Universidad Nacional de Córdoba

Facultad de Ciencias Exactas Físicas y Naturales

Trabajo Práctico Nº1

Programación Concurrente

Grupo “Los Simuladores”

Integrantes:

Bernaus, Julieta	42077611
Bottini, Franco Nicolas	41525391
Cabrera, Augusto Gabriel	42259653
Lencina, Aquiles Benjamín	43496784
Robledo, Valentín	42555866

Fecha

22/04/2022

Enunciado

En un sistema de adquisición de datos existen dos buffers y tres categorías de actores. Los buffers se denominan Buffer Inicial y Buffer de validados y tienen cada uno una capacidad de 100 datos (trabajan a pura pérdida, es decir si están llenos al venir un creador se descarta el dato). Los actores del sistema pueden ser Creadores de Datos, Revisores de Datos o Consumidores de Datos. El ciclo de funcionamiento normal del sistema comienza con la creación de un dato por parte de un "Creador de Datos". Este proceso lleva un tiempo aleatorio en ms (no nulo, a elección del grupo); una vez creado es almacenado en el Buffer Inicial. En este buffer debe permanecer hasta que el total de "Revisores" hayan revisado el mismo, la revisión del dato lleva un tiempo en ms (no nulo a elección del grupo). Una vez que todos los "Revisores" hayan revisado el dato, el último "Revisor" guardará una copia del mismo en el Buffer de Validados. Los "Revisores" no pueden revisar más de una vez cada dato. Los consumidores de datos son los encargados de eliminar los datos de ambos buffers, siempre y cuando ya hayan sido validados; la eliminación de un dato lleva un tiempo en ms (no nulo a elección del grupo).

Consigna

1. Hacer un diagrama de clases que modele el sistema de datos con TODOS los actores y partes.
2. Hacer un solo diagrama de secuencias que muestre la siguiente interacción:
    a. Con el buffer inicial vacío, un "Creador" genera un dato y lo deposita.
    b. Luego, un "Creador" y un "Revisor" llegan al mismo tiempo al buffer. Mostrar qué pasa con cada uno.
3. Modelar e implementar un sistema con los dos buffers con la capacidad mencionada y 4 Creadores de datos, 2 revisores de datos y 2 consumidores de datos.
4. Las demoras del sistema (creación, revisión, consumo) deben configurarse de tal manera de poder procesar 1000 datos (desde creación hasta consumo) en un periodo mínimo de 5 segundos y máximo de 10 segundos.
5. El grupo debe poder explicar los motivos de los resultados obtenidos. Y los tiempos del sistema.
6. Debe hacer una clase Main que al correrla, inicie los hilos.

Introducción

En primera instancia, realizamos el diagrama de clases del programa. Luego de esto, basándonos en el diagrama realizado procedimos a codificar la estructura básica del programa.

Una vez estructurado el software, procedimos a realizar el diagrama de secuencias que describe el proceso completo de creación, revisión y consumo de un dato. Esto nos permitió implementar la lógica de funcionamiento de los distintos actores que intervienen en el programa e identificar las zonas de acceso concurrente a memoria compartida, en donde debíamos aplicar secciones críticas. Para aplicar dichas secciones críticas, decidimos utilizar como herramienta de sincronización la interfaz **ReadWriteLock**, ya que la consideramos como la mejor alternativa para este caso.

Con la funcionalidad principal del programa ya implementada, procedimos a desarrollar el log que nos permitió debuggear nuestro código y recopilar información sobre la ejecución del programa. Finalmente, analizamos los resultados obtenidos y desarrollamos conclusiones al respecto.

Estructura del código

I. **Data:** clase que representa la estructura de datos que vamos a utilizar en nuestro programa. El valor del dato se representa mediante dos atributos *id* de tipo *int* y *value* de tipo *Date*.

   A. **Data():** constructor de la clase. Crea un nuevo dato asignando un *id* y estableciendo el valor del atributo *value* con la fecha y hora de la PC.

   B. **boolean isVerified(Reviewer reviewer):** método que recibe como parámetro un *reviewer* y verifica si este ya realizó la revisión del dato, en caso de ser así, retorna *true*, caso contrario, retorna *false*.

   C. **void verify(Reviewer reviewer):** método que registra la revisión del dato por parte *reviewer* que recibe como parámetro.

