Concurrencia

Ticiano Ian Morvan

Octubre 2024

1 Introducción

Entramos en el concepto de **hilos**, donde un mismo programa tiene más de un punto de ejecución. Puede verse a cada hilo como un proceso separado, pero que comparten el mismo espacio de direccionamiento y por lo tanto acceden a la misma información. Para lograr su ejecución, guardamos y traemos el estado de los hilos desde los **bloques de control de hilos** (**TCBs**, del inglés *thread control blocks*), permitiendo persistir sus registros y, a través un cambio de contexto, recuperar su estado anterior para seguir ejecutándolos. Es importante notar que este cambio de contexto **no altera el espacio de redirección**.

Un hilo, a pesar de compartir el espacio de redirección con los demás, mantiene su propia pila, que ocupa una porción de la memoria del proceso. Si bien las pilas se colocan una encima de otra, normalmente no se superponen.

Es importante pensar también el *¿por qué utilizamos hilos?*, dando lugar a dos principales casos de uso: **paralelización** y **superposición**. Cuando corremos un programa con más de un hilo a la vez, podemos acelerar su ejecución, haciendo que cada uno de ellos calcule una porción del resultado final, logrando **paralelización**. Por otro lado, a veces necesitamos de atender otras actividades mientras se ejecutan distintas operaciones de I/O, por lo que utilizar más de un hilo puede permitir **superposición** de estas responsabilidades, logrando que el programa no se bloquee y mejorar su eficiencia.

El problema que puede surgir al trabajar con hilos, es el orden en que estos se ejecuten. Pues si bien podemos esperar a que terminen de ejecutarse (a través de `Pthread_join()`), no podemos incidir directamente en el orden que toma el planificador. Éste último debería implementar correctamente un algoritmo que permita la ejecución lógica del programa, pero no siempre es así. Tomemos por ejemplo el código del Listing 1.

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *new_thread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  int rc;
  printf("main: begin\n");
  Pthread_create(&p1, NULL, new_thread, "A");
  Pthread_create(&p2, NULL, new_thread, "B");
  // join espera por la finalización de todos los hilos
  Pthread_join(p1, NULL);
  Pthread_join(p2, NULL);
  printf("main: end\n");
  return 0;
}
```

Listing 1: Creación de hilos en C

Este programa podría, de forma lógica, imprimir "A" y luego "B", pero también podría imprimir "B" y luego "A" y ambas son ejecuciones válidas del mismo. Esto puede llevar a contradicciones y errores inesperados, como cuando queremos acceder a una variable compartida entre hilos.

La parte de un programa donde varios hilos acceden a un recurso compartido, que no debería ser ejecutada concurrentemente por más de un hilo, se llama **sección crítica**. En esta parte, podemos sufrir de **condiciones de carrera** que pueden terminar en resultados erróneos, siendo **indeterminado**, contrario a la noción general de que debería ser un cómputo **determinista**. Lo que buscamos para este tipo de casos, es **exclusión mutua** que garantice que solo un hilo ejecute la sección crítica del programa.

Por último, la **atomicidad** garantiza un estado "binario" de una instrucción: o se ejecuta completamente, o directamente no lo hace. Esta atomicidad se puede obtener al recibir ayuda del *hardware*, que provee instrucciones sobre las que podemos construir un conjunto de **primitivos de sincronización**. Estos primitivos permiten escribir código multi-hilo que acceda a secciones críticas de manera sincronizada y controlada, obteniendo resultados correctos a pesar de la naturaleza desafiante de la ejecución concurrente.

2 Cerraduras

Para combatir el problema que mencionamos anteriormente, utilizaremos **cerraduras** (o en inglés *locks*) que nos permitirán lograr la **exclusión mutua** necesaria para una correcta sincronización de nuestros hilos, evitando condiciones de carrera y acercándonos más a un resultado determinado.

La librería POSIX nos provee para esto de distintas ayudas interesantes. Podemos declarar cerraduras con el tipo `pthread_mutex_t`, inicializarlas con `PTHREAD_MUTEX_INITIALIZER` y, en adelante, bloquearlas o desbloquearlas con las funciones `Pthread_mutex_lock()` y `Pthread_mutex_unlock()`. Un ejemplo simple de esto puede verse a continuación

