Universidad Rey Juan Carlos

Memoria Práctica Final:
Los Almacenes de Santa Claus

Presentado por: *Fernando González Ramos*
Titulación: *Ingeniería Telemática*
Fecha: *01/02/2021*

- Índice

1.- Introducción

Este programa se trata de una extensión de la funcionalidad del programa anterior. Por lo tanto, de aquí en adelante, haré referencia exclusivamente a los cambios que se han llevado a cabo para la implementación de dicha extensión. Para conocer el funcionamiento principal del programa, el lector deberá hacer uso de la memoria de la *Práctica 2*.

El problema añadido principal que entraña este programa es que a partir de ahora, los elfos no construyen juguetes sin parar sino que, encontramos dos batallones de elfos integrados por 16 de los mismos cada uno. De esos 16 elfos que conforman cada batallón, uno es el **líder**. A continuación, explico con mayor detalle, el papel del líder en cada uno de los batallones:

1.1.- Papel del Líder:

La fábrica de juguetes posee un almacén que contiene todos los juguetes que hay que preparar. A partir de ahora, nos referiremos a este almacén como *Almacén Principal.*

Además, cada batallón de elfos posee su propio almacén, al que llamaremos, *caché*.

Por lo tanto, la tarea del líder es recoger periódicamente un juguete del almacén principal y llevarlo a sendas memorias caché de los batallones. De esta manera, ambos líderes “leen” del almacén principal (simulando una memoria principal) y “escriben” en ambas cachés (simulando memorias caché). Este proceso de escritura se efectúa siguiendo un protocolo denominado *Write Through*, esto es, se escribe en ambas cachés al mismo tiempo.

Por lo tanto, en este programa, los elfos no deben construir juguetes “ficticios” sin parar, sino que deben construir los regalos que se encuentren almacenados en sus respectivas cachés.

Por último, y como novedad, ¿qué ocurre si el almacén principal se vacía por completo? En este caso, aunque aún no hayan llegado todos los renos de sus vacaciones, el programa debe terminar, pues no hay más regalos que preparar. De modo que el programa posee dos puntos de salida: Que lleguen todos los renos, o que se acaben los juguetes del almacén principal

2.- Estructura General del Programa

No hay mucho más que añadir con respecto a la estructura general del programa anterior. Como novedad, y, de nuevo, por razones estéticas, de claridad, organizativas y de comprensión, he decidido fragmentar mi programa en una librería más, la cual contiene todos los tipos, métodos y funciones que modelan a los elfos. Este es el paquete *elves*.

3.- Estructura Detallada del Programa

3.1.- Tipos de Datos:

A continuación, se presentan los tipos de datos modificados, es decir, los que ya existían pero han sufrido variaciones; y los nuevos.

1º ToyFactoryType: Este tipo de datos se encuentra, tal como se indica en la memoria anterior, en el paquete *toy_factory*. Este tipo contenía el atributo *Time_to_deal_ch*, un canal mediante el cual, se notificaba que era momento de repartir porque los renos ya habían llegado.

Ahora, se posee un canal más denominado *Presents_finished_ch*, encargado de notificar que el almacén principal está vacío y que por tanto, no hay más juguetes que fabricar y hay que terminar la ejecución del programa.

Tal como se aprecia, este atributo también es *exportado,* pues ha de ser visible desde el programa principal.

El resto de modificaciones y de nuevos tipos se encuentran en el paquete *elves* y los detallo a continuación:

2º ElvesType: Tipo nuevo que contiene como atributos:

· los batallones (Battalions), formado por un array de *ElvesBattalionType*.

· *U*n slice (array dinámico en GoLang) de canales denominado *Waiting_elves d*onde se irán guardando los canales que cada elfo tiene reservado para despertarlo después de esperar a que Santa Claus haya terminado de ayudarle. Esto es necesario debido a que ya no tenemos un grupo de N elfos sino que hay dos batallones separados. Por lo tanto, al no encontrarse todos los elfos en el mismo array, necesito de este atributo para que Santa Claus "sepa" qué están esperando por él y poder comunicarles que ya terminado de ayudarlos.

