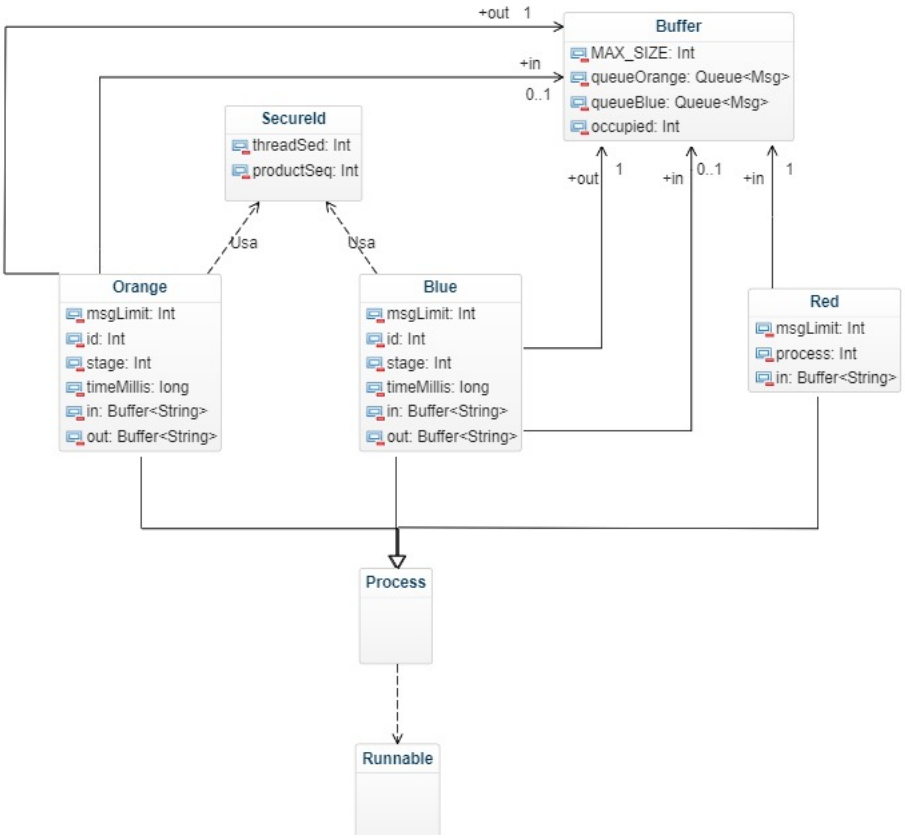
Pedro Luis Lobato Barros
202012490
p.lobato
Jaime Andres Torres Bermejo
202014866
j.torres16
Sofia Torres Ramírez
202014872
s.torres21

Caso 1
Infraestructura Computacional -
Universidad de los Andes

2023-02-23

1. Estructura del Proyecto

1.1. UML



Por medio de este diagrama UML se puede entender a grandes rasgos el funcionamiento del programa. Iniciando por la clase `Process` podemos ver que esta clase extiende de `Runnable` en pro de seguir el lenguaje manejado en el caso. Siguiendo con esta idea se aprecian las clases `Orange`, `Blue` y `Red` que representan los diferentes procesos que puede tener el programa, para eso cada uno implementará la interfaz `Process`.
Tal como lo evidencia el diagrama, las clases que implementan a `Process` tienen el objetivo de interactuar con procesos (Creando, transformando o enviándolos). Sin embargo, su funcionamiento interno es diferente y este será especificado en la sección 1.3.
Los atributos que los tipos de procesos comparten son:

- `String msgLimit`: El número de productos o mensajes que se deben crear por cada tipo de proceso y que por ende se deben imprimir al finalizar el programa. Este atributo es estático, por lo tanto, está subrayado.
- `int id`: El identificador de cada proceso creado.*
- `int stage`: El identificador de la etapa en la cual se encuentra el proceso creado.*
- `long timeMillis`: El tiempo que debe dormir el proceso para simular la modificación del mensaje/-producto.*

- `Buffer in` : El buzón de entrada del proceso (explicado más adelante).
- `Buffer out` : El buzoón de salida del proceso (explicado más adelante).*
- `int process` : La cantidad de procesos por etapa.

