Documentación caso 1

sa.ospino - 201913643

Este breve documento pretende explicar cada clase, sus métodos y las relaciones que existen entre estas. Para el UML, si algo está subrayado es porque es una constante, y si está en negrilla, es porque es estático.

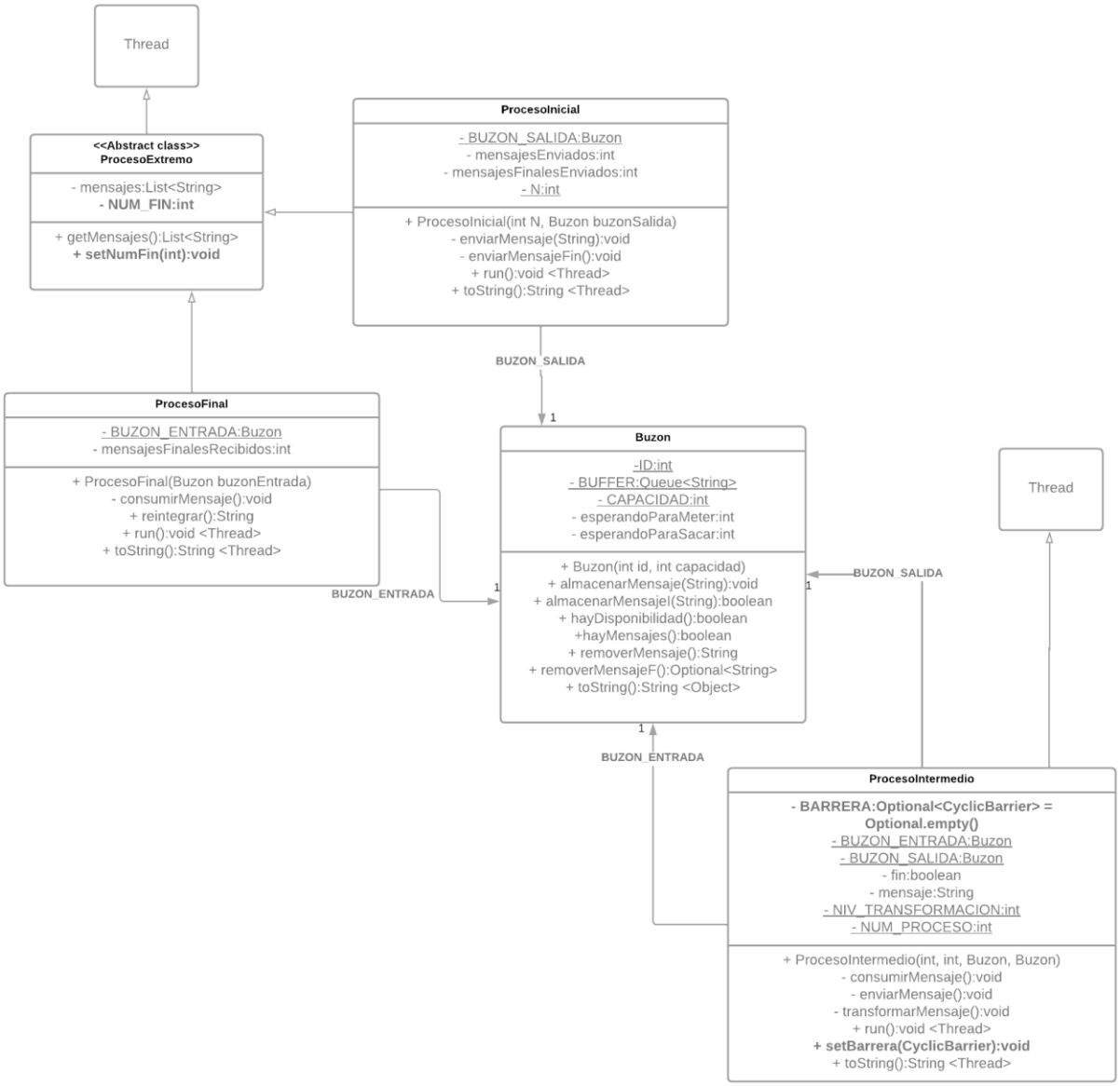

Diagrama UML del caso

Clase Utiles

Esta clase no aparece en el UML porque no hace parte del modelo del problema. Contiene los métodos necesarios para verificar que los mensajes salientes del proceso inicial sí llegaron al proceso final, estos métodos son:

1. **codigoHash**: Es un método estático privado que recibe una lista por parámetro y retorna un entero que representa el código Hash o identificador de dicha lista. El criterio es, si dos listas tienen los mismos elementos y en igual cantidad, entonces ambas listas tendrán el mismo código Hash.
2. **checkSum**: Es un método estático público que recibe dos listas por parámetro y retorna un booleano. Si las dos listas tienen los mimos elementos y en igual cantidad, entonces retorna true. Se usa el método códigoHash antes para esto.

