

## Parte 2: Conciencia

### 26. Conciencia: una introducción

Agregaremos una abstracción más al concepto de proceso: los hilos, éstos se pueden ver de varias formas como un programa que tiene varios puntos de ejecución (PC) o como procesos separados que comparten el mismo espacio de direcciones.

El estado de un hilo es muy similar al de un proceso, cada hilo tiene su conjunto privado de registros, por lo que si hay más de un hilo ejecutándose en la misma CPU, se debe hacer un cambio de contexto, en el que se almacenan los registros del hilo en una estructura llamada **thread control block (TCB)**. La diferencia con los cambios entre procesos es que se conserva el mismo espacio de direcciones.

Cada hilo almacena sus propios registros y valores en su propio stack, llamado **thread local storage**, dividiendo el stack del proceso. Por lo general los stacks son pequeños, excepto cuando se hace mucha revisión.



### 26.1 ¿Por qué usar hilos?

Por un lado, **parallelizar** nuestros programas puede significar un gran aumento del desempeño, aprovechando al máximo los recursos hardware como múltiples CPUs.

Por otro, un mismo proceso puede hacer algo parecido al **multiprogramming**.

### 26.2 Creación de un hilo

Los hilos se pueden crear con la función

**pthread\_create**, pero no necesariamente se ejecutan en el orden de creación: el **schedule** es el que se encarga de determinarlo, y es muy difícil saber quién será ejecutado.

### 26.3 Datos compartidos

Se presenta un problema cuando dos hilos intentan actualizar la misma variable, como un contador global. No se presenta un resultado determinista. Es decir, a veces da un resultado y a veces no.

### 26.4 Scheduling descontrolado

El error de 26.3 es el resultado de una serie de eventos desafortunados, por ejemplo, un hilo carga el valor del contador y le suma 1, y ante de actualizar el valor, debe cambiar el proceso, que carga el mismo valor y vuelve a sumarle uno, dando el mismo resultado, para luego ambos sobreescibir el mismo valor.

Esto se conoce como **condición de carrera** (y para este caso específico, **carrera de datos**) los resultados dependen de los tiempos de ejecución del código, llevando a un programa a tener un resultado **indeterminado**.

Cuando varios hilos pueden acceder a variables compartidas, lo llamamos **una sección crítica**, y para prevenirla lo que buscamos es una **exclusión mutua**.

### 26.5 El deseo de atomicidad

Una forma de evitar estados intermedios es con **instrucciones atómicas** que hagan todo en un paso, o no hagan

Pero hace instrucciones atómicas para todo puede representar un seto gigante para el hardware. En vez de eso, lo que constituyes es un conjunto de primitivos de sincronización, que al usar este soporte de HW combinado con el sistema operativo, permite construir código multihilo que accede a las secciones críticas de manera sincronizada.

## 26.6 Esperar a otro hilo

Además del problema de acceso a datos, también está el problema de que un hilo debe esperar a otro, por ejemplo cuando un hilo debe procesar los datos que está obteniendo un hilo en una tarea de I/O.

## 27. Thread API

Para crear un hilo, llamamos al método `pthread-create`, al que se le pasan varios parámetros:

- Una dirección de una estructura `pthread`
- Atributos o inicializar, `NULL` para inicializarlos por defecto
- Un apuntador a una función
- Un conjunto de argumentos

```
#include <stro.h>
#include <pthread.h>
```

```
typedef struct {
    int a, b;
} myarg_t;
```

```
void *mythread (void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf ("%d", args->a);
    return NULL;
}
```

```
int main (int argc, char * args[]) {
    pthread_t p;
    myarg_t args = {10, 20};
    int rc = pthread_create(&p, NULL,
                           mythread, &args);
    ...
}
```

Cabe resaltar que cuando se recibe un parámetro de tipo `void`, hay que hacerle el cast respectivo.