```c
// Declaramos globalmente una cerradura
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
...
// Por ejemplo, si estuviesemos iterando sobre un arreglo
// al que le asignamos valores almacenados en los registros
Pthread_mutex_lock(&lock);
a[i] = value_in_register;
Pthread_mutex_unlock(&lock);
```

Puede verse como pasamos la variable `lock` a las rutinas, esto permite concurrencia de grado fino, pues podemos tener más de una cerradura al mismo tiempo.

Si quisiésemos construir nuestras propias cerraduras, deberíamos primero cumplir tres pincipios necesarios: proveer **exclusión mutua**, que sea **justo** (que todos los hilos sean capaces de obtener la cerradura) y que sea **competente**, es decir, que sea eficiente. Para ello, se han diseñado con los años distintos primitivos y estrategias que, algunas mejores que otras, permiten la concurrencia.

2.1 Controlando las interrupciones

En sistemas con un solo procesador, deshabilitar las interrupciones es el camino más sencillo, puesto que al no poder impedir que un hilo pare de ejecutarse, nos aseguramos de que termine. Esto, si bien funciona, trae consigo muchos problemas. En particular, podemos tener hilos que monopolicen la CPU, ya que al deshabilitar las interrupciones debemos *confiar* en ellos para ejecutar operaciones privilegiadas; en sistemas con más de un procesador esta opción provoca que un hilo pueda ejecutarse en otros y acceder así a la sección crítica o incluso podrían darse casos de indeterminación en el funcionamiento del sistema, ya que muchas interrupciones avisan de cambios de estado, por ejemplo cuando termina una operación I/O.

2.2 Usar solo *loads* y *stores*

Si definimos una `flag` que valga 1 si la cerradura está bloqueada y 0 si no lo está, podríamos lograr que el hilo que tome control de la cerradura se ejecute normalmente, mientras que los otros que esperan por su turno caen en un bucle de espera o *spin-waiting*. El problema de esta estrategia es, por un lado, que más de un hilo podría tomar el control de la cerradura y así, permitir el acceso de varios hilos a una sección crítica. Por otro lado, dejar que un hilo espere en un bucle no es para nada eficiente.

2.3 Test-and-set

Las instrucciones ***test-and-set*** se encuentran en varias arquitecturas modernas y permiten devolver el valor original de una variable a la vez que se le asigna uno nuevo. Para nuestras cerraduras, esto nos permite la exclusión en dos casos. Si la cerradura no estaba bloqueada, el valor de `flag` será 0 y el hilo que la solicite lo hará; cuando termine de ejecutarse, lo liberará de nuevo. Si la cerradura ya estaba bloqueada, entonces el nuevo hilo que lo solicite esperará hasta que `flag` sea 0, y cuando lo sea, se ejecutará y le asignará 1, desbloqueándolo luego.

Otras formas de construir *spin locks* son con instrucciones *compare-and-swap*, *fetch-and-add* y *load-linked* con *store-conditional*.

2.4 ¿Cómo mejoramos el rendimiento?

Mientras que un hilo ejecuta la sección crítica, podemos tener muchos que esperan en un bucle, perdiendo incontables unidades de tiempo de CPU. Para evitar esto, podríamos simplemente hacer que los hilos que se encuentran esperando la cerradura, liberen la CPU y permitan a otros ejecutarse. Con pocos hilos esto funciona de maravilla, pero a medida que aumentamos la cantidad empezamos a ver muchos cambios de contexto que, a la larga, son muy costosos.

Si en vez de esperar en un bucle infinito, ponemos estos hilos a "dormir" y los organizamos en una cola que lleve registro de ellos, podemos evitar el problema de *starvation* y manejar mejor los recursos disponibles, ya que solo despertamos al hilo indicado cuando la cerradura está libre. Sobre esta estrategia distintos sistemas han diseñado alternativas particulares, como lo son `park()`, `unpark()` y `setpark()` de Solaris, o **futex** en Linux. En particular, la solución de Linux es **híbrida**, puesto que combina el bucle de espera (por un corto periodo de tiempo) y, en caso de no haber podido obtener la cerradura, pone el hilo a dormir.

3 Estructuras de datos basadas en cerraduras

Ahora que sabemos como funcionan y se implementan estas cerraduras, *¿cómo las utilizamos?*

3.1 Contadores

La estructura de datos más simple a la que podemos aplicar una cerradura es un contador. Normalmente, tendremos un código similar a este

