Universidad Carlos III de Madrid 2013/2014

SISTEMAS OPERATIVOS

Practica 3: Programación de una biblioteca de gestión de stock concurrente (Multithread)

Carlos Contreras Sanz 100303562
Álvaro Gómez Ramos 100307009

Contenido

Esta práctica estaba pensada para iniciarnos en la programación multi hilo, la creación de procesos concurrentes y para controlar las posibles condiciones de carrera que pudiesen darse.

Se trataba de simular un pequeño sistema de control de productos para un almacén, con las operaciones de creado y borrado de producto, consulta del número de los mismo, incremento y decremento del stock de un producto y consulta del stock de cualquiera de ellos.

La práctica se puede dividir en dos parte, la primera el control de los proceso que afectan a la base de datos global, procesos que llamaremos globales (creación, borrado y consulta de número de productos activos), y procesos que llamaremos locales (incremento, decremento y consulta del stock de un producto), ya que actuaran sobre alguno de los productos específicos del almacén.

Se ha implementado de tal forma que borrado y creación de productos sean las únicas operaciones no se puedan ejecutar de forma concurrente con ninguna otra (ni entre ellas), ya sea local o global. De la misma forma, incremento y decremento no pueden ejecutarse de forma concurrente con ninguna otra (ni entre ellas) de forma local (dentro de cada producto). En cambio, las operaciones de consulta del número de productos o de consulta del stock, si pueden realizarse concurrentemente.

De tal forma que todas las operaciones locales se consideran como lecturas a la base de datos global, además de la consulta del número de productos. Por lo tanto, para la creación y borrado de productos se deberá comprobar que no haya lectores globales, y para el incremento y decremento, que no haya lectores locales en ese producto.

Además conviene diferenciar que si hay una operación que bloquea localmente un producto (incremento o decremento de stock) sí que se puede operar sobre otro producto diferente.

Vamos a analizar método por método del código, describiéndolo y explicando las principales decisiones tomadas.

El código

Partes comunes

Para llevar a cabo lo que nos pedían hemos hecho uso de dos mutex, dos variables de condición y 2 contadores, además de dos variables para guardar los máximos números de productos y máximos de lectores concurrentes.

> mut_global: mutex que se usara para controlar la concurrencia global, y proteger las zonas necesarias del cogido.
>
> con_lectores: variable de condición que se usara para parar y despertar procesos dadas una serie de condiciones globales (número máximo de lectores, número máximo de productos creados)
>
> no_lectores: número contador para saber cuántos lectores hay en un determinado momento sobre la base de datos global.

max_lectores: número que se usa como límite para lectores concurrentes (globales), en principio se iguala a MAX_READERS en el concurrent_init ().

no_productos: número contador para saber cuántos productos hay creados en un determinado momento (y no tener que hacer llamadas a ningún método).

max_prod: número que se usa para saber el límite de productos creados que podemos tener, por defecto se pone a 16 en el concurrent_init ().

struct local: estructura que se usa para pasar a un producto recién creado, un mutex, variable de condición y contador para controlar concurrencia local. Formada por:

mut_local: mutex para controlar concurrencia sobre ese producto.

con_lec_local: variable de condición para dormir y despertar procesos locales.

leyendo_local: número contador, para saber cuándo puede entrar un escritor local (cuando no haya lectores locales).

En el método concurrent_init () se inicializa el mutex global y la variable de condición global, además de los contadores a 0, el máximo de productos a 16 y se iguala el máximo de lectores a MAX_READERS. Por último se llama a db-warehose_init ().

En el concurrent_destroy () se destruyen el mutex global y la variable de condición global, además de hacer una llamada a db_warehouse_destroy ().

int concurrent_create_product(char *product_name)

En este método se ha implementado la creación de un producto del almacén. Para ello lo primero ha sido controlar la concurrencia, y asegurarnos de que somos los únicos haciendo algo en toda la base de datos. Usamos para ellos un lock () del mutex global, y comprobamos en bucle que no tenemos lectores (esperando si es preciso, con un wait () de la variable de condición global, y liberando el mutex hasta que pudiésemos seguir).