Para completar la ejecución de un hilo se hace uso de la rutina `join`, ésta recibe dos argumentos

- Un hilo, de tipo `pthread`
- Un apuntador para retomar el valor

Se debe pasar un apuntador y no el valor ya que el `join` modifica el valor del argumento que se le pasa.

```
typedef struct { int x, y;} myret_t;
void *mythread ( void *arg) {
    myret_t *rvals = Malloc(sizeof(myret_t));
    rvals->x = 1;
    rvals->y = 2;
    return (void *) rvals;
}
```

```
int main() {
    pthread_t p;
    myret_t *rvals;
    myarg_t args = {10, 20};
    pthread_create(&p, NULL, mythread, &args);
    pthread_join (p, (void **) &rvals);
    printf ("returned %d", rvals->x);
    free (rvals);
    return 0;
}
```

Para bloquear los hilos en secciones críticas, tenemos los locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex)  
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Esto hace que si un hilo tiene el lock, otros hilos no lo puedan hacer hasta que el que lo posea, lo suelte. Hay que tener mucho cuidado con la inicialización de los locks.

Estas rutinas pueden fallar, por lo que se recomienda crear wrappers que verifiquen si se ejecutó correctamente.

Otra cosa muy útil son las variables de condición, que sirven cuando se debe hacer un tipo de señalización entre hilos. Los dos rutinas principales son

```
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)  
pthread_cond_signal(pthread_cond_t *cond),
```

Wait pone al hilo que lo invocó como dormido y espera que otra señal lo despierte

Intenta:

- Mantener el código simple
- Minimizar la interacción entre hilos
- Inicializar locks y variables de condición
- Ser cuidadoso al poser y liberar valores a los hilos
- Cada hilo tiene su propio stack, por lo que la información que se quiera compartir debería estar en el heap del proceso
- Usar variables de condición para enviar señales entre procesos.
- Usar los manuales y documentación

## 28. Locks

Los locks se ponen alrededor de secciones críticas para que se ejecute como si fuera una sola instrucción atómica

### 28.1 Locks: La idea básica

Un lock es una variable de un tipo, la cual contiene el estado del lock en un momento dado, el cual puede ser **locked** o **unlocked** para asegurarse que solamente un hilo está en la sección crítica.

Un hilo llama a la función **lock()** y si el sección crítica no está siendo ejecutado por ningún otro hilo entonces entra en la sección crítica, el lock pasa a estado **locked** y el hilo pasa a ser el **owner** del lock. Otros hilos pueden intentar pedir el lock, pero si este no ha sido liberado por el owner, no podrán ejecutar nada. Esto le devuelve un poco de control de la ejecución del programa al programador.

### 28.2 Pthread locks

POSIX llama a los locks como **mutex**, que viene de **mutual exclusion**. Estos locks se hacen a través de variables, por lo que uno puede bloquear distintas secciones con distintos locks, en vez de tener un super lock que bloquee todo.

### 28.3 Construyendo un Lock

Para poder construir un lock funcional, se deben usar funcionalidades del Hardware

## 28.4 Evaluando Locks

Para construir y evaluar un lock se definen los siguientes criterios:

- Exclusión mutua: ¿el lock impide que múltiples hilos accedan a una sección crítica?
- Justicia (fairness): ¿Todos los <sup>hilos</sup> tienen oportunidad de obtener un lock cuando éste es disponible?
- Rendimiento: ¿Cuánto es la penalización al tomar o soltar un lock? ¿Cómo va en múltiples hilos en una CPU? ¿Y en múltiples hilos en múltiples CPUs?

## 28.5 Controlando Interrupciones

Una aproximación en procesadores de una sola CPU era deshabilitar las interrupciones en la sección crítica, pero esto tiene muchos inconvenientes, como

- Pasar a modo privilegiado y controlar el proceso
- Que el hilo monopolice la CPU
- No funciona en multiprocesadores
- No da cuenta de otros eventos, como finalizar una lectura de disco
- Ineficiencia

Este sólo es útil en ambientes muy controlados, como el OS accediendo a sus propias estructuras de datos.

