PRÁCTICA 3: PROGRAMACIÓN DEL FUNCIONAMIENTO DE UNA FÁBRICA (MULTITHREAD)

SISTEMAS OPERATIVOS

Curso 2014-2015

Grado en Ingeniería Informática

Universidad Carlos III de Madrid

JAVIER PRIETO CEPEDA: 100 307 011

RUBÉN RODRÍGUEZ MAXIMIANO: 100 303 579

ÍNDICE

TABLAS DE PRUEBAS

1 Introducción

En éste documento se procede a la explicación de la práctica 3, *"Programación del funcionamiento de una fábrica (Multithread)"*. La realización de la misma la han llevado a cabo los alumnos Javier Prieto Cepeda y Rubén Rodríguez Maximiano.

En primer lugar, se describirá el código, indicando las decisiones de diseño pertinentes para la correcta implementación de la práctica.

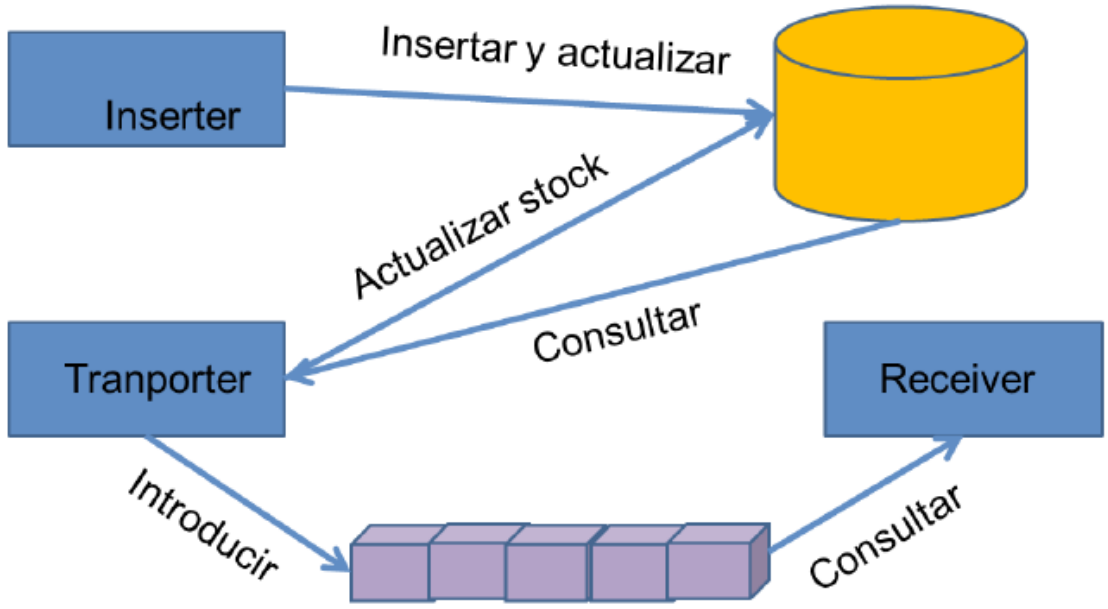
Una vez descrito el código, se mostrará una batería de pruebas en las cuales se puede comprobar su correcto funcionamiento.

Por último, se realizarán una serie de valoraciones y conclusiones sobre la realización de la práctica.

2 Descripción del código

La práctica, ha sido desarrollada con la utilización de *"threads"*, para lo cual ha sido necesario implementar los mecanismos necesarios para que exista concurrencia, ejecutando de forma adecuada.

En primer lugar, mediante la siguiente imagen, se va a indicar el funcionamiento deseado de la fábrica:



2.1 Init Factory

Para conseguir éste funcionamiento, mediante un fichero de configuración, se indican los parámetros deseados para la ejecución del código, parámetros como el nº de *"inserters"*, nº de *"receivers"*, y la cantidad de elementos que los *"inserters"* crean, modifican y unidades que se les aumenta a su stock.