Cuando en efecto no los tenemos, comprobamos si existe ese producto, en caso negativo, lo creamos. Además creamos una nueva estructura de tipo local, inicializamos sus atributos (mutex, variable de condición y contador) y se la asignamos a ese producto (con los datos de esa estructura, será con lo que controlaremos la concurrencia local). Tras esto incrementamos el contador de productos creados.

Por último, enviamos signal () a los procesos que estuviesen dormidos por la variable de condición global, y liberamos el mutex.

int concurrent_delete_product(char *product_name)

De la misma forma que en el caso anterior, lo primero es comprobar que somos los únicos actuando en la base de datos. Seguimos los mismos pasos para asegurarnos de ello (bloquear el mutex y comprobar que no hay lectores).

Una vez que estamos seguros de que somos los únicos actuando, obtenemos la estructura del producto. Destruimos su mutex, su variable de condición y liberamos la memoria. Actualizamos los datos de la estructura en el producto (tamaño a 0 y puntero a ningún sitio), y procedemos a borrar el producto, y descontar uno del total de productos activos.

Por ultimo despertamos a los dormidos por la variable de condición global y liberamos el mutex.

Una vez en este punto, nos damos cuenta de que es posible que no sea haya borrado un producto (porque no existe) y aun así se descuenta uno del contador de productos activos.

Se solucionaría añadiendo una comprobación de ret==0 (el del delete) y entonces, si eso se cumple, descontamos del contador. Pero hemos entendido que esto no debíamos modificarlo, de modo que no lo hemos llegado a implementar.

int concurrent_get_num_products(int *num_products)

Para este método, hay que tener en cuenta, que sus acciones en sí mismas, ni hay que protegerlas, y han de ser concurrentes con otras lecturas. Por lo tanto solo protegeremos las modificaciones de variables globales (del contador de lectores globales).

Por tanto empezamos protegiendo con el mutex global y comprobando que no hemos llegado al máximo de lectores. Si esto se cumple, incrementamos los lectores y liberamos el mutex.

Realizamos la operación de consulta, y una vez finalizada, volvemos a proteger mutex para decrementar el contador de lectores globales, despertamos a los que estuviesen parados por la variable de condición global y liberamos el mutex.

int concurrent_increment_stock(char *product_name, int stock, int *updated_stock)

En este caso, se realiza una operación de lectura a nivel global y una de escritura a nivel local, por lo que lo que haremos será añadir uno al contador de lectores globales, y bloquear el mutex local.

Empezamos bloqueando el global (esperando a que el número de lectores actuales sea menor del máximo) e incrementamos el contador de lectores globales. Tras eso despertamos a los dormidos globalmente y liberamos el mutex global.

Con eso (y el código del crear y del borrar) nos aseguramos de que no nos borren ese producto mientras lo estamos editando (ya que no se crearan o borraran productos si hay lecturas globales). Además al haber liberado el global, hacemos que puedan entrar más lectores.

Tras eso obtenemos la estructura que almacena mutex y demás variables necesarias para controlar la concurrencia local. De la misma forma que controlamos la global, vamos a controlar la local, primero lock () al mutex local y luego un bucle en que se compruebe que no hay nadie leyendo este producto (o si no wait () por la variable de condición local).

Una vez estamos seguros de que somos los únicos actuando sobre ese producto, realizamos las operaciones necesarias para incrementar el stock en las unidades que nos hayan indicado.

Tras eso, hacemos signal () por la variable de condición local, y liberamos el mutex local. Por último, volvemos a bloquear mutex global para decrementar lectores globales, hacer signal () por la variable de condición global y liberamos el mutex global.

int concurrent_decrement_stock(char *product_name, int stock, int *updated_stock)

Este método es muy similar al anterior, con la diferencia de que en lugar de aumentar un stock, lo que hacemos es disminuirlo, decrementarlo. Por tanto funciona de la misma forma.

En este caso, se realiza una operación de lectura a nivel global y una de escritura a nivel local, por lo que lo que haremos será añadir uno al contador de lectores globales, y bloquear el mutex local.

Empezamos bloqueando el global (esperando a que el número de lectores actuales sea menor del máximo) e incrementamos el contador de lectores globales. Tras eso despertamos a los dormidos globalmente y liberamos el mutex global.