```
1   typedef struct __counter_t {
2     int value;
3     pthread_mutex_t lock;
4   }
5
6   // Inicializamos el contador y la cerradura
7   void init (counter_t *c) {
8     c->value = 0;
9     pthread_mutex_init (&c->lock, NULL);
10  }
11  // Tomamos el control e incrementamos el contador
12  void increment (counter_t *c) {
13    pthread_mutex_lock (&c->lock);
14    c->value++;
15    pthread_mutex_unlock (&c->lock);
16  }
17
18  // Tomamos el control y decrementamos el contador
19  void decrement (counter_t *c) {
20    pthread_mutex_lock (&c->lock);
21    c->value--;
22    pthread_mutex_unlock (&c->lock);
23  }
24
25  // Obtenemos el valor del contador
26  void get (counter_t *c) {
27    pthread_mutex_lock (&c->lock);
28    int rc = c->value;
29    pthread_mutex_unlock (&c->lock);
30    return rc;
31  }
```

Listing 2: Implementación de un contador básico

Lo cual es muy simple (¡y funcional!) pero en términos de concurrencia, llega a perder rendimiento de forma exponencial cuando más de un hilo opera sobre esta estructura. Buscaremos entonces lograr **escalado perfecto**[1]

3.2 Conteo escalable

Para mejorar las operaciones concurrentes trabajando con contadores, se crearon los **contadores aproximados**. Estos son una especie de modularización de los contadores tradicionales donde, en vez de tener un solo contador físico, tendremos uno por cada núcleo del procesador involucrado en el programa, con un contador lógico global. Además, cada uno de los contadores físicos tiene asignada una cerradura local, como también se designa una cerradura para el contador global. El funcionamiento es sencillo, usaremos una variable S que nos servirá de límite. Cada núcleo actualizará solamente su contador y, cuando su valor llegue al límite S, actualizaremos el contador global. Luego, reiniciamos el contador local a cero. Esto logra que distintos hilos trabajen independientemente sobre sus contadores y, en cierto punto, se comuniquen con el contador global para actualizar su valor y seguir operando.

El rendimiento de esta estrategia dependerá del valor asignado a S, puesto que a un valor menor le corresponde un tiempo de actualización más corto y por lo tanto mayor cercanía al valor real, pero también implica una cantidad mayor de comunicaciones con el estado global y en consecuencia un peor rendimiento. Por otro lado, mientras más agrandamos el valor de S mejor es el rendimiento que podemos obtener, pero un valor menos actualizado.

[1] Diremos que hay escalado perfecto cuando múltiples hilos en múltiples procesadores pueden operar en el mismo tiempo que un solo hilo en un procesador único.

3.3 Otras estructuras concurrentes