Clase ProcesoExtremo

Esta clase extiende de Thread, es abstracta y representa un proceso que se encuentra al comienzo o al final. Tiene mensajes y un método que los retorna. Además, tiene un atributo estático y constante llamado NUM_FIN, que representa el número de mensajes finales que van a existir en todo el flujo del programa, es decir, los mensajes finales que va a generar el proceso inicial.

Clase ProcesoInicial

Esta clase extiende de ProcesoExtremo. Representa el proceso inicial del programa, es decir, de aquí salen todos los mensajes.

a. **Atributos**
    1. **BUZON_SALIDA:** Este es el primer buzón del flujo del programa al que llegan mensajes, todos son enviados por el proceso inicial, además, existen varios procesos intermedios que consumen mensajes de este buzón. Es constante, por lo que su pointer no cambia a lo largo de todo el programa.
    2. **mensajesEnviados:** Este número representa el número de mensajes normales que el proceso inicial ha enviado.
    3. **mensajesFinalesEnviados:** Este número tiene una utilidad similar al anterior, representa el número de mensajes finales enviados. Sirve para saber cuándo dejar de enviar mensajes finales. Este valor debe llegar a N, cuando esto sucede, se deben dejar de enviar mensajes finales, e idealmente, el Thread debe parar su ejecución.
    4. **N:** Es el número de mensajes que se tienen que producir.
b. **Métodos**
    1. **ProcesoInicial:** Es el constructor, recibe por parámetro N y el buzón de salida. Inicializa una lista vacía de mensajes y define en 0 los mensajes normales y finales enviados.
    2. **enviarMensaje:** Este método es usado dentro de la misma clase y encapsula la lógica para enviar un mensaje al buzón de salida. Recibe por parámetro el mensaje que se requiere enviar. Se conforma de un loop while que no acaba hasta que se confirme que el mensaje se ha enviado correctamente. Además, dentro de este loop, existe otro loop que se asegura de que el Thread no avance y ceda el procesador en caso de que no haya disponibilidad en el buzón de salida para almacenar un mensaje. En caso de éxito, se suma uno a los mensajes enviados y se agrega el mensaje a la lista de mensajes del

proceso, y finalmente se sale del loop más grande y por lo tanto del método. En caso de fracaso, las instrucciones del método se vuelven a intentar.

3. **enviarMensajeFin:** Este método es usado dentro de la misma clase y encapsula la lógica para enviar mensajes finales, es decir, de terminación. La lógica es exactamente a la anterior, a excepción de que el método no recibe ningún parámetro, el mensaje final es una constante.
4. **run:** Es la rutina que corre el Thread del proceso. La primera parte es un loop que envía mensajes hasta que mensajesEnviados no cumpla con cierto predicado. Este predicado es: el número de mensajes enviados no debe sobrepasar a N, es decir, los permitidos. La segunda parte consta de otro loop que envía mensajes finales hasta que mensajesFinalesEnviados no cumpla con cierto predicado. Este predicado es: el número de mensajes finales enviados no debe sobrepasar a NUM_FIN, es decir, los permitidos. Después, el Thread finaliza.
5. **toString:** Simplemente define una forma de imprimir en pantalla el proceso.

Clase ProcesoFinal

Esta clase extiende de ProcesoExtremo. Representa el proceso final del programa, es decir, aquí deben llegar todos los mensajes transformados para que puedan ser reintegrados.

c. Atributos

1. **BUZON_ENTRADA:** Este es el último buzón del flujo del programa al que llegan mensajes, todos son enviados por los procesos de la última transformación. Es constante, por lo que su pointer no cambia a lo largo de todo el programa.
2. **mensajesFinalesRecibidos:** Este número representa el número de mensajes finales, es decir, de terminación, que el proceso final ha consumido desde su buzón de entrada.

d. Métodos

1. **ProcesoFinal:** Es el constructor, recibe por parámetro el buzón de entrada. Inicializa una lista vacía de mensajes y define en 0 los mensajes finales recibidos.
2. **consumirMensaje:** Este mensaje es usado por la misma clase y encapsula la lógica detrás de consumir un mensaje del buzón de entrada. La primera parte consta de un while que retiene al Thread en caso de que no haya mensajes para consumir en su buzón de entrada, esto lo hace cediendo el procesador. Luego, intenta consumir un mensaje de su buzón de entrada. Si tuvo éxito, puede haber posibilidades: 1) mensaje normal; se almacena el mensaje totalmente transformado en la lista de mensajes del proceso. 2) mensaje de fin; se le suma uno a los mensajes finales recibidos. En caso de que no se haya podido consumir el mensaje, no se hace nada.
3. **reintegrar:** Este método simplemente consiste en concatenar todos los mensajes, en el orden que se hayan almacenado en la lista de mensajes del proceso.
4. **run:** Consiste en un loop que consume mensajes, o intenta consumir mensajes, mientras el número de mensajes finales recibidos no haya llegado a su máximo, puesto que esto indica su fin. Este valor es NUM_FIN.
5. **toString:** Simplemente define una forma de imprimir en pantalla el proceso.