· *main_store* de tipo *CacheType*, que modela el almacén (memoria) principal.

· *main_store_empty_ch,* se trata de un canal que se inicializa apuntando a la misma dirección de memoria que *Presents_finished* que posee *ToyFactoryType.* Su función es, por tanto, notificar que el almacén está vacío.

· Por último, contiene el atributo *Start_working_*ch que sirve para notificar que se ha traído un regalo nuevo a las cachés y, por tanto, algún elfo (el que primero lea del canal, por eso es unbuffered, que garantiza una correcta sincronización) debe ponerse con la preparación de dicho juguete. **Nota: Esto no era necesario para la implementación de la extensión, sin embargo, lo he incluido como mejora, ya que facilita la implementación, reduce la extensión del código y ofrece un mejor rendimiento en comparación con el sistema que se implementaba anteriormente basado en una variable incremental protegida con un read-write mutex.**

3º ElvesBattalionType: Hace referencia a un batallón de elfos. Contiene:

· Elves: array de 16 – 1 elfos (en este caso) de *ElfType*.

· cache: de tipo *CacheType*. Es la caché del batallón.

4º ElfType: Caracteriza a un elfo. Contiene:

· problems: Indica si el elfo se encuentra en problemas (su juguete ha producido un fallo). No se requiere para el funcionamiento operativo de la práctica pero es un parámetro de interés que permite su escalabilidad y, por esa razón, lo he mantenido.

· is_working: Indica si el elfo está trabajando en un instante dado o no (porque está esperando a que haya juguetes para fabricar en la caché o porque está esperando a que Santa Claus resuelva el fallo con su juguete).

· battalion: Batallón a que pertenece. Empieza en 0, ya que indica el índice del array de batallones. Por lo que, el 0 hace referencia a que pertenece al batallón 1; el 1, al 2, etc.

· mutex: RWMutex utilizado para acceder convenientemente en lectura o escritura a cualquiera de los atributos anteriores.

5º CacheType: Contiene el atributo *mutex* para sincronizar el acceso a la caché y el atributo *elems* que se trata de un array bidimensional de tipo bool, donde el booleano indica si hay o no hay un regalo en dicha posición.

3.2.- Funciones (Métodos):

Algunos de los métodos presentes en el paquete *elves,* son los mismos que los que había en *toy_factory* (sin ninguna modificación), que han sido movidos a este nuevo paquete por cuestiones de organización y legibilidad del código. Estos métodos son *SetWorking, SetProblems* e *IsWorking.*

En este momento, comenzaré a hablar de los que ya existían pero han sido modificados para su adaptación a las especificaciones del problema:

Todas las siguientes pertenecen al paquete *elves*.

1º Init: Opera sobre *ElvesType*. Este método, inicializa todo lo relativo a los elfos. En primer lugar, inicializa el almacén principal como “lleno”, es decir, todas sus posiciones ocupadas (esto lo hace el método al que se invoca llamado *setAll(bool)*). Posteriormente, inicializa el mutex y ambos canales. Por último, para cada uno de los batallones de elfos, se inicializa el mismo llamando a *initBattalion(int32)* y se lanza la Go rutina que ejecutará el comportamiento del líder.

2º WaitForHelp: Este es el método que se ejecuta mientras el elfo está esperando a que Santa Claus solucione su problema. Lo único que hace es escuchar en su canal *wake_up_ch* mencionado anteriormente y, si llega algo, el elfo se configura como “sin problemas” y “trabajando”. Además, se imprime una traza* para que se sepa qué se está haciendo.

***NOTA:** El uso de trazas, resulta notablemente pernicioso para la buena ejecución del programa, ya que lo ralentiza notoriamente pero, ciertamente son necesarias para que al probar su funcionamiento se sepa lo que se está haciendo.