## 28.6 Un intento fallido

Otra aproximación insuficiente es poner una bandera en 1 para indicar que un hilo se encuentra en una sección crítica, sin embargo esto tiene varios inconvenientes, por un lado se puede dar la situación de que ambos hilos pongan la variable bandera en 1,

permitiéndoles ejecutar a ambos la sección crítica, y por otro lado, un hilo que no tenga el lock tendría que estar en un ciclo verificando constantemente si el valor de la bandera cambió o no, esto se conoce como **Spin-waiting** y es desperdiciar tiempo de CPU.

## 28.7 Construyendo locks con Test-And-Set

Una aproximación funcional consiste en tener una bandera, consultarla y retocarla en la misma instrucción, esto se logra con el método **Test-And-Set**, que actualiza el valor del lock y retorna el viejo, así, otros hilos se quedan en un spin hasta que se libere el lock.

Para que esta aproximación funcione, se necesita un **preemptive scheduler**, es decir, uno que cada tanto genere una interrupción.

## 28.8 Evaluando los spin locks

A pesar de que se asegura la correctitud, el spin lock no garantiza fairness: un hilo podría quedar ejecutándose para siempre y su rendimiento es malo: puede darse el caso de que  $N-1$  hilos se quedan en el spin.

## 28.9 Compare and Swap

La idea es comparar a un puntero con un valor esperado. Si ambos coinciden, entonces actualiza al puntero. Si no, nada. En ambos casos retorna el valor viejo del puntero.

### 28.10 Load-Linked y Store-Conditional

Esta aproximación utiliza dos instrucciones, load-linked se encarga de cargar una dirección de memoria. El store-conditional lo que hace es que sólo funciona si no se ha realizado un store en la dirección especificada, si falla, retorna 0, si no, retorna 1.

Usándolas en conjunto se puede crear un mecanismo de bloqueo, primero con load-linked se verifica que la variable bandera esté en cero, es decir, disponible, luego, con el store-conditional se escribe en el lock, bloqueándolo. Si esta última funciona, todo, sino, debemos empezar de nuevo.

### 28.11 Fetch and Add

Consiste en formar como un mecanismo de turnos, llamando una posición de memoria y sumándole 1, esto se convierte en el mi-turno, que se compara con un turno global, y se desbloquea al aumentar el turno global, esto garantiza que todos los hilos tengan tiempo de ejecución, cosa que no se lograba con las aproximaciones anteriores.

### 28.12 Demasiado spinning

Algunas aproximaciones anteriores pueden llegar a ser muy inefficientes, haciendo que se gaste mucho tiempo de CPU, para solventar esto, puede ser necesario el soporte del OS.

### 28.13 Just yield

Una forma de mitigar el spinning es que si el lock está bloqueado, en vez de quedarse en spin, se cede

la CPU haciendo un llamado a `yield()` que se encargará de hacer un llamado al sistema que cambie el hilo que se está ejecutando, pasando al que lo llamó al estado ready. Esto a pesar de que mejora lo del tiempo de CPU, igual trae el costo del cambio de contexto y sigue existiendo el problema de la starvation.

### 28.14 Usar colas: sleep en vez de spin

En vez de dejar la ejecución de los hilos a la suerte, definir un mecanismo de control explícito, usando colas para saber qué hilos esperan el lock.

Una aproximación puede ser acudir a Test And Set, y si falla, se encola al hilo actual, usando un guard. Esto no quita por completo la inactividad, pero reduce considerablemente el tiempo de spin.

El lock se pasa directamente de un hilo a otro, cuando se despierta el receptor. Sólo se pone en cero cuando no hay hilos encolados.

Se puede presentar una situación de competición al intentar parquear un hilo, conocida como **wakeup/dating race** que puede llevar a que se ejecute un cambio de contexto que haga que un hilo despierte para siempre.

### 28.16 Locks de dos fases

La aproximación que usa Linux es parecida a una muy vieja a la que hoy se refiere como lock de dos fases. En la primera fase se spins un poco, con la esperanza de obtener

el lock, si no lo consigue se entra en una segunda fase en la que se duerme al "llamado" y sólo se despierta cuando el lock se libera. Es un tipo de aproximación híbrida.