* `Red` no posee estos atributos marcados por su posición al final del todo y por el hecho que no necesita transformar el mensaje.
• `Red` es el único tipo de proceso con este atributo; atributo necesario para saber cuando finaliza la ejecución.
Con respecto a la clase `Buffer` podemos ver que tiene los siguientes atributos:

- `int MAX_SIZE` : La capacidad del buffer.
- `Queue queueOrange` : Una cola que almacena los productos/mensajes que fueron creados y modificados por procesos naranjas.
- `Queue queueBlue` : Una cola que almacena los productos/mensajes que fueron creados y modificados por procesos azules.
- `int occupied` : La cantidad de productos/mensajes que hay en el buffer.

Tanto el proceso naranja (Orange) como el azul (Blue) tienen dos relaciones con el Buffer. El hecho que la relación con el buzón de entrada posea una cardinalidad de 0..1 está relacionado a la etapa del proceso en si, si se es la etapa inicial no hay producto que "sacar" pero si un mensaje que producir. Para el resto de etapas ya hay un producto que "sacar".
Por otro lado, para todo proceso azul o naranja siempre existirá un buzón de salida en el cual se colocan los mensajes ya modificados o recién generados para que un proceso de la siguiente etapa pueda interactuar con el producto.
Sin embargo, el proceso rojo no funciona igual debido a que este proceso solo tendrá interacción con el último buzón, del cual sacará la totalidad de mensajes para luego imprimirlos.

1.2. Main.java

1.2.1. La interfaz `Process`

Su único objetivo es el de estipular que los otros tipos de procesos son... Procesos, en otras palabras es una decisión tomada para que se apegue al lenguaje del caso.

1.2.2. La clase `SecureId`

Su objetivo es el de otorgar las ids necesarias para seguir el proceso de los mensajes y procesos. Para cumplir esa función están sus atributos estáticos:

- `int threadSeq` : Para identificar a los hilos (procesos).
- `int productSeq` : Para identificar a los mensajes.

1.2.3. La clase `Main`

La principal funciíon de la clase `Main`, además de correr el programa, es preparar al resto del programa para su funcionamiento óptimo.

Una de las primeras cosas que van a saltar a la vista es el uso de variables definidas con el estado de `final` dentro de la propia lógica de java. Con esto, se pretende hacer que las variables definidas sean inmutables y no puedan ser alteradas una vez definidas ni por el usuario ni por ninguno de los procesos que el programa corra. Las variables que caen dentro de este conjunto inmutable son:

- `int MESSAGE_NUM` : Define el número de mensajes a enviar.
- `int PROCESS_NUM` : Define el número de procesos del programa.
- `int BUFFER_CAP` : Define el límite del Buffer
- `int BUFFER_NUM` : Define el número de Buffers
- `int STAGE_NUM` : Define el número de etapas por las que pasarán los procesos

Estas constantes son luego utilizadas para la creación de los objetos que serán usados en el programa, y conectará la lógica de esos nuevos objetos creados.

1.3. Estructura de los Procesos

1.3.1. Blue

```
class Blue implements Process {
    private static int msgLimit;
    private final int id, stage;
    private final long timeMillis = new Random().nextInt(bound: 451) + 50;
    private final Buffer<String> in, out;

    public Blue(final int id, final int stage, final Buffer<String> in, final Buffer<String> out) {
        this.id = id;
        this.stage = stage;
        this.in = in;
        this.out = out;
    }

    public static void setMsgLimit(int msgLimit) {Blue.msgLimit = msgLimit;}
```

En la parte inicial del código se puede ver la definición de los atributos de la clase `Blue`, su método constructor y el setter para el atributo estático `msgLimit`. Luego de esto podemos ver el método run del proceso, en donde, si la etapa del proceso creado es igual a 0, correrá el método `runZero( )`; de lo contrario, correrá el método `runNonZero( )`. Esto se hace debido a que, si el proceso pertenece a la etapa cero, quiere decir que aún no se han producido los productos/mensajes, por ende, toca crearlos. Si la etapa es diferente de 0, ya están generados los mensajes así que el proceso debería obtenerlos del buffer de entrada.

