Qué es un Thread

Es un subproceso, que representa una secuencia de instrucciones que puede ejecutar una CPU, dándole un tiempo determinado de posesión de una CPU (dado según el scheduler del sistema operativo). Suelen estar asociados a funciones que se ejecutan en un proceso, pero que a su vez permiten que otras partes del proceso se estén ejecutando. Dos threads de un mismo proceso pueden o no estar en el mismo procesador del computador, dependiendo esto de la cantidad de threads versus la de procesadores, del lenguaje que se esté utilizando, entre otras cosas.
Dentro de cada proceso puede haber más de un thread, en este caso decimos que existe multithreading. Si tenemos más de un thread en un proceso estos comparten ciertas características como el espacio de memoria, variables globales, procesos hijos entre otras variables generales de un proceso. Pero entre dos thread de un mismo proceso, no se comparte ni el stack, ni registros de la CPU.

Los threads tienen diversas utilidades para un programador. En primer lugar pueden ser utilizados para distribuir las tareas de un programa de modo de poder entregarle una responsabilidad a cada thread. Esto puede significar una mejora en rendimiento del programa, ya que se tienen más recursos de hardware realizando tareas, o simplemente ser necesarios para la lógica de un programa, como es el caso de un juego de más de un personaje, donde cada uno debe ser representado como un thread (o una extensión de estos), de modo de poder ver movimientos simultáneos de todos los personajes.
En segundo lugar, tenemos la posibilidad de paralelizar una misma sección de código, permitiendo que dos threads estén ejecutando una acción, de modo de ejecutarla más rápido. Por ejemplo, si queremos enviar 1.000.000 de emails, es más rápido tener 4 threads que envíen 250.000 emails cada uno (en caso de tener 4 procesadores).
Otro caso de uso es la programación en base a eventos, de modo de que cada vez que se gatille un evento, se ejecute un nuevo thread que realice lo gatillado, sin interrumpir las distintas acciones que ejecuta el proceso.
Así como estas, hay muchas utilidades que un programador le puede dar a un thread, y depende del caso el cómo se debe usar.

Sincronización de threads

La sincronización de threads se refiere a técnicas que se utilizan para permitir una correcta ejecución de dos o más threads en un mismo proceso, ya sea cuando dos quieren acceder a un mismo trozo de código o memoria, o cuando debemos ejecutar acciones en un orden determinado. Es sumamente importante para un programador hacer un correcto uso de la sincronización de múltiples threads, ya que, si no se ejecuta correctamente se puede perder la capacidad de un programa de ser determinista, es decir, si tengo dos threads, donde cada uno debe imprimir resultados en cierto orden. Si no están sincronizados, dos ejecuciones iguales del programa, pueden tener resultados distintos(por lo que sería un programa no determinista), si es que el sistema operativo les entrega distintos ordenes de uso de la cpu para cada una de las distintas ejecuciones.

Por otro lado, se puede perder la estabilidad de un programa, es decir, tener errores por el no preocuparse de mantener a dos threads coordinados, como sería el ejemplo de eliminar un elemento de una lista ligada, y que otro thread quiera acceder a este posteriormente. Este elemento no existirá, por lo que tendremos un error del tipo Error de Índice, que implica que estamos fuera del rango de la lista ligada(esto depende del lenguaje,pero en general este error existe).

También, si no ejecutamos correctamente la sincronización de threads, puede ocurrir que uno de estos se adueñe de alguna sección de código, sin dejar acceder a otros, haciendo un programa más ineficiente, o inútil, dependiendo del caso. Un ejemplo de esto sería un programa que desea agregar resultados de distintos thread a una lista, pero si uno de estos se adueña de la lista durante la ejecución, no le permitirá agregar los resultados a los distintos threads.

Siguiendo la misma línea tenemos la necesidad de un programa consistente, donde si unthread modifica algo, queremos que todos los otros threads se enteren, y hagan el cambio respectivo.

La sincronización está manejada principalmente por el programador, que debe utilizar los elementos que le entrega el hardware para lograr que la lógica de su programa funcione correctamente. Para esto el desarrollador debe lograr repartir los recursos del programa de forma justa, y bloquear accesos cuando sea necesario, con herramientas que serán explicadas más adelante.

