compañeros de equipo, mi participación se basó en implementar las validaciones de datos en la interfaz gráfica (GUI) e implementar la clase ArrayList para utilizarla como el buffer para los consumidores y productores del proyecto. Para la primer parte de esta participación alteré diferentes Spinners dentro de la interfaz para que: los valores de los tiempos de espera (marcados por milisegundos) pudieran recibir solo valores entre 1 y 10000, los valores tanto de los consumidores como de los productores pudieran poner valores entre 1 y 10, que el buffer pudiera aceptar de uno a cien procesos simultáneos y, finalmente, que los rangos de valores para las operaciones dentro de la GUI solo aceptarán valores de 1 a 10.Todos estos cambios fueron realizados dentro de la interfaz.

Además de lo ya mencionado, se importó e implementó la librería de ArrayList para darle cuerpo al buffer. Esto se hizo así pues fue la manera en la que se podían manejar diferentes cadenas de caracteres (*strings*) representando las operaciones que se deben realizar.

Finalmente le damos la opción al usuario de escoger el tamaño del buffer, que se traduce a cuantos procesos simultáneos puede hacer el proyecto. Esto se traduce a que el valor numérico que el usuario ingrese en la GUI dictará la cantidad de elementos válidos para el ArrayList del buffer.

```
import java.util.ArrayList;

public class Buffer {

    public boolean isActive;
    public int counter = 1;
    public int bufferLimit;
    ArrayList<String> bufferPool = new ArrayList<String>(bufferLimit);
```

Finalmente, creé el primer borrador para la función que regresara el resultado de la operación de scheme. Este recibía la operación, la recorría caracter por caracter y obtenía los datos numéricos y el operador para posteriormente realizar la operación en java. A su vez también implementé un contador interno en el buffer y otras dos funciones para aumentar y decrementar el contador dentro del buffer para verificar que los elementos en este buffer estén entrando y saliendo y comprobar que no excede el tamaño del mencionado buffer. Estas funciones posteriormente fueron aplicadas y corregidas por mi compañero **Emilio** dentro de los hilos.

```
public String translate(String mensaje){
        int a = 0;
        int b = 0;
        char op = 0;
        String response;
        String numbers = "0123456789";

        for(int i = 0; i < mensaje.length(); i++){
            char c = mensaje.charAt(i);
            switch(c){
                case ' ':
                    continue;
                case '(':
                    continue;
                case '-':
                    op = c;
                case '+':
                    op = c;
                case '/':
                    op = c;
```

```
                case '*':
                    op = c;
                default:
                if(numbers.contains(c)){
                    if(a != 0){
                        a = c;
                    }else{
                        b = c;
                    }}
            }
            response = a + op+ b;
        }
        return response;
    }
```

```
public int counter = 0;

     public int countProductor(){
        return this.counter += 1;
    }
     public int countConsumidor(){
        return this.counter -= 1;
    }
```

Omar:

Mi participación en el proyecto, fue la implementación de una función la cuál simulará una función de scheme dentro de java. Está función se pudo realizar correctamente mediante un análisis de las clases, ya que el entender cómo es que se tienen que enviar los datos desde una clase al GUIFrame fue confuso en un principio.

Para empezar se creó una clase **String** llamada scheme la cual, su principal función es crear un nuevo String, copiando el que recibe como parámetro a la par se incluirá el modificador **synchronized,** la cual permite que un hilo o algún proceso ejecute la función y hasta que este termine, ningún otro podrá acceder hasta que el primero acabe.

```
synchronized String scheme(int n, int m){
```

De igual forma se usó esta librería para que se pueda utilizar la función Random

```
import java.util.Random;
```