## 29. Lock-Based concurrent Data Structures

Para que una estructura sea segura y lo puedan usar varios hilos de manera concurrente se deben aplicar locks en ella.

### 29.1 Concurrent counters

Una estructura muy simple es el contador, que guarda y actualiza su valor con métodos que se pueden sincronizar aplicando un simple lock. Esto puede ser útil, pero puede conducir a un bajo rendimiento cuando hay muchos hilos. Lo que se busca entonces es un **escalamiento perfecto**, de tal manera que no se afecte el rendimiento de la estructura al ejecutar más hilos.

#### Contadores aproximados

Esta aproximación consiste en que cada CPU tiene un contador local que sincroniza con su propio lock, y hay un contador global que se actualiza cada **5 unidades** de tiempo, reiniciando los contadores locales.

Mientras mas grande sea S, más escalable es la estructura, pero menos actualizado va a estar el valor del contador.

### 29.2 Concurrent linked lists

Se examina una lista bastante sencilla, sólo el caso de inserción. En ella se aplica un lock en las rotinas de inserción y de búsqueda. En la inserción se pude dar el caso de que el llamado

a malloc falle. Las secciones críticas que deben ser cuidadosas con el lock son la actualización de la lista (cuando se inserta) y el ciclo con el que se recorre la lista, (cuando se busca). Esto evita que se introduzcan bugs.

#### Escalando listas ligadas

Una aproximación que conceptualmente tiene sentido es, en vez de bloquear toda la cola, mejor sólo bloquear un nodo. Esto a pesar de que habilita un alto nivel de concurrencia, no logra mejorar la velocidad de la estructura.

### 29.3 Concurrent Queues

En esta estructura se aplican los locks tanto en la cabeza como en la cola de la lista, bloqueando los últimos de encolados y desencolados. Se ayuda de un head temporal. Sin embargo una cola con sólo locks no cumple con todas las necesidades que tienen las aplicaciones multi-hilo, como casos en los que la cola esté vacía o casi llena.

### 29.4 Concurrent Hash Table

Se puede construir un Hash Table usando las estructuras vistas anteriormente, así, cada bucket de la tabla es una lista ligada. Esto le permite soportar muy bien la concurrencia ya que no se bloquea toda la estructura, sino cada uno de los buckets. Esta estructura escala muy bien comparado con la lista normal.

## 30 Variables de condición

Para escribir programas concurrentes no sólo necesitamos locks, también necesitamos mecanismos para comunicarlos, por ejemplo cuando un hilo padre quiere saber si un hilo hijo terminó o no.

Usar una variable que se pasa como argumento es ineficiente, principalmente en los spins, por lo que se debe buscar una alternativa.

### 30.1 Definiciones y rutinas

Para esperar que una condición sea verdadera, un hilo puede usar algo que se llama **variable de condición**, que es una cola en la que los hilos se pueden meter cuando un estado de ejecución (como una condición) no es la deseada; luego, otro hilo puede despertarlos cuando cambia el estado.

Para ello se debe declarar la estructura **p\_thread\_cond\_t c**; a la que se le asocian dos rutinas **wait()** que dormir a un hilo y **signal()**, que despierta a un hilo que esté esperando esa condición.

**Wait** recibe un lock, y asume que va a estar bloqueado cuando sea invocado, así se encarga de liberar el lock y poner a dormir al hilo invocado. Cuando se despierta el hilo, busca nuevamente el lock, lo que introduce cierta condición de carrera cuando un hilo se intenta dormir a sí mismo.

Pueden ocurrir dos casos que se dice el hilo y el padre siga corriendo, llama al **join**, coge el lock, verifica si el hijo ya terminó, y si no, se pone a dormir. El otro caso es que no siga el padre sino el hijo, modifique

la bandera de "listo" y despierte al padre. Puede ocurrir un error: que el hijo llame a **thr\_exit** inmediatamente sin que haya un hilo en la condición, haciendo que cuando el padre llame a **wait**, se quede esperando para siempre. Por eso se empieza a hacer relevante el uso del "done".