   D. **int getNumberOfReviews()**: método que retorna el número de revisiones que se realizaron sobre el dato.

   E. **boolean isCopiedToValidBuffer():** método que se encarga de verificar si el dato fue copiado a una instancia de la clase **ValidBuffer**, en caso de ser así, retorna *true*, caso contrario, retorna *false*.

   F. **boolean tryCopiedToValidBuffer():** método que verifica si el dato fue copiado a una instancia de la clase **ValidBuffer**, en caso de ser así, retorna *false*. En caso contrario, establece el estado del dato como “copiado” y retorna *true*.

   G. **Date getValue():** método *getter* del atributo *value*.

   H. **void setId(int id):** método *setter* del atributo *id*.

   I. **int getId():** método *getter* del atributo *id*.

II. **Buffer:** clase abstracta que representa un espacio de almacenamiento de datos.

   A. **Buffer(int maxSize, int targetAmountOfData):** constructor de la clase. Recibe como parámetro la cantidad máxima de datos almacenables en el *buffer* y la cantidad objetivo de datos que se van a querer almacenar dentro del mismo.

   B. **int getSize():** método que retorna el número de datos almacenados en el *buffer*.

III. **InitBuffer:** clase que extiende de la clase **Buffer**. Implementa el almacén donde los datos se guardan luego de ser creados.

   A. **InitBuffer(int maxSize, int targetAmountOfData)**: constructor de la clase. Recibe como parámetro la cantidad máxima de datos almacenables en el

buffer y la cantidad objetivo de datos que se van a querer almacenar dentro del mismo.

B. **boolean add(Data value) throws InterruptedException:** método que se encarga de añadir al *buffer* el dato recibido como parámetro y retornando *true* como resultado. Si el *buffer* está lleno, no se añade el dato y el método retorna *false*.

C. **void remove(Data value) throws InterruptedException:** método que se encarga de remover del *buffer* el dato que recibe como parámetro.

D. **Data getItem(Data last) throws InterruptedException:** método que retorna el dato del *buffer* que se encuentra a continuación del dato que recibe como parámetro (*last*). En caso de no existir ningún dato a continuación de *last*, retorno un dato aleatorio del *buffer*.

E. **boolean isNotCreationCompleted()**: método que retorna *true* en caso de que, la cantidad de datos que almacenó el buffer a lo largo de su vida sea distinta al objetivo dado (*targetAmountOfData*), en caso contrario, retorna *false*. En otras palabras verifica si el proceso de creación de datos ha terminado ya que se alcanzó el objetivo propuesto.

IV. **ValidBuffer:** clase que extiende de la clase **Buffer**. Implementa el almacén donde los datos que ya han sido validados son guardados.

A. **ValidBuffer(int maxSize, int targetAmountOfData):** constructor de la clase. Recibe como parámetro la cantidad máxima de datos almacenables en el *buffer* y la cantidad objetivo de datos que se van a querer almacenar dentro del mismo.

B. **void add(Data value) throws InterruptedException:** método que se encarga de añadir al *buffer* el dato que recibe como parámetro. Si el buffer se encuentra lleno, la ejecución del hilo queda suspendida a la espera de un lugar en el *buffer*.

C. **Data poll() throws InterruptedException**: método que se encarga de remover y retornar el primer elemento del buffer. Si el *buffer* está vacío y el proceso de revisión no ha finalizado, la ejecución del hilo queda suspendida a la espera de que ingrese un dato para poder retornar. En caso de que haya terminado el proceso de revisión y el buffer se encuentre vacío, el método retorna *null*.

D. **boolean isNotReviewCompleted()**: método que retorna *true* en caso de que, la cantidad de datos que almacenó el *buffer* a lo largo de su vida sea distinta al objetivo dado (*targetAmountOfData*), en caso contrario, retorna *false*. En

otras palabras, verifica si el proceso de revisión de datos ha terminado ya que se alcanzó el objetivo propuesto.

V. **Creator:** clase que implementa la interfaz **Runnable**. Se encarga de crear los datos y almacenarlos en una instancia de la clase **InitBuffer**. Este procedimiento de creación de datos se realiza a pura pérdida, es decir, en caso de estar lleno el *buffer*, el dato creado se descarta.

   A. **Creator(InitBuffer initBuffer, String name)**: constructor de la clase. Recibe como parámetros: la instancia de la clase **InitBuffer** (*initBuffer*) sobre la que va a operar y un nombre para identificar al creador (*name*).