Método `runZero( )`

```
private void runZero() {
    var context = new Object() {
        int counter = 0;
    };
    while (context.counter < msgLimit) out.synchronizedBlue(() -> {
        while (out.isFullBlue()) {
            try {
                out.waitBlue();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        out.putBlue("B.Thread[" + SecureId.nextPID() + "]ID" + id + "-Stage" + stage + "(" + context.counter + ")");
        ++context.counter;
        out.notifyBlue();
    });

}
```

En este método se vale de un objeto anónimo para poder interactuar adecuadamente con la expresión lambda que se encuentra más adelante, dicho objeto contiene al atributo `counter` que servirá para saber cuando se ha llegado al límite de los mensajes con los que debe interactuar el proceso. Prosiguiendo se destaca un while que verifica lo mencionado en la explicación anterior relacionada al límite de mensajes para estar inmediatamente continuado por un método que se vale de la expresión lambda o función anónima. (La forma en que se maneja será explicada más adelante). Siguiendo se encuentra la parte del código encargada de asignar la espera pasiva para los procesos azules y la creación de los productos mientras haya espacio para finalizar ubicando el mensaje en el buzón, aumentando el contador de `counter` y notificando a algúno de los hilos que interactúan con ese buzón.

Método `runNonZero( )`

```java
private void runNonZero() {
    var context = new Object() {
        String msg;
        int counter = 0;
    };
    while (context.counter < msgLimit) {
        in.synchronizedBlue(() -> {
            while (in.isEmptyBlue()) try {
                in.waitBlue();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            context.msg = in.getBlue();
            in.notifyBlue();
        });
```

Por otra parte, este método tiene un comportamiento similar al anterior: Para la sección de obtener hay una variación representada sobre **que** objeto se aplica la sincronización y la condición ahora es para comprobar si está vacío.
Por otra parte, la sección de poner solo presenta la variación que pone el mensaje ya transformado en lugar de uno creado. Aclarando, se ejecuta un ciclo en donde, si la cola de mensajes azules del buffer de salida está lleno, se notifica que se está intentando agregar un nuevo mensaje a la cola y luego espera a que haya espacio en ella para poder agregar el mensaje. Cuando la cola no está llena se agregará el mensaje modificado y el contador de mensajes procesados aumenta en uno. Luego de esto, se notifica a todos los que estén esperando sobre esta cola por si algún proceso de la siguiente etapa estaba esperando por él. No se realiza un `notify( )` debido a que es posible que se notifique a un proceso que ya haya terminado de procesar sus mensajes.
Nota: Para cumplir lo anterior se sigue valiendo de la expresión lambda.
Entre ambas secciones mencionadas se ejecuta la transformación necesaria del mensaje agregándole esta nueva etapa por la que pasó y luego el proceso se duerme por un tiempo aleatorio simulando el tiempo que toma dicha transformación.

1.3.2. Orange

```java
/** 🟠 Thread */
@SuppressWarnings("BusyWait") class Orange implements Process {
    private static int msgLimit;
    private final int id, stage;
    private final long timeMillis = new Random().nextInt(bound: 451) + 50;
    private final Buffer<String> in, out;

    Orange(final int id, final int stage, final Buffer<String> in, final Buffer<String> out) {
        this.id = id;
        this.stage = stage;
        this.in = in;
        this.out = out;
    }

    public static void setMsgLimit(int msgLimit) {Orange.msgLimit = msgLimit;}
```