Con eso (y el código del crear y del borrar) nos aseguramos de que no nos borren ese producto mientras lo estamos editando (ya que no se crearan o borraran productos si hay lecturas globales). Además al haber liberado el global, hacemos que puedan entrar más lectores.

Tras eso obtenemos la estructura que almacena mutex y demás variables necesarias para controlar la concurrencia local. De la misma forma que controlamos la global, vamos a controlar la local, primero lock () al mutex local y luego un bucle en que se compruebe que no hay nadie leyendo este producto (o si no wait () por la variable de condición local).

Una vez estamos seguros de que somos los únicos actuando sobre ese producto, realizamos las operaciones necesarias para decrementar el stock en las unidades que nos hayan indicado.

Tras eso, hacemos signal () por la variable de condición local, y liberamos el mutex local. Por último, volvemos a bloquear mutex global para decrementar lectores globales, hacer signal () por la variable de condición global y liberamos el mutex global.

int concurrent_get_stock(char *product_name, int *stock)

Por último, en este método se va a llevar a cabo una lectura global y una local. De la misma forma que en ocasiones anteriores, bloqueamos el mutex global y esperamos hasta que los lectores sean menores al límite. Una vez comprobado aumentamos en uno los lectores globales, y liberamos el mutex (haciendo signal () a la variable de condición global).

Ahora, procedemos a obtener la estructura que contiene los datos necesarios para controlar la concurrencia local. Una vez obtenida, hacemos lock () del mutex local, en incrementamos la variable local de lectores. Tras eso liberamos el mutex local.

En este punto no tenemos ningún mutex bloqueado, pero si hemos añadido un lector al contador global y otro al local (es por tanto concurrente).

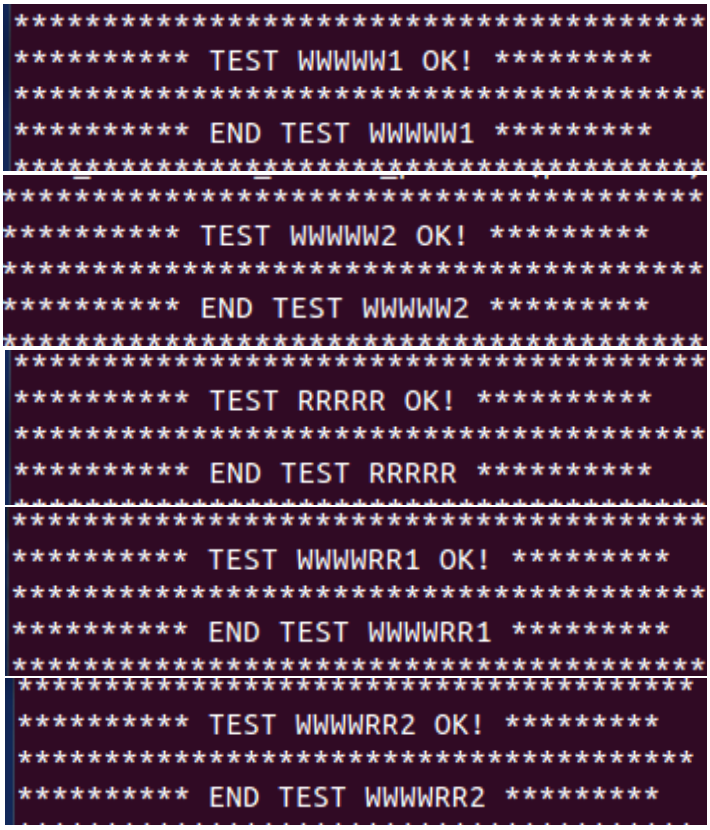
Realizamos la operación para obtener stock, y realizamos el proceso inverso.