Dentro de la clase se obtendrán los valores n y m, posteriormente se hará una comparación la cuál n tendrá que ser estrictamente menor que m, después se crearán los rangos de forma aleatoria por medio de los valores n y m que son mis límites y por último se retornará un variable tipo string la cuál es una representación de una función de scheme, cabe recalcar que todo esto se hace de forma aleatoria, desde los símbolos hasta los números.

```
    int n_1, m_1;
    String x;

    int RangoN =  n;
    int RangoM =  m;

     if(RangoN < RangoM){

         Random random = new Random();


         n_1 = random.nextInt((RangoM - RangoN) + 1) + RangoN;
         m_1 =  random.nextInt((RangoM - RangoN) + 1) + RangoN;

         String setOfCharacters = "*/+-";
         int randomInt = random.nextInt(setOfCharacters.length());
         char randomChar = setOfCharacters.charAt(randomInt);
         x = "(" + randomChar +" " + n_1 + " " + m_1 + ")" ;

     }
     else{
         x = " El primer valor n tiene que ser menor que m.";
     }

     return x;
 }
```

Posteriormente para que esta clase reciba los parámetros adecuados puestos por el usuario, se tiene que llamar desde el GUIFrame.java los valores obtenidos desde la interfaz desplegada. Esta información se obtiene mediante el uso de .getValue(); el cual sirve para obtener los valores y posteriormente **castearlos,** básicamente es el proceso para transformar una variable primitiva de un tipo a otro.

```
int x  = (Integer) n.getValue();
int y  = (Integer) m.getValue();
```

Y para que la función sea desplegada dentro del producer se tiene que mandar a llamar a la función scheme dentro de run(), para que vaya desplegando mediante la cantidad de producers que estén activos.

```
@Override
public void run() {
    System.out.println("Running Producer " + this.id + "...");
    String product = "";
    while (this.buffer.isActive) {
        if(this.buffer.counter <= this.buffer.bufferLimit) {

//          for(int i=0 ; i<10 ; i++) {
            product = scheme(n,m);
            this.buffer.produce(product, waitTime);
            //System.out.println("Producer produced: " + product);
            this.buffer.print("Producer " + this.id + " produced: " + product);
            this.gui.addProducts(this.id, product);
```

Emilio:

Mi participación dentro del proyecto final se enfocó en modificar el comportamiento de la aplicación para poder manejar múltiples hilos en lugar de uno solo. Para ello, ligué el número de consumidores y productores obtenido a través del input de la GUI, y el código generaba esa misma cantidad de hilos. Es decir, uno por cada consumidor y cada productor.

```
public class GUIFrame extends javax.swing.JFrame {
      Buffer buffer;


      System.out.println("OPERATIONS ARE ABOUT TO GET SOLVED!! ");

      for(int i=1 ; i <= producers ; i++) {
          Producer producer = new Producer(this.buffer, this, i, prodWaitMs, x, y);
          producer.start();
          System.out.println("Producer "+ i + " created");
      }

      for(int i=1 ; i <= consumers ; i++) {
          Consumer consumer = new Consumer(this.buffer, this, i, consWaitMS);
```

```
            consumer.start();
            System.out.println("Consumer "+ i + " created");
        }
}
```

También, guardé los valores de la GUI, tales como el número de productores, el número de consumidores, los milisegundos a esperar para cada uno, y el tamaño del buffer.

```
this.buffer = new Buffer((Integer) bufferQtty.getValue(), this);
int producers = (Integer) producerQtty.getValue();
int consumers = (Integer) consumerQtty.getValue();
int prodWaitMs = (Integer) prodWait.getValue();
int consWaitMS = (Integer) consWait.getValue();
```