Problemas que enfrenta la sincronización y sus soluciones e implementaciones

Hay algunos problemas clásicos que enfrenta la sincronización, debido, entre otras cosas, a que los *threads* a menudo comparten recursos. Se mencionan algunos que son relevantes, y se proponen soluciones y sus implementaciones para resolver dichos problemas.

Un problema sencillo: condición de carrera

La primera aproximación a un problema se puede ver simplemente como dos threads intentando modificar una misma variable. Ambos threads tienen que realizar estos pasos:

1. Leer la variable
2. Modificar su valor
3. Escribir su nuevo valor en el mismo lugar donde estaba la variable original

Si el problema no es abordado considerando casos límites, el comportamiento puede ser distinto al esperado, teniendo una **race condition**, es decir, una condición de carrera. Se

puede pensar como que los dos threads compiten entre ellos, como en una carrera, y el resultado depende de cuál de los dos llegue antes.

Instante	**Thread 0**	**Thread 1**	**Valor de x**
1	Lectura de x		5
2	x = x + 3		5
3	Escritura de x		8
4		Lectura de x	8
5		x = x + 1	8
6		Escritura de x	9

Tabla 1: ejecución no simultánea

Instante	**Thread 0**	**Thread 1**	**Valor de x**
1	Lectura de x		5
2		Lectura de x	5
3		x = x + 1	5
4	x = x + 3		5
5	Escritura de x		8
6		Escritura de x	6

Tabla 2: ejecución simultánea

Este problema, en apariencia tan sencillo, puede tener al menos dos soluciones distintas, que se verán implícitamente en los problemas enunciados más adelante.

Problema del productor y el consumidor

Este problema consiste en un modelo en que se tiene al menos un *thread* que está encargado de producir unidades de un ítem determinado, y al menos otro que está encargado de consumir unidades. La idea es que estos threads están insertando y removiendo ítemes de una *cola* o *queue*. Esta cola puede quedar vacía en un momento

determinado, como también puede llenarse en otro momento, debido a que su capacidad es limitada.

En la práctica, los productores y consumidores están trabajando tan pronto como les es posible; es decir, el productor va a hacer su labor apenas tenga el espacio disponible, y lo mismo va a suceder con el consumidor. Entonces, cada productor y consumidor debe tener información acerca de cuántos ítems existen en cada momento, para bloquear las operaciones en caso de que no se puedan seguir ejecutando.

Un primer modelo con un consumidor y un productor

En este caso, se tiene solamente un consumidor y un productor y es necesario coordinarlos entre ellos. Una implementación posible es que el consumidor y el productor están verificando constantemente la cantidad de ítems que hay en la cola, y si se verifica que hay cero elementos, el consumidor queda en un estado de *sleeping* (es decir, queda en pausa hasta que haya al menos un elemento que consumir). De forma análoga, el productor queda en un estado de *sleeping* hasta que haya al menos un espacio de la cola que llenar con elementos.

El modelo originalmente asume que tanto el consumidor como el productor pueden despertarse entre ellos, lo cual se puede pensar de manera similar a si ambos fueran personas. El consumidor despierta al productor cuando ya ha consumido un elemento, para que siga produciendo. El productor hace lo propio con el consumidor cuando ya ha producido un elemento.

El problema con este modelo es el caso en que el consumidor ve que no hay elementos en la cola y entra a estado de *sleeping*. El productor lo va a despertar cuando ponga un elemento, pero si lo despierta antes que él entra a estado *sleeping* (basta que el consumidor se tarde más en cambiar su estado a *sleeping* que lo que se demora el productor en generar un elemento), entonces lo va a despertar mientras aún está despierto. No va a volver a despertarlo, porque nunca más habrá exactamente un elemento en la cola, siempre habrá más. Finalmente, el productor se irá a *sleeping* también, cuando termine de llenar la cola.

Este problema que se acaba de describir, se resuelve mediante el uso de dos ***semáforos***, variables que determinan si se puede o no acceder a una determinada sección del código o ***sección crítica***, en este caso.