Clase ProcesoIntermedio

Esta clase extiende de ProcesoExtremo. Representa el proceso final del programa, es decir, aquí deben llegar todos los mensajes transformados para que puedan ser reintegrados.

a. **Atributos**
    1. **BUZON_ENTRADA:** Es el buzón de donde el proceso intermedio consume mensajes, no cambia.
    2. **BUZON_SALIDA:** Es el buzón de donde el proceso intermedio envía mensajes, no cambia.
    3. **BARRERA:** Es una barrera que es global y constante para todos los procesos intermedios, sirve para definir un punto común que todos los procesos deben alcanzar para poder ejecutar una instrucción final, este punto común es la finalización del proceso intermedio y la instrucción final es imprimir en pantalla que todos los procesos intermedios han terminado su ejecución. Este objeto es opcional, lo que quiere decir que puede no existir, es una implementación de Java para hacer programas seguros de los nulls.
    4. **fin:** Es un booleano que sirve de bandera para saber si el proceso intermedio ya debe terminar.
    5. **mensaje:** Es el mensaje que el proceso contiene.
    6. **NIV_TRANSFORMACION:** Este número representa el nivel de transformación del proceso, es una constante.
    7. **NUM_PROCESO:** Este número representa el número del proceso, es una constante.

b. **Métodos**
    1. **ProcesoIntermedio:** Es el constructor, recibe por parámetro los buzones y el número de transformación y proceso. Establece inicialmente la bandera fin como falsa, puesto que el proceso apenas está iniciando.
    2. **consumirMensaje:** Este método extrae o consume un mensaje del buzón de entrada del proceso en cuestión, simplemente remplaza el mensaje por el resultado del llamado al método de removerMensaje que se encuentra el buzón de entrada.
    3. **enviarMensaje:** Este método envía el mensaje del proceso intermedio al buzón de la salida, simplemente llama al método almacenarMensaje definido en el buzón de salida y le manda por parámetro el mensaje.
    4. **transformarMensaje:** Este método transforma el mensaje que actualmente se encuentra almacenado en el proceso. Simplemente le añade al mensaje actual la cadena TNM donde N es el número de la transformación y M es el número del proceso.
    5. **run:** Este es el método que se ejecuta cuando se empieza el thread. Al principio hay un while que se encarga de consumir, transformar y enviar mensajes siempre y cuando el thread esté despierto y la bandera de fin sea falsa. Cabe resaltar que se verifica que el mensaje consumido sea 'FIN', puesto que, en caso de serlo, no se transforma el mensaje y se establece que fin ahora es verdad, posteriormente se envía le mensaje. Por último, se aguarda con la barrera en caso de que esta exista o esté definida.
    6. **setBarrera:** Recibe por parámetro la barrera que se quiere establecer y la define.
    7. **toString:** Simplemente define una forma de imprimir en pantalla el proceso.

Clase Buzon

Esta clase modela los buffers o colas de comunicación entre los distintos procesos. Cabe resaltar que solo el buzón inicial tiene múltiples procesos sacando mensajes de este y que solo el buzón final tiene múltiples procesos almacenándole mensajes. Los métodos especiales, que se describirán más abajo, fueron desarrollados de tal manera de que el yield no fuera ejecutado en el método sincronizado del buzón, sino que fuera ejecutado desde los procesos extremos, ya que usar yield en un método sincronizado cede el procesador, pero no hace que el thread ceda el monitor.

a. Atributos

1. **ID:** Un identificador único del buzón, es útil visualmente para saber si los buzones se asignaron de manera correcta a los distintos procesos.
2. **BUFFER:** Es una cola que almacena los mensajes que actualmente están en el buzón. Implementa la funcionalidad de añadir y sacar elementos de la cola, es decir, lo primero que entra es lo primero que sale. Esto garantiza que los mensajes de finalización sean sacados únicamente cuando se hayan sacado todos los normales, de esta manera no existe la posibilidad de que se pierdan mensajes (recordar que los mensajes finales son los últimos que manda el proceso inicial).
3. **esperandoParaSacar:** Este número representa el número de procesos que están esperando o que están dormidos en el monitor para sacar un mensaje del buffer del buzón. La razón para dormir un proceso, en este caso, se debe a que el buffer está vacío.
4. **esperandoParaMeter:** Este número representa el número de procesos que están esperando o que están dormidos en el monitor para meter un mensaje del buffer del buzón. La razón para dormir un proceso, en este caso, se debe a que el buffer llegó a su máxima capacidad.