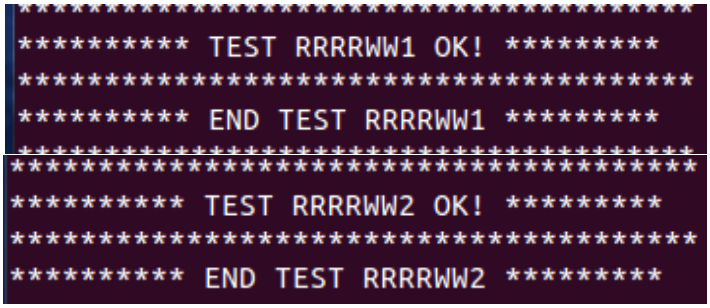
Primero decrementamos el número de lectores locales y hacemos signal a la variable de condición local (con el mutex local bloqueado), y después hacemos lo propio con las variables globales, decrementando los lectores globales y haciendo signal en la variable de condición global.

Las pruebas

Las primeras pruebas a las que hemos sometido al código, han sido las de nivel global. Es decir, que la creación y destrucción de productos no pueden ser concurrentes con nada, ni entre ellos, y que la obtención del número de productos si debe ser concurrente.

Para ellos nos hemos valido del ejemplo proporcionado por los profesores de la asignatura, pasando todos los test:





Creando y borrando la base de datos de forma satisfactoria.

Con esto y alguna pequeña prueba nuestra mas, hemos comprobado el correcto funcinamiento de la creacion, borrado y consulta del numero de productos.

Para probar los metodos que afectan a productos (locales) y su integracion con los aneriores, de la base de datos entera (globales), hemos llevado a cabo una serie de pruebas en las que creabamos productos, incrementabamos el stock, lo decrementabamos, lo consultabamos, consultabamos el numero de productos y borrabamos esos productos. Una prueba de todos los metodos por tanto.

Para estas pruebas se han creado una serie de threads para lanzar los metodos (reutilizando los dados por los profesores de la asignatira para la prueba global), ademas de creando unas estructuras para pasar cantidades a incrementar y decrementar de los productos:

```
struct update_stock_st *link=(struct update_stock_st *) malloc(sizeof(struct update_stock_st));
sprintf(link->product, "producto_1");
link->stock=1;
struct update_stock_st *link_2=(struct update_stock_st *) malloc(sizeof(struct update_stock_st));
sprintf(link_2->product, "producto_2");
link_2->stock=10;
```

En una primera prueba hemos creado dos productos, y hemos hecho incrementos de producto, 15 en caso del primero y 5 incrementos en caso del segundo. Por las cantidades pasadas a las estructuras, deberíamos obtener como resultados 15 y 50.

Si obtenemos dicho resultados, significaría que no se han producido condiciones de carrera, y que todo ha funcionado como debería

.

Se han ido creando en este orden:

```
pthread_create (&th_test[2], NULL, increment_stock_thread, link);
pthread_create (&th_test[3], NULL, increment_stock_thread, link);
pthread_create (&th_test[4], NULL, increment_stock_thread, link);
pthread_create (&th_test[5], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[6], NULL, increment_stock_thread, link);
pthread_create (&th_test[7], NULL, increment_stock_thread, link);
pthread_create (&th_test[8], NULL, increment_stock_thread, link);
pthread_create (&th_test[9], NULL, increment_stock_thread, link);
pthread_create (&th_test[10], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[11], NULL, increment_stock_thread, link);
pthread_create (&th_test[12], NULL, increment_stock_thread, link);
pthread_create (&th_test[13], NULL, increment_stock_thread, link);
pthread_create (&th_test[14], NULL, increment_stock_thread, link);
pthread_create (&th_test[15], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[16], NULL, increment_stock_thread, link);
pthread_create (&th_test[17], NULL, increment_stock_thread, link);
pthread_create (&th_test[18], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[19], NULL, increment_stock_thread, link);
pthread_create (&th_test[20], NULL, increment_stock_thread, link);
pthread_create (&th_test[21], NULL, increment_stock_thread, link_2);
```

Obteniendo como resultado:

```
_______EL STOCK DE producto_1 AHORA ES DE 15
       EL STOCK DE producto_2 AHORA ES DE 50
```