Tras la lectura de éste fichero de configuración, se crean los *"threads"* indicados mediante el archivo de configuración. Para la creación de los *"inserters"*, ha sido necesario crear una estructura en la cual se indica su identificador (de 0 a n-1), número de elementos a crear, número de elementos a modificar, y unidades a aumentar en el stock de cada elemento. Una vez creada y rellenada dicha estructura, se crean los *"inserters"*, el *"transporter"* y los *"receiver"*. Se esperará haciendo *"Join"* a la finalización de cada *"thread"*, mientras la ejecución sea sin errores. En caso de producirse un error, se llamará

al cierre de la fábrica y se saldrá de la ejecución del código mediante *"exit()"* con valor 1.

2.2 Inserters

Se encargan de crear nuevos elementos en la base de datos. Cada *"inserter"*, tiene asignados un número de elementos a crear, modificar, y a estos últimos, un número de unidades a incrementar en su stock.

En primer lugar, para poder crear un elemento, será necesario generar un nombre. Se ha tomado la decisión de que el nombre de cada elemento a crear sea único, tomando el siguiente formato concatenado: **idThreadElementoCreadoDelThread**

- **idThread:** Cada *"thread"* tiene un id, con un rango de 0 a n-1, siendo n el número de *"inserters"*.
- **ElementoCreadoDelThread:** Indica el índice del elemento creado por el *"thread"*, estando en un rango de 0 a n-1, siendo n el número de elementos a crear por el *"thread"*.

Una vez generado el nombre, es necesaria la creación de una estructura, mediante la cual poder controlar la concurrencia del elemento. Dicha estructura, es creada mediante la función *"malloc"* para que su creación sea en la zona del *"heap"* en memoria, para que así tras la finalización del método, la estructura permanezca en memoria.

Una vez creados el nombre y la estructura necesaria para la creación del elemento, se crea el elemento, y tras ello, se le pasa la dirección de memoria en la cuál ha sido generada su estructura para concurrencia. Al ser una escritura global de la base de datos, será realizada en una sección crítica protegida.

Una vez el *"thread"* ha insertado todos los elementos que le correspondían, procede a modificar el stock a los n primeros elementos de la base de datos, lo cual, al ser una escritura local, se realizará en una sección crítica local del elemento.

2.3 Transporter

El *"thread transporter"*, se encarga de ir transportando los elementos de la base de datos a la cinta. Realizará su función hasta que transporte todos los elementos que se insertan en la base de datos.

En primer lugar, consulta si hay algún elemento en la base de datos, en caso de no estarlo, esperará hasta que alguno sea insertado. Una vez ha sido insertado el primer elemento, cuando le corresponda su turno, irá iterando en la base de datos desde el primer elemento, hasta que un elemento no esté *"ready"*, en ese caso, si todos los elementos han sido transportados, finalizará, sino, volverá al primer elemento y repetirá su función hasta transportar todos los elementos.

Cuando el elemento está *"ready"*, consulta su stock, si éste es mayor que 0, procederá a mover todos sus elementos a la cinta. Para ello, existe una sección crítica de acceso a la base de datos, en la cual, coge el elemento de la base de datos, y disminuye su stock en una unidad. Una vez realizado, sale de la sección crítica de la base de datos, y entra en una sección crítica de acceso a la cinta para realizar la inserción. La cinta, será un buffer circular, que tendrá un apuntador de entrada, y otro de salida. Para insertar en la cinta, en primer lugar, mirará si existe un hueco en ella, y en caso de no haberlo, esperará hasta que no esté llena. Una vez no esté llena, procede a insertar el elemento en la cinta en la posición que indica el apuntador de entrada. En la cinta, cada posición consta de id del objeto y de su nombre. Una vez insertados estos datos en su posición de la cinta, en caso de que el número de elementos en la cinta sea uno, envía una señal avisando de que la cinta no está vacía y finaliza la sección crítica de la cinta.

2.4 Receivers

Los *"threads receiver"*, se encargarán de ir sacando de la cinta cada elemento insertado. Estarán en funcionamiento mientras el número de elementos sacados sea menor al número de elementos insertados en la base de datos. Su implementación es muy sencilla, ya que constará de una única sección crítica de la cinta, en la que consulta el número de elementos en la cinta y el número de elementos sacados de ella. En caso de que se hayan sacado de la cinta tantos elementos como elementos fueron insertados en la base de datos, cada receiver finalizará su función. En caso contrario, en la sección crítica de la cinta, consultará el elemento de la posición de salida correspondiente, y procede a sacarlo. Una vez sacado, aumento la posición de salida de la cinta, disminuye en uno la cantidad de elementos de la cinta, y en caso de que esta cantidad sea igual al tamaño máximo de la cinta menos uno, enviará una señal indicando que la cinta no está llena, para que el *"transporter"* pueda continuar insertando en la cinta.