En resumen se sigue la estructura del azul, pero con la variación que en la sección de la expresión lambda ahora usa dos fragmentos de código para separar la parte no sincronizada de la sincronizada (Mejor explicado en la sección dedicada a la clase `Buffer`).
Ahora ahondando más: En la parte inicial del código se puede ver la definición de los atributos de la clase Orange, su método constructor y un método para darle valor al atributo `msgLimit`. Luego de esto podemos ver el método run del proceso en donde, si la etapa del proceso creado es igual a 0, correrá el método `runZero( )`, de lo contrario correrá el método `runNonZero( )`. Esto se hace debido a que, si el proceso pertenece a la etapa cero, quiere decir que aún no se han creado los productos/mensajes, por ende, toca crearlos. Si la etapa es diferente de 0, ya están creados los mensajes así que el proceso tendría que obtenerlos del buffer de entrada.

```
private void runNonZero() {
    var context = new Object() {
        String msg;
        int counter = 0;
    };
    while (context.counter < msgLimit) {
        in.synchronizedOrange(() -> {
            while (in.isEmptyOrange()) Thread.yield();
        }, () -> {
            context.msg = in.getOrange();
        });
```

En la primera parte del método `runNonZero( )` se crea un objeto anónimo `context` el cuál tendrá dos atributos `msg` y `counter`, en donde `msg` será el mensaje sacado del buffer de entrada y `counter` será el número de mensajes que se han procesado. Teniendo en cuenta esta información, luego se ejecutará un ciclo mientras todavía no se hayan procesado todos los mensajes azules. Siguiente a esto se llama a la método `in.synchronizedOrange( )` la cuál permite la distinción entre bloques, en el primer bloque de instrucciones aún no se sincroniza respecto a la cola de mensajes naranjas del buffer, mientras que en el segundo sí exista esta sincronización. En el primer bloque se evalúa si la cola de mensajes naranjas está vacía. Si esto es cierto, se ejecuta la instrucción `Thread.yield( )` la cual generará que el proceso ceda el procesador y lo vuelva a solicitar inmediatamente para volver a consultar hasta que ya no esté vacía. Es aquí en donde se puede evidenciar el comportamiento de espera semi-activa por parte del proceso naranja y también se identifica la importancia de dejar la instrucción del `yield` por fuera de la sincronización del recurso para no generar un `deadlock`. Cuando la cola de mensajes naranjas ya no está vacía se ejecuta el segundo bloque del método `in.synchronizedOrange( )` lo que quiere decir que en esta parte del código sí se estará sincronizando respecto a la cola de mensajes naranjas. En esta parte del código se tomará un mensaje de la cola de mensajes naranjas para luego liberar al recurso. Es decir, ya no se sincroniza la cola.

En la siguiente parte del código se realiza la transformación necesaria del mensaje agregándole esta nueva etapa por la que pasó y luego el proceso se duerme por un tiempo aleatorio simulando dicha transformación. Luego de modificar el mensaje es necesario agregarlo al buffer de salida (out) debido a esto, primero se llama a la función `out.synchronizedOrange( )` (Cuyo funcionamiento ya ha sido explicado). El primer bloque de este método será ejecutar un ciclo en donde si la cola de mensajes naranjas del buffer de salida está llena, se hace `Thread.yield( )` en donde el proceso naranja cederá el procesador y lo volverá a solicitar inmediatamente hasta que la cola ya no esté llena. Luego, en el segundo bloque de en el cual se sincronizará la cola de mensajes naranjas del buffer de salida, cuando la cola no esté llena se agregará el mensaje modificado y el contador de mensajes procesados aumentará en uno.

```
private void runZero() {
    var context = new Object() {
        int counter = 0;
    };
    while (context.counter < msgLimit) {
        out.synchronizedOrange(() -> {
            while (out.isFullOrange()) Thread.yield();
        }, () -> {

            out.putOrange(
                    "O.Thread[" + SecureId.nextPID() + "]ID" + id + "-Stage" + stage + "(" + context.counter + ")");
            ++context.counter;
        });
    }

}
```