Tip: por simplicidad es mejor mantener el lock

## 30.2 The producer/consumer (Bounded Buffer) problem

Vamos a abordar el problema del productor/consumidor. Un productor se encarga de producir unos datos y un receptor/consumidor los recibe de alguna manera. Por ejemplo

### ls | grep algo

Se produce una salida que le llega a grep a través de un pipe de UNIX en lo que sería el **buffer compartido ligado**. Al ser el buffer un recurso compartido, debe ser sincronizado.

Una aproximación mala:

Primero definimos un tipo de buffer sencillo que almacene un único valor entero. En él se escribe a través de la función **put** y se saca con la función **get** y un conjunto de operaciones que permitan acceder al buffer.

Para implementarlo no basta con un lock, hay que añadirle una variable de condición que sólo les permita producir o consumir cuando el buffer esté vacío o esté lleno, respectivamente.

Se puede presentar un problema. Puede que hayan dos hilos consumidores,

y que uno se disponga a consumir el buffer, pero luego otro se le entromete y vacíe el buffer primero. Luego el primer consumidor llamará a `get()`, pero el buffer estará vacío. La señalización que se hace sigue sólo como una pista, pero no garantiza que cuando el hilo que se despertó corra, el estado del buffer permanezca como se deseaba. La interpretación del significado de la señal se conoce como **Mesa semantics**. Su contraste se conoce como **Hole semantics**, que es más difícil de construir, pero da mejor garantía de que el hilo despertado corra inmediatamente.

Mejor, pero sigue malo

La modificación que se sigue, teniendo en cuenta las Mesa Semantics, es usar un `while` para chequear el estado. Siempre es mejor usar `while`s que `ifs`, por pura seguridad. Sin embargo, aca hay un bug, puede darse el caso de que los tres hilos (dos consumidores y un productor) se queden todos dormiendo a pesar de que en un punto (por ejemplo si el buffer está vacío, debería despertarse el productor) se debió despertar algún hilo.

Esto lleva a concluir que la señalización es necesaria, pero debe ser más dirigida. Un consumidor no debe despertar a otros consumidores, sólo a productores, y vice versa.

The single buffer producer/consumer solution

La solución es usar dos variables de condición, cambiando un poco el comportamiento de los hilos

wait	signal
producer	on empty fill
consumer	on fill empty

## El producer/consumer correcto.

Para generalizar un poco lo anterior, se cambian los valores individuales por colas y se adecúa la lógica para que los productores produzcan si hay algún espacio en los buffers y así.

Esto es más eficiente con un solo productor y consumidor, pues reduce los cambios de contexto.

## 30.3 Covering conditions

Son como medidas que se toman para cubrir ciertos casos. Por ejemplo, puede que haya un espacio de memoria libre para un hilo, pero puede que este esté dormido, por lo que no se ejecuta, y puede que haya otro despertado pero necesite más memoria, entonces se duerme. Para evitar eso, simplemente se despiertan todos los hilos y aí.

## 31. Semáforos

Dijkstra y su equipo introdujeron el concepto de semáforo como una primitiva de sincronización que sirve tanto como un lock y como variable de condición.

### 31.1 Una definición

Un semáforo es una estructura con un valor entero que puede ser modificado por las rutinas `sem-wait` y `sem-post`. El valor inicial del semáforo determina su comportamiento.

Un semáforo puede ser compartido entre hilos y entre procesos. Por ahora no limitamos a uno entre hilos, que se declararía así

```
sem_t s;  
sem_init(&s, 0, 1);
```

Después de declarado, se pueden usar las funciones, que hacen lo siguiente

```
sem_wait() {  
    decrementa el valor del semáforo en uno  
  
    espera si el valor es negativo  
}
```

```
sem_post() {  
    incrementa el valor del semáforo en uno  
  
    si hay uno o más hilos esperando,  
    despierta uno  
}
```

### 31.2 Semáforos binarios

Inicializando un semáforo con un valor de 1 podemos obtener el comportamiento de un lock. Cuando un hilo adquiere el lock, su baja a cero. Si otro hilo intenta adquirir el lock, baja a -1, por lo que se va a dormir. Cuando el hilo con el lock finaliza, aumenta su a cero y despierta al hilo siguiente.