Segundo modelo: solución con semáforos

Lo que se hace es una solución más sencilla. Siguen existiendo estos dos threads, pero para que puedan acceder a agregar o quitar elementos de la cola, según sea el caso, dependen de los semáforos.

Un **semáforo** funciona de la siguiente forma: se pueden pensar como enteros que tienen dos operaciones (***wait*** y ***signal***), las cuales permiten decrementar o incrementar su valor, respectivamente. Sin embargo, el decremento solamente se hace hasta el valor cero. Cuando el semáforo llega a cero, los siguientes decrementos son recordados en el orden que llegaron, y hacen que el thread que los esté ejecutando quede esperando en esa línea de código. Si hay *n* threads que están esperando el mismo semáforo en un determinado orden, y llega uno que lo incrementa, el primer thread que quedó esperando (el que lleva más tiempo) podrá decrementarlo a 0 nuevamente y seguir ejecutando código. Dejará a los otros (*n*-1) threads esperando.

¿Cómo aplicar esto al modelo? Antes que un thread tome un elemento de la cola o lo ponga, comprueba que puede hacerlo. Si el semáforo que representa la cantidad de elementos que hay en la cola, está en cero, entonces el productor **antes** de entrar a tomar un elemento, verifica que hayan elementos. Cuando comprueba que no hay, se queda esperando. Cuando el productor llega e incrementa nuevamente al semáforo (después que efectivamente puso un elemento en la cola), el semáforo alcanza un valor positivo y puede ser decrementado, por lo que el consumidor deja de esperar.

En el caso de que la cantidad de elementos que producir sea cero, pasa algo análogo. Es importante notar que los valores de los semáforos siempre tienen un valor que puede ser menor o igual a la cantidad de elementos reales.

¿Por qué esto soluciona el problema anterior? En el primer modelo, se supuso que el consumidor se tardaba lo suficiente (después de chequear que no había elementos en la cola) en irse a dormir como para que el productor lo despertara antes. **El productor despertaba al consumidor estando aún despierto**. ¿Puede esto suceder acá? No se puede, debido a que **el chequeo de cantidades en la cola** y de **ir a dormir** ocurren, en términos prácticos, **al mismo tiempo**. La operación del consumidor de irse a dormir cuando se da cuenta que no hay elementos en la cola (un ***wait*** sobre un valor 0) es **atómica**, es decir, es indivisible, no puede ocurrir nada entre el chequeo y el cambio de estado. Esto le impide al productor ir a despertar en un momento que no debería hacerlo.

Tercer modelo: varios consumidores y productores

Por completitud, el segundo modelo se puede extender a una cantidad cualquiera de productores y de consumidores. Acá surge otro problema. ¿Qué pasa si dos productores van a producir al mismo tiempo? Suponiendo que la implementación de la función de poner un elemento consta de dos pasos:

1. Determinar la posición en que se pondrá el elemento (y ponerlo).

2. Incrementar en uno la posición para que el siguiente productor la use para poner el siguiente elemento.

Entonces puede suceder que dos productores ejecuten el paso 1 antes de que se ejecute el paso 2 por parte de alguno de ellos. Eso hará que ambos escriban el elemento en la misma posición, y por lo tanto, uno sobrescribirá lo que haya escrito el otro. Más aún, como la posición se incrementa dos veces, se está contando una posición que está vacía.

Se puede observar que el origen del problema tiene la misma naturaleza: dos operaciones que deberían ocurrir juntas ocurren separadas. **Otra forma** de hacer que esto no ocurra es no juntando las operaciones, sino impidiendo que, mientras ocurre esta secuencia de dos operaciones, **ningún** otro thread pueda estar ejecutando.

Para esto se usa un **mutex** (exclusión mutua), que funciona de forma similar a un semáforo binario (que sólo toma valores 0 y 1). La diferencia con el semáforo es que éste sólo puede ser manejado por un thread, es decir, que al ser decrementado, sólo podrá ser vuelto a incrementar por el mismo thread.

De esta manera, lo que se hace es que el mutex se bloquea (decrementa) justo antes de modificar la cola, y se incrementa justo después, por lo que esa operación queda protegida de concurrencia. Puede pensarse como el acceso a una cabina telefónica: puede haber a lo más una persona adentro haciendo uso de ella, y sólo ella tiene acceso a desocuparla.