   B. **@Override void run():** método sobrescrito de la interfaz **Runnable**. Implementa la lógica del proceso de creación de datos.

   C. **int getDataAccepted():** método que retorna la cantidad de datos añadidos exitosamente al *buffer*.

   D. **int getDataLost():** método que retorna la cantidad de datos descartados en el proceso de creación.

   E. **String getName():** método *getter* del atributo *name*.

VI. **Reviewer:** clase que implementa la interfaz **Runnable**. Se encarga de revisar los datos almacenados en una instancia de la clase **InitBuffer** y copiarlos a una instancia de la clase **ValidBuffer** cuando ya están validados.

   A. **Reviewer(InitBuffer initBuffer, ValidBuffer validBuffer, int totalReviewers, String name)**: constructor de la clase. Recibe como parámetros: las instancias de las clases **InitBuffer** (*initBuffer*) y **validBuffer** (*validBuffer*) sobre las que va a operar, el número de revisores totales que tendrá el programa (*totalReviewers*) y un nombre para identificar al revisor (*name*).

   B. **@Override void run():** método sobrescrito de la interfaz **Runnable**. Implementa la lógica del proceso de revisión de datos.

   C. **int getDataMoved():** método que retorna la cantidad de datos que han sido copiados al *validBuffer*.

   D. **int getDataVerified():** método que retorna la cantidad de revisiones que se han realizado.

   E. **String getName():** método *getter* del atributo *name*.

VII. **Consumer**: clase que implementa la interfaz **Runnable**. Se encarga de remover los datos almacenados en una instancia de la clase **ValidBuffer** y en una instancia de la clase **InitBuffer**. Operación que denominamos consumo.

A. **Consumer(InitBuffer initBuffer, ValidBuffer validBuffer, String name)**: constructor de la clase. Recibe como parámetros: las instancias de las clases **InitBuffer** (*initBuffer*) y **validBuffer** (*validBuffer*) sobre las que va a operar y un nombre para identificar al consumidor (*name*).

B. **@Override void run():** método sobrescrito de la interfaz **Runnable**. Implementa la lógica del proceso de consumo de datos.

C. **int getDataConsumed():** método que retorna la cantidad de datos que se han consumido.

D. **String getName():** método *getter* del atributo *name*.

VIII. **Log:** clase que extiende de la clase **Thread**. Se encarga de guardar estadísticas en un archivo de texto durante la ejecución del programa, para la realización de *testing*.

A. **Log(InitBuffer initBuffer, ValidBuffer validBuffer, Creator[] creators, Consumer[] consumers, Reviewer[] reviewers, Thread[] creatorsThreads, Thread[] consumersThreads, Thread[] reviewersThreads)**: constructor de la clase. Recibe como parámetros todas las instancias de los distintos objetos que van a intervenir en la ejecución del programa.

B. **static void clearFile():** método que se encarga de borrar todo el contenido del archivo de texto sobre el que estamos trabajando.

C. **@Override void run():** método sobrescrito de la clase **Thread**. Implementa la lógica de funcionamiento del log.

D. **void writeLog():** método encargado de escribir en el archivo de texto información acerca de la ejecución del programa.

La estructura del programa queda plasmada en el siguiente diagrama de clases:

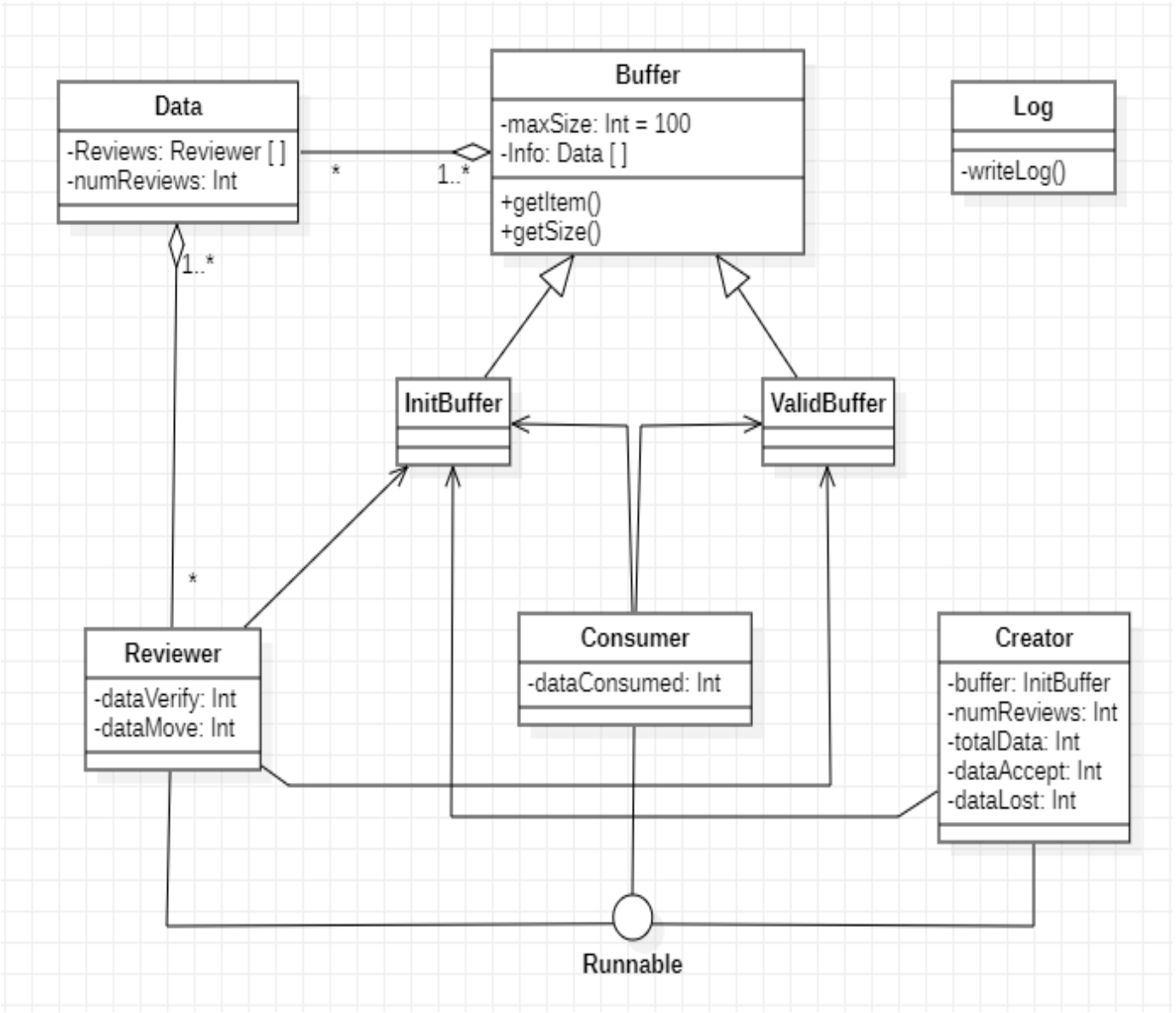

Figura 1: Diagrama de clases del programa.

Funcionamiento del código

Creamos las instancias de las clases y las variables que vamos a utilizar. A continuación, inicializamos los arreglos de los distintos actores: *creators*, *consumers* y *reviewers* y los arreglos de los hilos (*threads*): *creatorsThreads*, *consumersThreads* y *reviewersThreads* de la siguiente manera: 4 hilos de *creators*, 2 hilos de *consumers* y 2 hilos de r*eviewers.* Además, creamos un objeto de la clase **Log**. Seguido de esto, se ponen a funcionar todos los hilos y comienza la ejecución del programa:

- **Creator:** se encarga de crear instancias de la clase **Data** y guardarlas dentro de *initBuffer*. Esto se va a repetir de manera iterada mientras que la cantidad total de datos agregados al *initBuffer* sea menor a la establecida como objetivo cuando este fue creado (*targetAmountOfData*). El creador cuenta con un contador que contabiliza la cantidad total de datos que ha agregado al *initBuffer* y podemos consultar su valor a través del método *getDataAccepted()*. Si el *initBuffer* está lleno, las instancias creadas no son agregadas y se descartan. El creador también cuenta con un contador que contabiliza la cantidad de datos descartados y podemos acceder a su valor por medio del método *getDataLost()*. Finalmente, el hilo duerme por 15 milisegundos entre iteración e iteración.