Además de contadores, podemos implementar cerraduras en **listas enlazadas**, **colas** e incluso **tablas *hash***. En todas ellas la idea es la misma: asegurar la exclusión mutua cuando se quiere **acceder** o **cambiar** un valor. Así, tomaremos la cerradura cuando queramos insertar un nuevo elemento en una lista enlazada, desencolar el primer elemento de una cola o buscar un elemento en una tabla. Son posibles ciertas optimizaciones, como tomar pequeñas cerraduras entre nodos de una lista (*hand-over-hand locking* o *lock coupling*) o tomar dos cerraduras en una cola: una al principio y otra al final. Algo a notar, es que estas optimizaciones **no necesariamente** mejoran el rendimiento, puesto que pueden agregar complejidad innecesaria o aumentar las probabilidades de fallos que en estructuras más sencillas no aparecen o son más fáciles de solucionar.

En general, habilitar la concurrencia no siempre mejora el rendimiento. Se debe procurar evitar la optimización prematura y solo arreglar dichos problemas cuando existan.

4 Variables de condición

Si creamos un hilo hijo y queremos asegurarnos de que una condición se cumpla antes de continuar, ¿qué debemos hacer?. Podríamos utilizar una variable compartida, pero en la implementación veríamos que nuestro hilo principal se quedaría atascado esperando a que la condición se verifique, desperidiciando recursos valiosos. Por otro lado, podemos utiliza una variable de condición, asociada a dos funciones `wait()` y `signal()` (en POSIX, declaramos un variable de condición con `pthread_cond_t` y tenemos ambos métodos `pthread_cond_wait()` y `pthread_cond_signal()`).

La función `wait()` duerme un hilo, asumiendo que la cerradura que se le añade como argumento está cerrada, por otro lado `signal()` se ejecuta cuando un hilo ha hecho un cambio en el programa y se requiere despertar aquellos procesos que dependieran de la variable de condición asociada. Es responsabilidad de `wait()` el recuperar el control de la cerradura antes de volver al proceso que dio origen a la llamada. Este manejo de las cerraduras permite evitar condiciones de carrera en los momentos en que se duerme un hilo, para minimizar problemas de estancamiento o resultados inesperados. Como buena práctica, se recomienda siempre tomar control de la cerradura cuando se llame a `wait()` o `signal()`, incluso cuando haya casos donde no sea completamente necesario.

4.1 El problema del Proveedor/Consumidor (*Bounded Buffer*)

En esta situación contemplamos dos tipos de hilos: productores y consumidores. Los hilos productores almacenan información en un *buffer* que luego será accedido por los hilos consumidores. Esto, simplemente entendiendo su propósito, nos da a entender que deberemos asegurar la correcta sincronización de ese acceso y escritura para que no tengamos, por ejemplo, dos consumidores que accedan al mismo valor al mismo tiempo. Para una explicación más sencilla, analizaremos el código válido que solventa los dos problemas asociados a esta estrategia: que un consumidor "le gane" el lugar a otro que estaba esperando anteriormente y que, dependiendo de como se estructure la cola de espera, pongamos a dormir un hilo incorrecto.