Vistos los resultados positivos de esta, hemos introducido decrementos de stock, concretamente dos decrementos de 10 unidades para el segundo producto, de la siguiente forma:

```
pthread_create (&th_test[2], NULL, increment_stock_thread, link);
pthread_create (&th_test[3], NULL, increment_stock_thread, link);
pthread_create (&th_test[4], NULL, increment_stock_thread, link);
pthread_create (&th_test[5], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[6], NULL, increment_stock_thread, link);
pthread_create (&th_test[7], NULL, increment_stock_thread, link);
pthread_create (&th_test[8], NULL, decrement_stock_thread, link_2);
pthread_create (&th_test[9], NULL, increment_stock_thread, link);
pthread_create (&th_test[10], NULL, increment_stock_thread, link);
pthread_create (&th_test[11], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[12], NULL, increment_stock_thread, link);
pthread_create (&th_test[13], NULL, increment_stock_thread, link);
pthread_create (&th_test[14], NULL, increment_stock_thread, link);
pthread_create (&th_test[15], NULL, decrement_stock_thread, link_2);
pthread_create (&th_test[16], NULL, increment_stock_thread, link);
pthread_create (&th_test[17], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[18], NULL, increment_stock_thread, link);
pthread_create (&th_test[19], NULL, increment_stock_thread, link);
pthread_create (&th_test[20], NULL, increment_stock_thread, link_2);
pthread_create (&th_test[21], NULL, increment_stock_thread, link);
pthread_create (&th_test[22], NULL, increment_stock_thread, link);
pthread_create (&th_test[23], NULL, increment_stock_thread, link_2);
```

En este caso nos hemos preguntado si el orden en que se ejecutan afectaría al resultado, ya que podría darse el caso de restar unidades cuando no hay ninguna actualmente (si existe el producto, pero su stock es 0).

De forma que primero hemos comprobado que obtendríamos en este caso:

```
_______EL STOCK DE producto_2 AHORA ES DE 30
_______EL STOCK DE producto_1 AHORA ES DE 15
```

Y como hemos visto que era todo correcto, hemos añadido al código anterior el decremento de 40 unidades para el producto dos después del código arriba mostrado:

```
pthread_create (&th_test[24], NULL, decrement_stock_thread, link_2);
pthread_create (&th_test[25], NULL, decrement_stock_thread, link_2);
pthread_create (&th_test[26], NULL, decrement_stock_thread, link_2);
pthread_create (&th_test[27], NULL, decrement_stock_thread, link_2);
```

Obteniendo como resultado:

```
_______EL STOCK DE producto_2 AHORA ES DE -10
_______EL STOCK DE producto_1 AHORA ES DE 15
```

Con esto nos hemos dado cuenta de que es posible tener stock negativo, y esa es la razón por la que (partiendo de la base de que los incrementos y decrementos funcionan correctamente y sin errores) se ejecuten en el orden que se ejecuten, el resultado siempre será el esperado.

Para la siguiente prueba se procedió a intercalar incrementos, decrementos, consultas de stock de número de productos, creaciones, borrados y consulta de tiempos (en los casos menos difíciles).

En todas las pruebas se ejecutó de la forma esperada y sin errores. Para comprobar este punto, hemos introducido trazas en el código informándonos de la situación actual en cada momento y la acción que se está realizando. Algunas las hemos quitado, pero varias de ellas las hemos dejado comentadas.

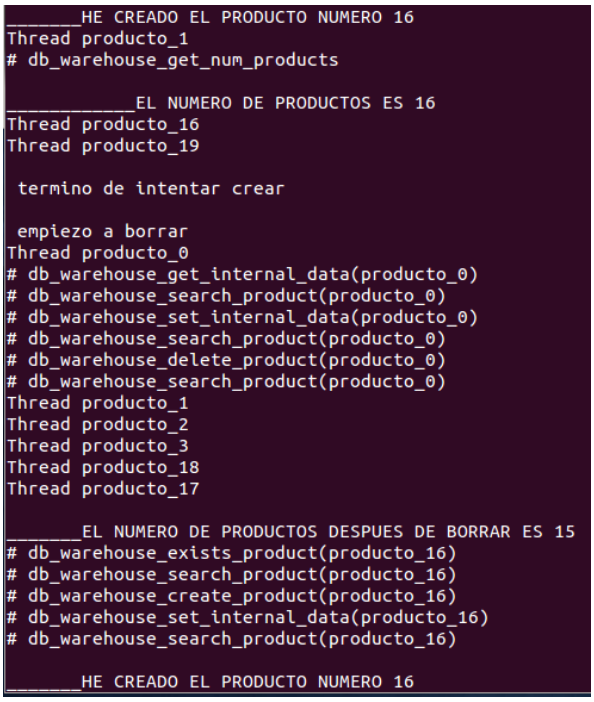
Llegados a este punto, lo único que nos quedaba por probar era el límite de productos (el límite de lectores ya se cubrió en la anterior). De modo que nos creamos un nuevo caso, intentando crear más de 16 productos (que por defecto es el límite). Obtuvimos que, en caso de añadir pthread_join después de las creaciones, el programa se pararía y no continuaría la ejecución, dado que las creaciones por encima de la numero 16, estarían bloqueadas en la variable de condición global, y sin borrados de productos, nadie les despertaría, o bien no variaría la condición que les tenia dormidos (el haber llegado al límite de productos creados).