Por último se tiene el método `runZero( )` en donde se crea un objeto anónimo `context` el cual tiene un atributo llamado `counter` que cuenta la cantidad de productos/mensajes creados. Luego de esto se ejecuta un ciclo mientras el número de mensajes producidos sea menor al número de mensajes a crear. Mientras esto suceda se hará uso del método `out.synchronizedOrange( )` (Con su funcionamiento ya explicado previamente) en donde el primer bloque de la función se evaluará si la cola de mensajes naranjas del buffer de salida está llena, de ser así se ejecuta la instrucción `yield` hasta que la cola de mensajes naranjas no esté llena. En el segundo bloque de la función, cuando esta cola ya no está llena, se crea el mensaje, se agrega a la cola y luego se aumenta el contador de mensajes creados en uno.

1.3.3. Red

```
class Red implements Process {
    private static int msgLimit, process;
    private final Buffer<String> in;

    Red(final Buffer<String> in) {
        this.in = in;
    }

    public static void setMsgLimit(int msgLimit) {Red.msgLimit = msgLimit;}

    public static void setProcess(int process) {Red.process = process;}
```

En la parte inicial del código se puede ver la definición de los atributos de la clase Red, y los métodos para darle valor a estos atributos. Adicionalmente, se puede la inicialización del método `run( )` y dentro él la creación de variables de control como `msg` el cual es un arreglo de tamaño igual al número total de mensajes creados. También se crean las variables `sentinelO` y `sentinelB` las cuales aseguran que no se lean más de `msgLimit` mensajes provenientes de procesos naranjas y azules correspondientemente.

Luego de esto se ejecuta dos ciclos los cuales ayudarán a obtener los mensajes y se ordenen por orden de creación para luego ser impreso, primero se ejecuta un ciclo para obtener y ubicar correctamente los mensajes naranjas y luego se ejecuta otro ciclo similar pero con los procesos azules. Este primer ciclo se ejecutará siempre y cuando exista al menos un proceso(el cual se sabe que es naranja debido al orden de creación de los mismo en el main) y mientras no se haya leído la totalidad de mensajes naranjas limitada por el límite de mensajes naranjas creados, se sincronizará la cola de mensajes naranjas del buffer para que este recurso le pertenezca al proceso rojo mientras se cumplan las condiciones mencionadas; debido a esto, ningún otro recurso puede modificar esta cola y los datos que se están leyendo en el momento son consistentes. Luego de esto se ejecuta un while el cual se dará mientras la cola de mensajes naranjas no esté vacía. Aquí se evidencia la espera activa por parte del proceso rojo debido a que, si la cola está vacía, consultará permanentemente (sin ceder el procesador) hasta que ya no lo esté para así tomar un mensaje de esta, analizarlo y adicionarlo al arreglo de mensajes `msg` en el orden correspondiente a su id (Con ayuda de la función `between` la cual se explicará más adelante) y luego aumentar el counter de mensajes leídos para comprarlo con `msgLimit` y así volver a evaluar `sentinelO`.
Al finalizar este ciclo, se habrán leído todos los mensajes del buffer de mensajes naranjas que se encontraban disponibles en ese momento y se habrán almacenado en el arreglo de mensajes msg. Esto mismo sucede en el siguiente ciclo dedicado a los procesos azules, pero en su lugar se verifica que la cantidad de procesos corriendo sea mayor a 1 (`process >1`), en donde en ese caso se sabrá que al menos uno de los procesos corriendo es azul, y se ejecutan las mismas instrucciones mencionadas anteriormente.
Cabe añadir que en ambos ciclos hay una línea que crea un marco el cual delimita los mensajes modificados finales que mostrarán al usuario (Decisión tomada con fines estéticos).

```
private int between(String s) {
    return Integer.parseInt(s.substring(s.indexOf(str: "[") + 1, s.indexOf(str: "]")));
}
```

Por último, el método `between( String s )` permite encontrar el id del mensaje que entra por parámetro, esto se hacer por medio del método substring para extraer una subcadena que comienza justo después del corchete de apertura y termina justo antes del corchete de cierre. (Esto es permitido de acuerdo a la estructura escogida para las transformaciones y producciones de los diversos mensajes). Esta subcadena se convierte en un número entero utilizando el método `Integer.parseInt( String s )` y se devuelve como resultado.