- **Reviewer:** se encarga de chequear por única vez cada instancia de la clase **Data** que está almacenada en el *initBuffer.* Esto se va a repetir de manera reiterada mientras que el proceso de creación de datos no haya finalizado y el initBuffer contenga elementos. El revisor contabiliza la cantidad de datos que ha verificado a través de un contador y podemos consultar su valor por medio del método *getDataVerified()*. En el momento que todos los revisores han revisado el dato, este es copiado en el *validBuffer* por el último revisor en realizar la revisión alterando el estado del dato y estableciendolo como “copiado”. La cantidad de datos que el revisor ha copiado al *validBuffer* es contabilizada por un contador y tenemos acceso a su valor por medio del método *getDataMoved()*. Finalmente, el hilo duerme por 2 milisegundos entre iteración e iteración.

- **Consumer**: se encarga de ir tomando y removiendo (*poll*) los datos almacenados en el *validBuffer* y removiendo la copia del dato almacenada en el *InitBuffer*, a este proceso lo llamamos consumo. Esto se va a repetir de manera iterada mientras que el proceso de revisión de datos no haya finalizado y el *validBuffer* contenga elementos. El consumidor contabiliza la cantidad de datos consumidos a través de un contador, al cual podemos acceder mediante el método *getDataConsumed()*. Finalmente, el hilo duerme por 10 milisegundos entre iteración e iteración.

- **Log:** se encarga de imprimir en un archivo de texto llamado “log.txt” la información acerca del estado del programa en intervalos de 2 segundos. Esto se va a repetir

hasta que finalice la ejecución de todos los hilos: *creators, consumers* y *reviewers*. Los valores que se imprimen en el archivo son los siguientes:

- *Generales:*
  - Tiempo de ejecución del programa **(“*Execution time*”)**.
  - Cantidad de datos almacenados en *InitBuffer* **(“*InitBuffer size*”)**.
  - Cantidad de datos almacenados en *ValidBuffer* **(“*validBuffer size*")**.
- *Creators:*
  - Total de datos descartados por **(“*Creators Totals: <data lost>*”)**.
  - Total de datos agregados al *initBuffer* **(“*Creators Totals: <data accepted>*”)**.
  - Total de datos producidos **(“*Creators Totals: <total data produced>*”)**.
  - Porcentaje total de pérdida de datos **(“*Creators Totals: <loss percentage>*”)**.
  - Datos descartados por el *creator n* **(“*Creator n: <data lost>*”)**.
  - Datos totales agregados al *initBuffer* por el *creator n* **(“*Creator n: <data accepted>*”)**.
  - Datos totales producidos por el *creator n* **(“*Creator n: <total data produced>*”)**.
  - Porcentaje de pérdida de datos del *creator n* **(“*Creator n: <loss percentage>*”)**.
  - Porcentaje de responsabilidad en la pérdida total de datos del *creator n* **(“*Creator n:* <responsibility *percentage in lost data>*”)**.
  - Porcentaje de responsabilidad en la agregación total de datos del *creator n* **(“*Creator n:* <responsibility *percentage in accepted data>*”)**.
  - Porcentaje de responsabilidad en la creación total de datos del *creator n* **(“*Creator n:* <responsibility *percentage in produced data>*”)**.
- *Reviewers:*
  - Total de datos revisados ***(“Reviewers Totals: <data revised>”).***
  - Total de datos copiados al *validBuffer* ***(“Reviewers Totals: <data copied>”).***

- Datos revisados por el *reviewer n* **(“Reviewer n: <data revised>”)**.
- Datos copiados al *validBuffer* por el *reviewer n* **(“Reviewer n: <data copied>”)**.
- Porcentaje de responsabilidad en la copia de los datos al *validBuffer* del *reviewer n* **(“Reviewer n: <responsibility percentage in copied data>”).**

◆ *Consumers:*

- Total de datos consumidos **(“Consumers Totals: <data taken>”)**.
- Datos consumidos por el *consumer n* **(“Conumer n: <data taken>”)**.
- Porcentaje de responsabilidad en el consumo de datos del *consumer n* **(“Consumer n: <responsibility percentage in taken data>”)**.

Finalmente, el programa finaliza cuando todos los hilos hayan terminado sus tareas. En caso de que el valor de la constante *NUMBER_OF_EXECUTIONS* de la clase **Main** sea mayor a uno, el programa se reinicia y repite tantas veces como indique esta constante.