Añadimos por tanto borrados (después de las creaciones) y procedimos a ejecutar con las trazas antes mencionadas, para ver cómo se comportaba el programa.

Teniendo un código tal que:

```
        for (j=0;j<15;j++){
                pthread_create (&th_test[j], NULL, create_product_thread, producto[j]);
        }
        for (k=0;k<15;k++){
                pthread_join (th_test[k], (void **) &st1);
        }
/*15*/  pthread_create (&th_test[15], NULL, count_products_thread, producto[1]);

        pthread_join (th_test[15], (void **) &st1);
        pthread_create (&th_test[16], NULL, create_product_thread, producto[15]);
        pthread_join (th_test[16], (void **) &st1);
/*16*/  pthread_create (&th_test[17], NULL, count_products_thread, producto[1]);
        pthread_join (th_test[17], (void **) &st1);

        pthread_create (&th_test[18], NULL, create_product_thread, producto[16]);
        pthread_create (&th_test[19], NULL, create_product_thread, producto[17]);
        pthread_create (&th_test[20], NULL, create_product_thread, producto[18]);
        pthread_create (&th_test[21], NULL, create_product_thread, producto[19]);
printf("\n termino de intentar crear\n");

printf("\n empiezo a borrar\n");
        pthread_create (&th_test[22], NULL, delete_product_thread, producto[0]);
        pthread_create (&th_test[23], NULL, delete_product_thread, producto[1]);
        pthread_create (&th_test[24], NULL, delete_product_thread, producto[2]);
        pthread_create (&th_test[25], NULL, delete_product_thread, producto[3]);

        for (k=18;k<25;k++){
                pthread_join (th_test[k], (void **) &st1);
        }
printf("\n SI deberia llegar aqui\n");
/*20*/  pthread_create (&th_test[26], NULL, count_products_thread, producto[10]);
        pthread_join (th_test[26], (void **) &st1);
printf("\n deberia de haber impreso el numero de productos\n");
```

El resultado es:



Es decir, tras crear el producto número 16, a pesar de que lo siguiente que le llega es otra creación, no la ejecuta, ya que ha llegado al máximo, y no es hasta que se ejecuta un borrado, que puede seguir ejecutando.

Con estas pruebas creemos que cubrimos los posibles escenarios interesantes para esta práctica, que son los relacionados con concurrencia y gestión de condiciones de carrera.

Conclusiones

Para terminar, decir que como siempre, se agradecen estas prácticas y sus explicaciones para similar mejor los conceptos de teoría, y ver su aplicación práctica. Sería bastante más difícil la parte teórica si no tuviésemos estas prácticas, que aunque trabajo, también son un apoyo y bastante útil por cierto.

Una vez entendida la teoría y la dinámica y estructura de la práctica, que no fue trivial, las principales dificultades las encontramos en cómo controlar la concurrencia local y en llegar a la conclusión de usar una estructura para pasar y obtener los datos necesarios para ello.

Tras superar esa punto, algunos detalles que, al ejecutar las pruebas provocaron que no se obtuviese el resultado esperado (o no se terminase de ejecutar nunca) fue la falta de algunos signal ().

Creemos que entregamos un ejercicio bastante completo (y correcto) y que cumple todo lo que de él se esperaba.