b. Métodos

1. **Buzon:** Es el constructor del buzón. Recibe por parámetro el id del buzón y la capacidad de este. Inicia un buffer vacío y establece que los números de espera son 0.
2. **almacenarMensaje:** Este método es usado por los procesos intermedios para poder consumir un mensaje, recibe por parámetro el mensaje que se desea guardar y está sincronizado, por lo que se encuentra en la zona crítica del monitor del buzón. En un principio, existe un while que retiene el flujo del thread en caso de que no haya espacio para almacenar mensajes, notar que cuando esto sucede, se aumenta en uno el número de mensajes que están esperando para meter y se duerme el thread en el monitor, si se llega a despertar el thread a través de un notify y este retoma el monitor, se disminuye en 1 el número espera para meter y sigue de nuevo con el while. Posteriormente, cuando haya espacio para almacenar mensajes, se añade el mensaje al buffer del buzón. En caso de que el número de espera para sacar sea mayor a 0, se notifican todos para que estos puedan competir para sacar el mensaje recién metido, sin embargo, notar que también se notifican los que se durmieron para esperar a meter, por lo que puede suceder lo contrario a lo deseado; que se almacene otro mensaje nuevo en vez de sacar el mensaje recién ingresado. Esto no supone un problema, puesto que para buzones intermedios siempre hay un proceso despierto y otro dormido en el monitor, de hecho, solo son estos dos.

3. **almacenarMensajeI:** Es un método especial que intenta almacenar mensajes que es usado exclusivamente por el proceso inicial y que es sincronizado, por lo que se encuentra dentro de la sección crítica del monitor del buzón, recibe por parámetro el mensaje que desea guardar en el buffer y retorna true si se guardó y false de lo contrario. Si hay procesos esperando para sacar mensajes del buzón inicial, entonces se les notifica para que puedan despertar y sacar un mensaje.
4. **removerMensaje:** Este es un método sincronizado que encapsula la lógica para remover un mensaje del buffer del buzón, retorna este mismo mensaje. Primeramente, contiene un loop que se encarga de retener el thread en caso de que no haya mensajes para sacar, se suma uno al número de esperando para sacar y se duerme el thread en el monitor. Una vez se notifica, se resta uno a los que están esperando para sacar y se continua con el while. En caso de haber disponibilidad, se notifican a los procesos dormidos y finalmente se retira y retorna el primer mensaje en la cola del buffer.
5. **removerMensajeF:** Este es un método especial sincronizado que es usado exclusivamente por el proceso final para que este pueda remover mensajes. En realidad, este método intenta remover un mensaje y retorna un opcional, si se verifica que hay mensajes, entonces el opcional almacena el mensaje retirado, sino entonces se retorna un opcional vacío. Optional es una implementación de java para protegerse de los nulls.
6. **hayDisponibilidad:** Es un booleano que informa de si hay disponibilidad para guardar mensajes en el buffer.
7. **hayMensajes:** Es un booleano que informa de si hay mensajes en el buffer.
8. **toString:** Simplemente define una forma de imprimir en pantalla el proceso.

Clase Main

Esta clase inicia y corre la estructura de Map reduce: 1) En la primera parte se pregunta, por consola, el valor de las variables necesarias para iniciar el programa y se definen los procesos extremos. Además, se establecen valores globales como la barrera de los procesos intermedios y el número de mensajes finales que el proceso inicial tiene que enviar. 2) En la segunda etapa se crean los buzones y las estructuras de datos que los configuran, específico, una matriz. 3) En la tercera etapa se crean los procesos, se les asignan buzones y se organizan en una estructura de datos que los organizan, en específico, un array. 4) Se imprimen las estructuras de datos anteriormente definidas, para confirmar que sí tienen la forma del Map Reduce. 5) Se inician los procesos desde el FINAL, es decir, primero se inicia el thread del proceso final, luego los de la última transformación, y así hasta llegar al proceso inicial, notar que este empieza a mandar mensajes cuando ya todos los demás procesos han empezado a correr (de hecho, todos los intermedios estarían dormidos). 6) Finalmente se espera a que termine el proceso final, luego se hace la reintegración de los mensajes transformados y se imprime en pantalla, y se verifica que hayan llegado todos los mensajes a través de la función checksum.