Otro problema: el de los lectores y escritores

Este problema consiste básicamente en la idea de que las lecturas y las escrituras en un documento no deberían hacerse al mismo tiempo, porque dan problemas de consistencia.

¿Qué pasa, por ejemplo, suponiendo que se tienen varios threads lectores y varios threads escritores, si un lector lee el documento completo, pero éste ha sido modificado por un escritor mientras se estaba leyendo? Se obtendrá más de un estado del mismo documento, por parte del lector.

Bajo el supuesto que las **operaciones de escritura pueden llevar a inconsistencias**, en este modelo se permite que haya, a lo sumo, accediendo en un instante dado el documento:

1. Un thread escritor, y ningún lector, por lo tanto.
2. Varios threads lectores, y por lo tanto, ningún thread escritor.

De esta manera, la incorporación de un **mutex** que permita acceder el documento de forma exclusiva, permitirá también que lo tome un escritor único, o que lo tome cualquier cantidad positiva de lectores.

Versión 1: versión injusta con los escritores

Entonces, cada vez que un escritor logra acceso al documento, para lograr que esto sea exclusivo, tiene que bloquear el mutex (y desbloquearlo al final de su uso). En cambio, sólo **el primer lector** en entrar a leer se encarga de bloquear el mutex (para esto, se tiene un contador de lectores en curso), permitiendo igualmente el acceso a lectores pero no así a escritores. El **último lector** en salir del documento se encarga de desbloquearlo.

Ahora, el problema es el siguiente. ¿Qué pasa si, habiendo *n* lectores, un escritor quiere pedir el documento? No puede obtenerlo y tiene que esperar, pero si después un lector pidiera el documento, bien podría obtenerlo, porque hay sólo lectores sobre el documento. Esto puede considerarse **injusto**, ya que se le está dando una **mayor prioridad** a un thread lector que a uno escritor.

Versión 2: versión justa

Para evitar que sea injusto, se puede suponer una versión en la cual se hace básicamente lo mismo, pero con la diferencia de que cada uno de los threads tiene acceso exclusivo a **solicitar acceso** al documento.

Hay una sección del código en la que solamente puede haber un thread (a lo sumo) y que es donde se espera a que el documento no esté siendo accedido de manera excluyente. Es decir, usando los casos anteriores:

1. Si hay un thread escritor usando el documento, él tiene acceso exclusivo al documento. Cualquier otro thread no puede acceder, por lo que debe quedar esperándose en esta sección.
2. Si hay un thread lector (o más) usando el documento, tienen acceso exclusivo al documento respecto de escritores. No puede acceder ningún escritor, por lo que un escritor debería quedarse esperando en esta sección.

Pero como en esta sección solamente cabe un thread a la vez, sólo un thread puede solicitar al mismo tiempo. Mientras uno está esperando que se le otorgue el acceso, ningún otro puede hacer solicitud de acceso.

Sólo una vez que el acceso exclusivo al documento haya terminado, el thread que hizo solicitud de acceso y que estaba impidiendo que eventualmente otro thread hiciera solicitud de acceso al documento, puede acceder al documento. Si hay otro thread esperando pedir acceso al documento, ahora puede hacerlo, pero debe esperar al thread que acaba de obtener acceso, si es que fuera necesario.

Entonces, suponiendo el mismo escenario en que hay *n* threads que están leyendo el documento, si la solicitud de acceso del documento está protegida por otro **mutex** (tanto para lectores como para escritores), si llega un escritor primero y asumiendo que no hay solicitantes al documento, va a intentar pedir el documento, entrando a esta sección del código, pero no va a poder obtener acceso, porque ya están accediendo los lectores. Sin embargo, **no va a permitir que otro thread haga una solicitud.** El thread escritor no va a salir de esa sección protegida de solicitud de acceso, hasta que su acceso sea **exitoso**, y sólo entonces va a permitir pedir acceso a otro thread. Esto significa que si había un lector que llegó después, se tiene que quedar esperando también y va a entrar después que el escritor.