De igual manera, modifiqué el comportamiento de la clase Buffer para que las funciones consume y produce pudieran manejar el ArrayList del buffer de manera correcta, funcionando en un bucle infinito hasta que un finish pare la ejecución, y solo dejando producir si es que el buffer no esta lleno, así como solo dejar consumir si es que hay productos dentro, caso contrario, se mantienen en un wait() dentro de un bucle. También imprimí en consola los valores dentro del buffer. Otra funcionalidad necesaria para los hilos fue notificar a todos (usando notifyAll()) para despertar a los hilos que estuvieron esperando una vez los productos comenzaron a producirse.

```
public class Buffer {
   public boolean isActive;
   public int counter = 1;
   public int bufferLimit;
   Buffer(int bufferLimit, GUIFrame gui) {
       this.isActive = true;
       this.bufferLimit = bufferLimit;
   }
   synchronized String consume(int waitTime) {
       if(this.bufferPool.isEmpty()) {
           try {
               wait(); // wait(); Esperar un tiempo indeterminado para poder consumir
           } catch (InterruptedException ex) {
               Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE, null, ex);
           }
       }
       String removed = "||||No operations to be resolved available||||";
       if( !this.bufferPool.isEmpty()){
           removed = this.bufferPool.remove(0);
       }
       notifyAll();

       return removed;
```

```
    }
       synchronized void produce(String product, int waitTime) {
        if(!this.bufferPool.isEmpty()) {
            try {
                wait();// wait(); Esperar un tiempo indeterminado para poder producir
            } catch (InterruptedException ex) {
                Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
        this.bufferPool.add(product);
        notifyAll();
    }
       void printBuffer() {
        System.out.println("\n.................");
        this.bufferPool.forEach(item ->{
            System.out.println(item);
        });
        System.out.println("\n.................");
    }

    synchronized int incrementCount(){
        return this.counter += 1;
    }
    synchronized int decrementCount(){
        return this.counter -= 1;
    }

    synchronized void stopProducerConsumer() {
        this.isActive = false;
        this.count = 0;
    }
```

Por otro lado, modifiqué las clases producer y consumer para que una vez iniciados los procesos a través del botón de **Inicio** (que también ligué a una función para que empezara el proceso), esto corrieran sin parar, produciendo y consumiendo siempre y cuando el contador fuera mayor a cero (caso del contador) y el contador fuera menor que el límite del buffer marcado por el usuario (caso del productor). Siguiendo, agregue la funcionalidad para que el botón de inicio mencionado se inhabilite una vez comienza a correr el proceso

```
public class Consumer extends Thread {
    Buffer buffer;
    int id;
    int waitTime;

    Consumer(Buffer buffer, int id, int waitTime) {
        this.id = id;
```

```
        this.buffer = buffer;
        this.waitTime = waitTime;
    }
       @Override
    public void run() {
        System.out.println("Running Consumer " + this.id + "...");
        String product;

        while (this.buffer.isActive) {
            if(this.buffer.counter > 0) {
                product = this.buffer.consume(this.waitTime);
                this.buffer.print("Consumer " + this.id + " consumed: " + product);
                try {
                    Thread.sleep(waitTime);
                } catch (InterruptedException ex) {
                    Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
                }
                this.buffer.decrementCount();
            }
        }
    }
}

public class Producer extends Thread {
    Buffer buffer;
    int id;
    int waitTime;

    Producer(Buffer buffer, int id, int waitTime) {
        this.id = id;
        this.buffer = buffer;
        this.waitTime = waitTime;
    }
    @Override
    public void run() {
        System.out.println("Running Producer " + this.id + "...");
        String product = "";
        while (this.buffer.isActive) {
            if(this.buffer.counter <= this.buffer.bufferLimit) {
                this.buffer.produce(product, waitTime);
                this.buffer.print("Producer " + this.id + " produced: " + product);

                try {
                    Thread.sleep(waitTime);
                } catch (InterruptedException ex) {
```

```
                    Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
                }
                this.buffer.incrementCount();
            }
            else {
                //Buffer empty notification for monitoring. Commented for clarity purposes.
                //System.out.println("Your buffer is empty.");
            }
        }
    }
}
```

Por último, agregué una función que detendría el proceso en ejecución, y lo ligué a un botón de **terminar**. Este botón de terminar también vuelve a habilitar el botón de **inicio** para comenzar el proceso de nuevo.