El accionar del proceso de creación, revisión y consumo de un dato se puede observar en el siguiente diagrama de secuencia:

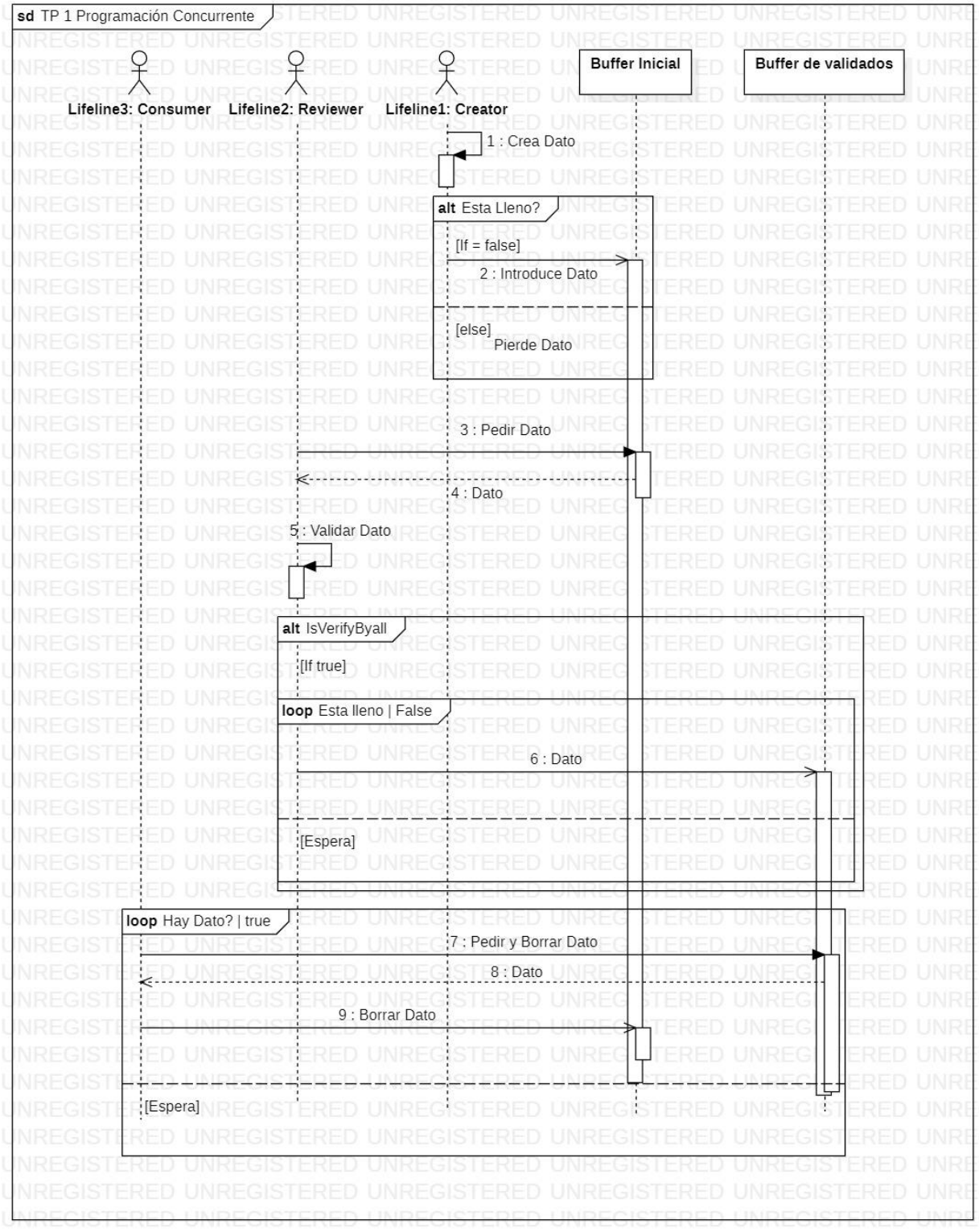

Figura 2: Diagrama de Secuencia del proceso creación, revisión y consumo de un dato.

Resultados típicos

En la figura 3 se observa la salida que por consola durante la ejecución del programa, donde se puede ver el accionar de los diferentes actores del sistema, *reviewers* agregando datos al *ValidBuffer*, *creators* agregando al *InitBuffer* y *consumers* removiendo datos de ambos *buffers*.