1.4. Estructura de Input

Tan pronto el usuario corra el programa la consola imprimirá estos tres mensajes secuencialmente:

```
Tamanio del buffer
5
```

Primero deberá ingresar el tamaño de los buffers, este número deber ser entero y mayor a 0. En este caso es 5.

```
Cuantos procesos por etapa?
5
```

Luego deberá ingresar el número de procesos por etapa, de los cuales se sabe que uno es naranja y los restantes son azules. Este número deber ser entero y mayor a 0. En este caso un proceso sería naranja y cuatro serían azules.

```
Cantidad de mensajes producidos
5
```

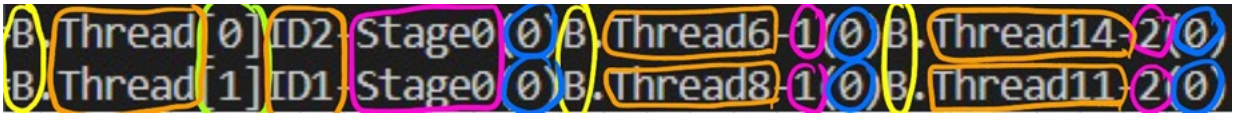
Por último, debe seleccionar la cantidad de mensajes que desea que se produzcan por cada tipo de proceso, este número deber ser entero y mayor a 0. En este caso se producirían 5 mensajes naranjas y 5 azules.
Si se desea, también se podría modificar el número de etapas que tiene el programa, sin embargo, por default el número de etapas se estableció como 3.

1.5. Estructura de Prints

Al ingresar correctamente los parámetros, el programa retornará una estructura de mensajes similar a esta, en donde la “R” impresa al inicio significa que por medio del proceso rojo se imprimieron los mensajes.

```
R
╔═══════════════════════════════════════════════════════════════════════════════?
╠B.Thread[0]ID2-Stage0(0)B.Thread6-1(0)B.Thread14-2(0)
╠B.Thread[1]ID1-Stage0(0)B.Thread8-1(0)B.Thread11-2(0)
╠B.Thread[2]ID2-Stage0(1)B.Thread7-1(0)B.Thread13-2(0)
╠B.Thread[3]ID1-Stage0(1)B.Thread9-1(0)B.Thread12-2(0)
╠B.Thread[4]ID2-Stage0(2)B.Thread6-1(1)B.Thread14-2(1)
╠B.Thread[5]ID4-Stage0(0)B.Thread9-1(1)B.Thread12-2(1)
╠B.Thread[6]ID4-Stage0(1)B.Thread6-1(2)B.Thread14-2(2)
╠B.Thread[7]ID2-Stage0(3)B.Thread8-1(1)B.Thread14-2(4)
╠B.Thread[8]ID1-Stage0(2)B.Thread7-1(1)B.Thread12-2(2)
╠B.Thread[9]ID2-Stage0(4)B.Thread6-1(3)B.Thread14-2(3)
╠O.Thread[10]ID0-Stage0(0)O.Thread5-1(0)O.Thread10-2(0)
╠O.Thread[11]ID0-Stage0(1)O.Thread5-1(1)O.Thread10-2(1)
╠O.Thread[12]ID0-Stage0(2)O.Thread5-1(2)O.Thread10-2(2)
╠O.Thread[13]ID0-Stage0(3)O.Thread5-1(3)O.Thread10-2(3)
╠O.Thread[14]ID0-Stage0(4)O.Thread5-1(4)O.Thread10-2(4)
╠B.Thread[15]ID1-Stage0(3)B.Thread9-1(2)B.Thread11-2(1)
╠B.Thread[16]ID4-Stage0(2)B.Thread6-1(4)B.Thread13-2(1)
╠B.Thread[17]ID1-Stage0(4)B.Thread8-1(2)B.Thread12-2(3)
╠B.Thread[18]ID4-Stage0(3)B.Thread7-1(2)B.Thread11-2(2)
╠B.Thread[19]ID4-Stage0(4)B.Thread9-1(3)B.Thread13-2(2)
╠B.Thread[20]ID3-Stage0(0)B.Thread9-1(4)B.Thread13-2(3)
╠B.Thread[21]ID3-Stage0(1)B.Thread8-1(3)B.Thread12-2(4)
╠B.Thread[22]ID3-Stage0(2)B.Thread7-1(3)B.Thread13-2(4)
╠B.Thread[23]ID3-Stage0(3)B.Thread8-1(4)B.Thread11-2(3)
╠B.Thread[24]ID3-Stage0(4)B.Thread7-1(4)B.Thread11-2(4)
╚═══════════════════════════════════════════════════════════════════════════════?
```