2.5 Close Factory

La función *"close factory"*, puede ser llamada en diferentes partes del código. En caso de no producirse ningún error, tras la finalización de toda la función de la fábrica. En caso contrario, en caso de producirse algún error, será llamada para posteriormente salir de la fábrica.

Su función es la de liberar todos los recursos consumidos por la fábrica, destruyendo la base de datos con todos los datos que posee, y realizar todos los "free" correspondientes a las reservas de memoria realizadas para su liberación de memoria.

2.6 Decisiones de diseño

Como detalles a destacar, debemos mencionar algunas peculiaridades del código:

- Al disponer de una variable global, que describe el estado de la base de datos, podemos comprobar si ésta se encuentra disponible (valor igual a 0), si se encuentra en lecturas o escrituras locales (valor igual a 1) o si se encuentra realizando escrituras globales (valor igual a 2).
- Por simplicidad, se ha decidido que el *"thread"* transporter, una vez procede a vaciar el stock de un elemento de la base de datos, continúe con éste hasta que su stock llegue a 0. Pero tal vez sería más óptimo el ir decrementando en una unidad su stock y pasar al siguiente. Ésta opción, tendría como problema, el caso de que únicamente exista un elemento en la base de datos, ya que intentaría iterar al siguiente elemento, y no podría, con su correspondiente pérdida de tiempo de ejecución.
- Nombres únicos para cada elemento de la base de datos, ya que a la hora de depurar las salidas de ejecución, simplifica en gran medida la complejidad.
- Elección de *"pthread_cond_broadcast"* en lugar de *"pthread_cond_signal"*, ya que para hacerlo más aleatorio, hemos creído más conveniente enviar la señal a todos los hilos esperando por la condición que a un único hilo.
- En caso de indicar un número de receivers menor o igual a 0, se da un error, y por tanto se realiza la finalización del programa con *"exit(1)"*, debido a que en caso de llenarse la cinta, el *"transporter"* se quedaría esperando a que algún *"receiver"* le indique que tiene hueco para insertar en la cinta,

pero al no exisitir éste, nunca despertaría, quedándose en bucle infinito.

- En caso de indicar que el número de elementos a modificar por un *"inserter"* o el número de stock a incrementar sea menor que cero, se considerará como un error, finalizando la ejecución con *"exit(-1)"*.

3 Batería de pruebas

En este apartado, vamos a proceder a exponer las pruebas realizadas sobre el código.

Parámetros de configuración no válidos	
Objetivo	Comprobar el comportamiento al insertar parámetros en el fichero de configuración no válidos.
Entradas	1 1 1 1 -1
Salida	Error al leer el fichero de configuración.
Conclusión	En caso de proporcionar un fichero de configuración que no cumpla con los requisitos del enunciado, que proporcione un número negativo como parámetro o que indique un número de receivers menor que 1, se produce un error controlado.

Tabla 1: Parámetros no válidos

Máximo número de threads	
Objetivo	Comprobar el máximo de threads que se pueden crear sin que se produzca una violación de segmento.
Entradas	1 20 1 1 1
Salida	Violación de segmento
Conclusión	Tras realizar varias pruebas, se ha llegado a la conclusión de que el número total de threads a crear entre inserters, transporter y receiver ha de ser como máximo 21 para que no se produzca una violación de segmento.

Tabla 2: Máximo número de hilos

Threads inserter sin creaciones	
Objetivo	Comprobar que se pueden crear inserters que no creen nuevos elementos en la base de datos.
Entradas	2 3 1 1 1 0 0 0
Salida	Export finished. CORRECTO
Conclusión	Si se crean inserters que no tienen que hacer nada, se crean, y tras crearse, se destruyen puesto que en la función no hacen nada, y la ejecución del código se realiza correctamente.