Por otra parte, en la figura 4 se muestra un fragmento de los resultados que se imprimen en el archivo *log.txt*, donde se puede ver información la información que se detallo anteriormente.

```
[InitBuffer (Size: 97)] Consumer 0 (Thread ID: 20) data removed <ID: 1031 - Value: Sun Apr 24 11:50:49 ART 2022>
[InitBuffer (Size: 98)] Creator 0 (Thread ID: 16) data added <ID: 1178 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer (Size: 99)] Creator 1 (Thread ID: 17) data added <ID: 1179 - Value: Sun Apr 24 11:50:50 ART 2022>
[ValidBuffer (Size: 51)] Consumer 1 (Thread ID: 21) data removed <ID: 1032 - Value: Sun Apr 24 11:50:49 ART 2022>
[InitBuffer (Size: 98)] Consumer 1 (Thread ID: 21) data removed <ID: 1032 - Value: Sun Apr 24 11:50:49 ART 2022>
[InitBuffer] Reviewer 1 (Thread ID: 23): verified data <ID: 1111 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer (Size: 99)] Creator 2 (Thread ID: 18) data added <ID: 1180 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer (Size: 100)] Creator 3 (Thread ID: 19) data added <ID: 1181 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer] Reviewer 0 (Thread ID: 22): verified data <ID: 1154 - Value: Sun Apr 24 11:50:50 ART 2022>
[ValidBuffer (Size: 52)] Reviewer 1 (Thread ID: 23) data added <ID: 1111 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer] Reviewer 0 (Thread ID: 22): verified data <ID: 1155 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer] Reviewer 1 (Thread ID: 23): verified data <ID: 1113 - Value: Sun Apr 24 11:50:50 ART 2022>
[ValidBuffer (Size: 51)] Consumer 0 (Thread ID: 20) data removed <ID: 1033 - Value: Sun Apr 24 11:50:49 ART 2022>
[InitBuffer (Size: 99)] Consumer 0 (Thread ID: 20) data removed <ID: 1033 - Value: Sun Apr 24 11:50:49 ART 2022>
[ValidBuffer (Size: 52)] Reviewer 1 (Thread ID: 23) data added <ID: 1113 - Value: Sun Apr 24 11:50:50 ART 2022>
[InitBuffer] Reviewer 0 (Thread ID: 22): verified data <ID: 1156 - Value: Sun Apr 24 11:50:50 ART 2022>
```

Figura 3: captura de lo presentado en consola durante ejecución promedio.

```
*--------------------------------------------------------------------------------*
Execution time: 4,182000 [Seg]
InitBuffer size: 100
ValidBuffer size: 74
*--------------------------------------------------------------------------------*

   Creators Totals:
       data lost: 182
       data accepted: 776
       total data produced: 958
       loss percentage: 18,997911 %

   Creator 0:
      data lost: 62
      data accepted: 180
      total data produced: 242
      loss percentage: 25,619835 %
      responsibility percentage in lost data: 34,065933 %
      responsibility percentage in accepted data: 23,195877 %
      responsibility percentage in produced data: 25,260960 %

   Creator 1:
      data lost: 44
      data accepted: 192
      total data produced: 236
      loss percentage: 18,644068 %
      responsibility percentage in lost data: 24,175825 %
```

Figura 4: captura de un fragmento del archivo log.txt generado por el programa.

Conclusión

Al finalizar con la realización de este trabajo práctico, podemos extraer las siguientes conclusiones:

- La conclusión principal que nos llevamos es la importancia de manejar bien la sincronización de hilos, ya que, si no son bien administradas las zonas donde existe acceso concurre a la memoria compartida (secciones críticas), se puede comprometer la integridad de los datos y producir resultados erróneos .

- En un comienzo, se utilizó los hilos *daemon* trabajando como *reviewers*, lo cual en ciertas ocasiones provocaba problemas de concurrencia, ya que el sistema operativo no maneja de la misma manera este tipo de hilos y los hilos convencionales. Por esto, concluimos en dejar a los *reviewers* también como hilos convencionales.

- Mientras determinamos cuál sería la mejor herramienta para el manejo de la sincronización de los hilos, pudimos comprobar por nosotros mismos las diferencias que existen entre unas y otras.

- Durante el proceso de desarrollo pudimos observar que es complejo manejar todas las posibles variables que intervienen en un proceso concurrente, y por tanto, a medida que crece el programa se hace más difícil no cometer errores.