Ahora entendiendo las partes de cada mensaje:



Las letras encerradas en círculos amarillos indican el color del proceso por el que pasó cada mensaje, si el proceso era azul, la letra representada en el mensaje será “B”.
Si el proceso era naranja la letra representada pasa a ser la “O”.

Las palabras señaladas en naranja con la estructura “Thread#” siendo # un número, representando el id del proceso porque el que pasó el mensaje.

Los números entre llaves encerrados en círculos verdes representan el id de cada mensaje.
Como se puede ver, los mensajes se imprimen conforme al orden otorgado por estos ids.

Los números encerrados en rosado representan en qué etapa del proceso se realizó cada modificación.

Los números entre paréntesis señalados en azul representan el número de mensajes que había procesado cada proceso hasta ese momento.

Teniendo en cuenta esto, se puede ver que el mensaje generado cuando se crea el proceso es más específico que los que se generan en las etapas posteriores, sin embargo, tienen una estructura que sigue otorgando gran información.

2. Pruebas del programa

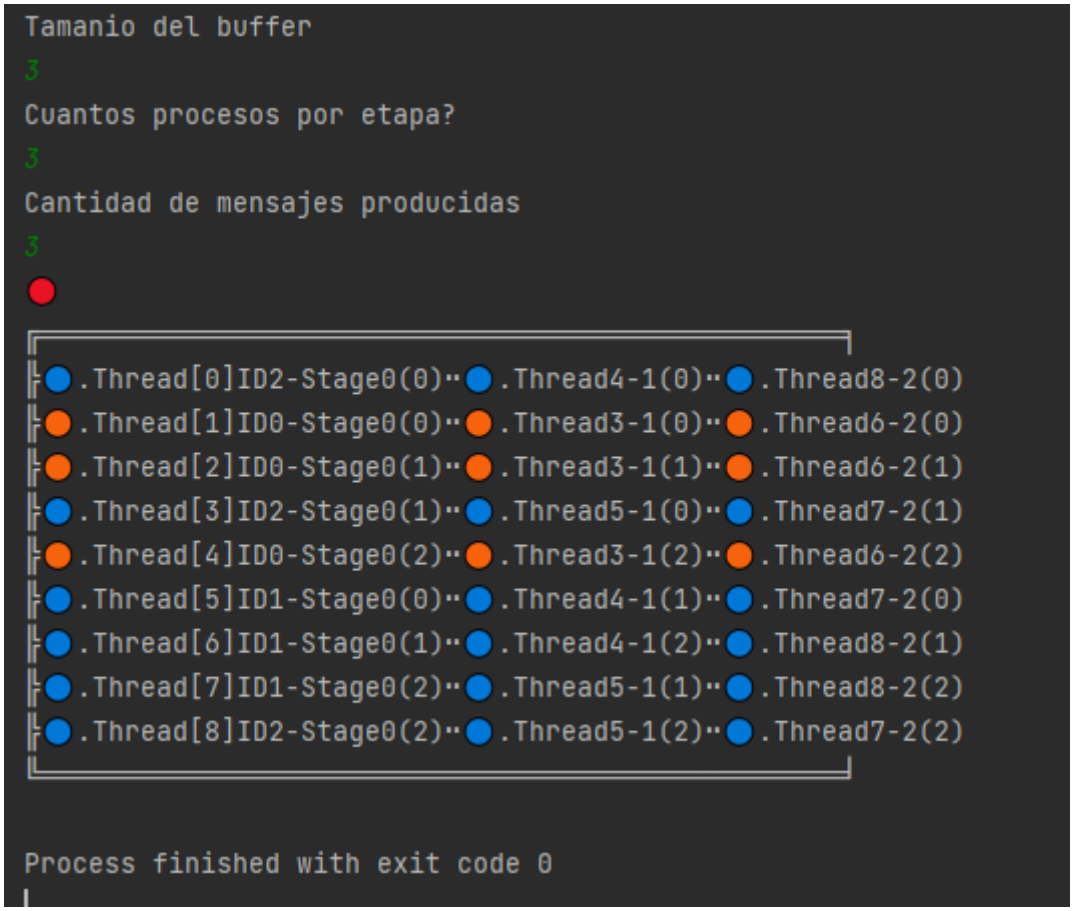
2.1. Estructura y requerimientos de prueba