Tabla 3: Inserters sin creaciones

Modificar el máximo número de elementos en base de datos	
Objetivo	Comprobar que se modifican correctamente los primeros 16 elementos de la base de datos, es decir, el máximo número de elementos que pueden crearse en ella.
Entradas	1 3 16 16 2
Salida	Export finished. CORRECTO
Conclusión	Pueden modificarse correctamente todos los elementos de la base de datos.

Tabla 4: Modificar máximo de elementos en base de datos

Crear más de 16 elementos en la base de datos	
Objetivo	Comprobar el funcionamiento al intentar crear más elementos en la base de datos de su máximo permitido.
Entradas	3 3 4 4 2 4 4 4 9 1 1
Salida	Error in create element.
Conclusión	Se verifica que el máximo número de elementos a crear en la base de datos es de 16.

Tabla 5: Crear más elementos en base de datos de su máximo permitido

No crear threads inserter	
Objetivo	Comprobar el funcionamiento al no crear ningún thread inserter.
Entradas	0 3
Salida	Export finished. CORRECTO
Conclusión	En caso de no crear ningún thread inserter, el número de elementos a crear en la base de datos será 0, y por tanto se ejecuta el código de la práctica, pero sin realizar ninguna acción con la base de datos, sin existir ningún tipo de error.

Tabla 6: No crear ningún thread inserter

No crear threads receiver	
Objetivo	Comprobar el funcionamiento al no crear ningún thread receiver.
Entradas	1 0 1 1 1
Salida	Error, receivers menor o igual a 0.
Conclusión	En caso de no crear ningún thread receiver, se produce un error para evitar caer en un bucle infinito.

Tabla 7: No crear receivers

Límite de creaciones de threads	
Objetivo	Comprobar si al crear 21 threads, con 17 inserters, 3 receiver y un transporter, con inserters sin creaciones, y un número elevado de stock total (29), el funcionamiento es correcto.
Entradas	17 3 0 0 0 2 1 1 0 0 0 2 1 1 1 1 1 0 0 0 1 0 0 1 1 2 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 3 1 1 1 1 0 0
Salida	Export finished. CORRECTO

Límite de creaciones de threads	
Conclusión	Al no superar la cuota máxima de threads creados, el funcionamiento es correcto. Si creamos un thread más, se produce violación de segmento como se ha comentado anteriormente.

Tabla 8: Límite de creaciones de threads

4 Conclusiones

Como conclusiones de ésta práctica, cabe destacar la necesidad e importancia de saber el alojamiento de las variables en memoria en función de su reserva de espacio, ya que si una variable o estructura se crea en la pila, no podrá ser accesible en cualquier momento de la ejecución, sino que solo será accesible mientras el método en el cual ha sido creada permanezca en la pila. Por tanto, cuando se requiere que permanezca accesible mientras se necesite de ella, lo correcto es crearla en el *"heap"* o montículo.

También debe destacarse la importancia de una correcta implementación de la concurrencia, ya que su depuración es bastante compleja. Para ello, ha sido necesaria la utilización de *"Valgrind"*, lo que ha sido de gran utilidad. Además, es muy importante tener claro que partes del código han de están protegidas para tener un acceso restringido y no simultáneo, para que no se produzcan colisiones y comportamientos no deseados. Otro aspecto importante, es la liberación de memoria, por lo que gracias a ésta herramienta, hemos podido observar que en caso de una ejecución correcta, se liberan todos los recursos antes del fin de programa, mientras que en caso de producirse algún error en los *"threads"*, no se libera toda la memoria. Esto se debe, ya que en la función *"init_factory()"*, se realizan reservas de memoria mediante *"malloc"*, y al no tener disponible de forma global ésta referencia de reserva de memoria, no podemos desde el los *"threads"* realizar la liberación de memoria debido a nuestra implementación, aunque si sería posible.

Otro problema que hemos encontrado al ejecutar en guernika, ha sido la cuota que te proporciona para la ejecución, ya que hemos podido observar que el máximo número de threads que te permite ejecutar sin producir una violación de segmento, son 21.

Por último, mencionar la peculiaridad de implementación de los sistemas *"lector-escritor"* y *"productor-consumidor"* entrelazados, lo cual ha sido bastante interesante para poder comprobar la dificultad de un buen diseño para una óptima implementación, ya que lo más correcto sería lograr una ejecución lo más rápida posible con los mismos parámetros de configuración.