3º GetBattalion: Opera sobre un elfo (tipo *ElfType*) y devuelve el índice (empezando por 0, como se dijo anteriormente) del batallón al que pertenece.

4º AddWaitingElf: Añade el canal *wake_up_ch* del elfoq eu recibe por parámetro a la lista de elfos esperando *Waiting_elves*. Gracias a esto, Santa Claus puede notificar a los elfos a los que estaba ayudando que ya pueden volver a sus tareas.

A continuación, encontramos una serie de funciones y métodos *unexported* del paquete *elves*. De todos ellos, el único que creo necesario explicar de forma más detallada es el método *runLeaderBehavior.*

5º runLeaderBehavior: Ejecuta el comportamiento del líder el batallón. Se ejecuta en una Go rutina, por lo que como tenemos dos batallones, habrá dos Go rutinas ejecutando este método. Este hecho, planteaba el siguiente problema… ¿Cuándo esta subrutina debe notificar que el almacén principal está vacío y terminar? ¿Cuando se haya accedido a todas las posiciones de la memoria? Pero… si ambas Go rutinas acceden a la misma memoria, ¿qué pasa si una de ellas ha accedido 35 veces y la otra, 55? Ninguna suma noventa, pero la suma sí. Por lo tanto, la memoria está vacía.

La solución inmediata a este problema es crear un método que dada una memoria (de tipo *CacheType*) devuelva si está vacía (todas sus posiciones se encuentran a false). Sin embargo, esta solución sería muy lenta e ineficiente, puesto que con un nivel de contienda tan alto, la región crítica es demasiado grande (90 posiciones en este caso). ¿Cuál ha sido la solución planteada? La explico a continuación:

Antes de lanzar las Go rutinas (en el método *Init*), se crea una variable de tipo puntero a int32 que hace referencia al número de accesos a la memoria que se han hecho (la posición de memoria a la que apunta se inicializa a 0). También se crea otra variable de tipo *RWmutex*. Estas dos variables se pasan como argumento a las Go rutinas, por lo tanto, ambas comparten el mutex y la variable incremental. De esta manera, ambas rutinas incrementan de forma **síncrona** la variable que representa el número de accesos a la memoria. De este modo, cuando alcance un valor igual al número de filas por columnas (de la memoria principal), la rutina lo notifica y termina. Y, si aún no ha de acabar, ¿qué hace?

Obtiene aleatoriamente una fila y una vez en ella, obtiene aleatoriamente una columna. Si está vacía, lo vuelve a intentar. Se repite este proceso hasta que se obtenga una posición ocupada por un juguete. En caso de dar con una posición ocupada, se configura dicha posición como vacía (haciendo uso del método *setAs(int32, int32, bool)*), se escribe el dato en ambas cachés (haciendo uso del método *writeOnChaches()*) y se duerme un tiempo aleatorio entre un rango definido como contantes antes de volver a empezar.

4.- Otras Consideraciones

En cuanto al paquete *santa_claus*, la única modificación realizada es la que atañe al método *WakeUp()* cuya función es despertar a Santa Claus. En la versión anterior, si Santa Claus se encontraba ya trabajando cuando se llamaba a este método, no se le despertaba (pues ya estaba despierto). Sin embargo, en la ejecución del comportamiento de Santa Claus (programado en *toy_factory)* ya se prevé que cuando ha de ayudar a un grupo de elfos, primero se espere a finalizar el trabajo actual (si lo tuviera). Por lo tanto, aunque no afectaba de forma perniciosa al funcionamiento del programa, lo consideraría un *bug*, aunque menor, bien es cierto. Esto ha sido corregido de modo que se “despierta” a Santa Claus siempre que un grupo de elfos lo requiere y será éste el que de forma ordenada vaya solucionando los problemas.