Cabe destacar que cuando  $s$  es negativo, ( $|s|$  es el número de hilos esperando).

### 31.3 Semáforos para ordenamiento

Se puede simular el comportamiento de una variable de condición para que por ejemplo un hilo padre espere a hilo hijo,

para ello debe inicializarse únicamente con el valor de 0, así si el hilo padre se ejecuta primero, el semáforo lo duerme, y si se ejecuta primero el hijo, como no hay nada más en la cola, se ejecuta primero el hijo.

¿Qué pasa si la cola no está vacía?

### 31.4 El problema del productor/consumidor

#### Un primer intento

Si tenemos un buffer con una capacidad de  $I$ , podemos añadir dos semáforos, uno para lleno y otro vacío, inicializados en 0 y MAX respectivamente. Con esto lograremos una implementación correcta, pero si el buffer tiene un tamaño mayor, cuando los productores intenten escribir en el buffer, se presenta una condición de carrera.

#### Añadiendo exclusión mutuo

Una solución consiste en agregar un semáforo con mutex para las secciones críticas del buffer, put y get. Sin embargo, se puede presentar una situación de deadlock.

#### Evitando el deadlock

Se puede presentar que el consumidor adquiere el mutex, llama a sem\_wait pero como está vacío el buffer, se va a dormir con el lock. Luego, el productor intenta producir, y podría despertar al consumidor, pero como no tiene el mutex, también se queda esperando.

### Una solución funcional

Basta con mover el mutex, dejando abierto el semáforo de la sección crítica, así se logra implementar un bounded buffer funcional.

sem-wait (& full)

sem-unlock (& mutex)

int tmp = get()

sem-post (& mutex)

sem-post (& empty)

### 31.5 Reader-Writer locks

Hay problemas que se presentan en las estructuras de datos, por ejemplo cuando hay que leer e insertar en una lista. Cuando se lee, no se debería modificar la lista, y deberían haber varios lectores. Esto se logra introduciendo dos locks nuevos: lock de lectura y lock de escritura.

El lock de escritura actualiza el semáforo impidiendo que otros hilos lo actualicen. El lock de lectura también bloquea la escritura mientras hayan uno o más hilos leyendo. Esto puede conducir a inanición de los hilos que escriben, y también puede dirigir a

### 31.6 La cena de los filósofos

El problema clásico, que son 5 filósofos, cada uno con un cubierto al lado para un total de 5 cubiertos, pero cada uno necesita dos para poder comer.

#### Una solución mala.

Podemos intentar hacer que todos los filósofos primero obtengan el cubierto de un lado y luego obtengan el otro, pero así cuando todos tengan uno de

los cubiertos, no habrá ningún otro disponible y entrañaría en un deadlock.

#### Otra solución

Consiste en cambiar el orden de por lo menos uno de los filósofos, así el orden de los recursos que usa es diferente y se rompe el ciclo.

### 31.7 Thread throttling

Consiste en sólo dejar acceder a un número determinado de hilos a la vez a una sección crítica, para así evitar un colapso de la aplicación, por ejemplo en una sección que se requiera más memoria que la que tiene el sistema disponible.

Esto se puede lograr con un semáforo inicializado con el número máximo de hilos que se quiere permitir.

### 31.8 Implementando Semáforos

En OS-TEP se propone una variante de los semáforos en la que el valor del semáforo nunca es menor que cero. Aunque no sé muy bien qué implica

### 32. Problemas comunes de concurrencia

Existen muchos bugs de concurrencia que tienden a aparecer en ciertos patrones. Los investigadores han destinado mucho tiempo para ello.