```
1   // Rutinas get() y put() que modifican el buffer .
2   int buffer [MAX];
3   int fill_ptr = 0;
4   int use_ptr = 0;
5   int count = 0;
6
7   void put (int value) {
8     buffer[fill_ptr] = value;
9     fill_ptr = (fill_ptr + 1) % MAX;
10    count++;
11  }
12
13  int get () {
14    int tmp = buffer[use_ptr];
15    use_ptr = (use_ptr + 1) % MAX;
16    count--;
17    return tmp;
```

```
18    }
19
20    // Hilos productor y consumidor
21    cond_t empty, fill;
22    mutex_t mutex;
23
24    void *producer (void *arg) {
25      int i;
26      for (i = 0; i < loops; i++) {
27        pthread_mutex_lock (&mutex);
28        while (count == MAX)
29          pthread_cond_wait (&empty, &mutex);
30        put (i);
31        pthread_cond_signal (&fill);
32        pthread_mutex_unlock (&mutex);
33      }
34    }
35
36    void *consumer (void *arg) {
37      int i;
38      for (i = 0; i < loops; i++) {
39        pthread_mutex_lock (&mutex);
40        while (count == 0)
41          pthread_cond_wait (&fill, &mutex);
42        int temp = get ();
43        pthread_cond_signal (&empty);
44        pthread_mutex_unlock (&mutex);
45        printf ("%d\n", tmp);
46      }
47    }
```

Listing 3: Implementación correcta de la solución al problema Proveedor/Consumidor

Un punto importante de este código es siempre usar `while` en vez de `if`. Esto permite, en un programa multihilo, evitar que más de un hilo se despierte a través de una sola señal, por lo que se considera recomendado utilizar `while` siempre que tengamos este tipo de concurrencia. Por otro lado, tenemos dos variables de condición, `fill` y `empty` que sirven para principios distintos. Esta separación evita que se despierte un hilo del tipo equivocado al cumplirse la condición del bucle. Las versiones primitivas (y erróneas) de la solución a este problema se encuentran detalladas en el libro y no serán cubiertas aquí.

5 Semáforos

Un semáforo combina cerraduras y variables de condición, inicializando un valor que determinará su comportamiento. En el estándar POSIX, podemos utilizar las rutinas `sem_wait()` y `sem_post()` para esperar (o tomar el control) de la ejecución si el valor del semáforo lo permite y para incrementar el valor de éste, respectivamente. Normalmente, cuando el valor del semáforo es uno o más, la rutina `sem_wait()` simplemente vuelve, si es menor o igual a cero, suspenderá la ejecución hasta la próxima llamada a `sem_post()`. Una invariante notable de este proceso, es que cuando un semáforo tiene un valor negativo, ese valor indica la cantidad de hilos esperando por él (puesto que cada llamada a `sem_wait()` lo decrementa).

5.1 Semáforos binarios

Un semáforo binario es aquel que toma como mayor valor al 1, por lo que cualquier llamada a `sem_wait()` provocará que los demás hilos que quieran acceder a él se suspendan, ya que el valor del semáforo será cero. Al ser de tan sencilla implementación y uso, se lo puede usar como cerradura. Si por ejemplo, tenemos dos hilos T_0 y T_1 y un semáforo $S = 1$, proponiendo que T_0 llame primero a `sem_wait()`, entonces $S = 0$ y por lo tanto, T_1 llamará también a `sem_wait()` y se suspenderá, pues $S = -1$. Una vez

que T_0 llame a `sem_post()`, valdrá $S = 0$ y T_1 se despertará y podrá ejecutar la sección crítica, llamando finalmente a `sem_post()` tal que $S = 1$.

5.2 Semáforos para ordenamiento

En un programa concurrente, podemos utilizar semáforos para ordenar la ejecución de los hilos participantes. Si, por ejemplo, utilizamos `sem_wait()` dentro de un proceso padre después de crear un proceso hijo, llamando a `sem_post()` cuando éste termine de ejecutar, entonces podremos lograr que el proceso padre *solo ejecute* una vez la condición que espere haya satisfecho el proceso hijo sea inequívocamente correcta.

Para inicializar un semáforo correctamente, podemos seguir la pista de Perry Kivolowitz y considerar el número de recursos que queremos dar inmediatamente después de la inicialización. Para una cerradura, el valor 1 indica que que queremos tomar el control de ésta inmediatamente después de la inicialización. Para ordenar, el valor 0 inidca que no hay nada que ceder al principio.

5.3 Otras aplicaciones

Podemos utilizar semáforos en problemas ya vistos como el **Proveedor/Consumidor**, para el cual tendremos que evitar el *deadlock* (cuando dos o más hilos que son dependientes entre sí esperan indefinidamente a que el otro termine) y utilizar correctamente la exclusión mutua, o problemas nuevos que, a pesar de que no siempre llegan a las soluciones más eficientes, han surgido para pensar acerca de concurrencia, como las **Cerraduras Lector-Escritor**, ***Dinning Philosophers's problem***, ***Thread Throttling***, entre otros. Implementar semáforos usando primitivos de bajo nivel es bastante sencillo usando exclusión mutua y variables de condición, pero a veces generalizar la función de un semáforo a la de una cerradura *puede no* ser correcta.