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
        this.buffer.stopProducerConsumer();
        System.out.println("EXECUTION TERMINATED");
        this.isProcessNotStarted = true;
        jButton1.setEnabled(true);

    }
```

Valeria:

Para la realización de este proyecto lo primero que realicé fue un análisis de qué es lo que necesitábamos para poder concluir el proyecto; una vez que comenzamos a separar por secciones el trabajo de mis compañeros me puse a investigar cómo es que se realizaba la lógica de interacción entre la tabla que tenía que visualizar la información de tareas, además de como funcionaba la progress bar que se muestran con las siguientes líneas de código:
Para las tablas:

```
DefaultTableModel model = (DefaultTableModel)jTable1.getModel();
DefaultTableModel model2 = (DefaultTableModel)jTable2.getModel();
        model.addRow(new Object[]{id, product});
        model2.addRow(new Object[]{id, product,result});
```

Para la progress bar que se va aumentando el valor cada que se necesita con una acción o una función) :

```
 int valor= 0;
valor ++;
jProgressBar1.setValue(valor);
```

Después de realizar las pruebas iniciales, además de estar apoyando a mis compañeros, realicé la función para convertir el producto que teníamos en scheme, que era un string generado por nosotros en el resultado mediante un switch regresando el resultado de la operación.

```
public double resultado(String product){
        double res =0;
        char ch  = product.charAt(1);
        char number  = product.charAt(3);
        char number2  = product.charAt(5);
        int num= Integer.parseInt(String.valueOf(number));
        int num2= Integer.parseInt(String.valueOf(number2));

        switch (ch) {
            case '/':
                res= (double)num/num2;
                break;
            case '*':
                res= num*num2;
                break;
            case '-':
                res= num-num2;
                break;
            case '+':
                res= num+num2;
                break;
        }
        return res;
    }
```

Una vez realizada la lógica para poder visualizar la tabla de tareas y se retiraba de ella agregue un contador para poder visualizar de cantidad de tareas que pasaron de ser realizadas a terminadas, esto se agregó en la función removeProducts(int id, String product), en donde el contador iniciaba en 0, hasta que cada tarea era realizada entonces se sumaba al contador y se visualizaba mediante jSpinner4.setValue(counter); .

Realicé en conjunto con mis compañeros, el análisis e implementación de cómo poner la función de la creación de las operaciones de Scheme dentro del producto para así poder pasarlo al buffer y llamarlo desde el consumidor.

Adicional a esto también realicé la lógica para poder mostrar de manera progresiva, esto iniciando la variable valor completo y cada que una tarea estaba en pendiente, se le quita un número al valor y se le setea a la variable en la función addProducts() y cada que pasa se ser una tarea pendiente a una realizada, se le suma el valor restado en addProducts() en la función removeProducts().

```
 int valor= 100;
valor = valor - 1;
       jProgressBar1.setValue(valor);
valor = valor + 1;
                   jProgressBar1.setValue(valor);
```

Por último, terminé actualizando que la función addProducts() para que se pudiera generar y llamar desde el buffer y no desde el Product.java, ya que su idea es una ventana a cómo es el buffer y cuales son las tareas que están dentro de él.

Guillermo:

Mi contribución al proyecto consistió en las actividades que se describirán a continuación. Primero, una vez que los compañeros correspondientes del equipo hubieran terminado la creación y ejecución de los hilos para mostrar en consola todo lo que los productores producían y también lo que cada consumidor consumía, primero retiré una impresión que nublaba la visibilidad en el monitoreo. Subsecuentemente dediqué gran cantidad de tiempo a arreglar el merge de las ramas que contenían los códigos finales y funcionales, ya que por haberlo trabajado sin excluir algunos archivos, la forma del JFrame se corrompió y fue necesario generar una nueva para que esto no causara daños posteriores y errores mayores (mucho aprendizaje para proyectos posteriores).

Después, me encargué de construir la función que permitiera a la GUI añadir a la Tabla 1 las filas que contienen el ID del productor con su producto. Esta función fue addProducts(), y para que funcionara también fue necesario añadir la GUI como atributo del Productor, y que el pase de datos fuera más fluido. Originalmente había pensado que lo mejor era usar esta función desde cada productor, sin embargo, la llamada de cada productor a la función a veces resultaba en hilos asíncronos y diversos errores que fueron solucionados aplicando la función desde el Buffer al momento de producir un producto. A continuación la función addProducts desde Buffer.java:

```
 public void addProducts(int id, String product,int div){
       //System.out.println("div "+ div+ " ");

       DefaultTableModel model = (DefaultTableModel)jTable1.getModel();
       model.addRow(new Object[]{id, product});
       valor = valor - 10;
       jProgressBar1.setValue(valor);
    }

Producer.java
```

```
@Override
    public void run() {
        System.out.println("Running Producer " + this.id + "...");
        String product = "";
        while (this.buffer.isActive) {
            product = scheme(n,m);
            this.buffer.produce(this.id, product);
            this.buffer.print("Producer " + this.id + " produced: " + product);

            try {
                Thread.sleep(waitTime);
            } catch (InterruptedException ex) {
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
```

De igual manera construí la función removeProducts() que se encarga de buscar en la primera tabla el producto que está siendo consumido y escribirlo en la segunda tabla junto con el id del consumidor y el resultado de esta operación en Scheme. Esta función la construí de la mano de mi compañera Valeria, ya que ella se encargaba directamente de supervisar el funcionamiento de la barra de progreso y cómo se relacionaba con el progreso de las tablas. Asimismo fue necesario construir al Consumidor con la GUI para que pudiera acceder a la función. La función originalmente se llamaba desde Consumidor y resultaba el mismo error asíncrono, por lo que la llamada a la función fue movida a Buffer.java.

Función removeProducts usada al momento de consumir desde Buffer.java:

```
public void removeProducts(int id, String product,int div){
        counter++;
        DefaultTableModel model = (DefaultTableModel)jTable1.getModel();
        DefaultTableModel model2 = (DefaultTableModel)jTable2.getModel();
        try{
            int size = model.getRowCount();
            for(int i = 0; i<=size; i++){
                if(product == model.getValueAt(i, 1)){
                    model.removeRow(i);
                    valor = valor + 10;
                    jProgressBar1.setValue(valor);
                    jSpinner4.setValue(counter);
                    result=resultado(product);
                    model2.addRow(new Object[]{id, product,result});
                }
            }
```

```
            if(model.getRowCount() == 0){
                System.out.println("Nothing to be removed.");
            }
        }catch(Exception e){
        }
    }
```

Finalmente hicimos unas cuantas correcciones en cuanto a la lógica en el orden de las llamadas de las funciones, y un pequeño error que no permitía resetear la tabla 2. Probamos el código en repetidas ocasiones con los valores extremos (más grandes y más pequeños) para observar el comportamiento del programa, y notamos que probablemente por la naturaleza de los hilos y su ejecución paralela, a veces no respetaban el tamaño delimitado por el buffer. No obstante los valores se veían acotados a la delimitación en la mayoría de las ocasiones. Después de la última fase de prueba que estuvo en manos de Valeria y de mí, hicimos el merge final del código solución.

Conclusión

Para concluir, podemos decir que aunque el proyecto parecía sencillo al principio para la cantidad de personas que lo conformamos, que en nuestro caso somos 5, sí tuvimos que afrontar diferentes dificultades que resultaron complicando más de lo esperado. Para empezar, debido a que no teníamos un completo entendimiento del proyecto, caímos en una mala división de tareas, resultando en que algunos compañeros trabajarán un poco más que otros. De igual manera, no manejamos las tareas adecuadamente, es decir, no identificamos correctamente el alcance de cada tarea, lo que resultó en que una tarea finalizada dejará otras tareas con dificultades que no corresponden per se a esas otras tareas.

Por otro lado, también podemos concluir que el proyecto sirvió para refrescar conceptos de hilos, sincronización, Java, y el uso del IDE NetBeans, siendo temas que vimos iniciando la carrera y ahora que estamos por terminar, seguramente no tendríamos otra oportunidad para volver a disfrutar de las complejidades y atajos que ofrece Java al mundo de la programación.

Presentación utilizada:

https://docs.google.com/presentation/d/1VkC-ja-lE8T1G07mXznJ3Ujo8RccID88n3rWvs1vbtU/edit?usp=sharing