Este programa fue probado en el IDE integrado IntelliJ IDEA 2022.3.2, con el JDK Java 11 y en la consola integrada del IDE. Fue probado en Windows 10 2H22 y Manjaro Linux 22.10, la estructura del proyecto está planteada con el sistema de Build dbl Java puro y es independiente librerías externas. Sin embargo, para conseguir los prints con símbolos integrados, la fuente de la consola en la cuál se compile debe ser compatible con simbolos Unicode, de lo contrario, solo será mostrado el código Unicode referente a los símbolos. Finalmente, solo es una decisión estética.

2.2. Pruebas

2.2.1. 10-10-10

2.2.2. 3-3-3


Tamanio del buffer
3
Cuantos procesos por etapa?
3
Cantidad de mensajes producidas
3
.Thread[0]ID2-Stage0(0)
.Thread4-1(0)
.Thread8-2(0)
.Thread[1]ID0-Stage0(0)
.Thread3-1(0)
.Thread6-2(0)
.Thread[2]ID0-Stage0(1)
.Thread3-1(1)
.Thread6-2(1)
.Thread[3]ID2-Stage0(1)
.Thread5-1(0)
.Thread7-2(1)
.Thread[4]ID0-Stage0(2)
.Thread3-1(2)
.Thread6-2(2)
.Thread[5]ID1-Stage0(0)
.Thread4-1(1)
.Thread7-2(0)
.Thread[6]ID1-Stage0(1)
.Thread4-1(2)
.Thread8-2(1)
.Thread[7]ID1-Stage0(2)
.Thread5-1(1)
.Thread8-2(2)
.Thread[8]ID2-Stage0(2)
.Thread5-1(2)
.Thread7-2(2)
Process finished with exit code 0

2.2.3. 5-5-5

3. Glosario

Función Lambda Es una función anónima, es decir, no depende de alguna clase. Similar a los métodos toman parametros y retornan algún valor, pero a diferencia de los primeros, estas no necesitan un nombre.

Objeto anónimo Es un objeto que no pertenece realmente a alguna clase, similar a las estructuras en C. Como dicho objeto no hace referencia a alguna clase ya declarada, no referencia a algún archivo .java y es principalmente usado para interactuar con bloques de expresiones lambda y si realmente no se considera valioso crear una clase para esta instancia.