Introducción a Sistemas Operativos: Concurrencia

Clips xxx
Francisco J Ballesteros

1. Cierres

¿Cómo podemos evitar una condición de carrera como la que hemos visto? La respuesta viene de la definición del problema. Necesitamos que la región crítica ejecute atómicamente (de forma indivisible en lo que se refiere a otros que utilizan el mismo recurso). Decimos que necesitamos **exclusión mutua**. Esto es, que si un proceso está en la región crítica, otros no puedan estarlo. Dicho de otro modo, que si un proceso está utilizando el recurso compartido otros no puedan hacerlo.

En la figura 1 puede verse lo que sucede si `cnt++` ejecuta de forma atómica. En este caso, al contrario que antes, uno de los procesos ejecuta `cnt++` antes que otro. El que llega después puede ejecutar `cnt++` partiendo del valor resultante del primero, sin que exista problema alguno. Incluso si no sabemos en qué orden ejecutan los `cnt++`, el valor final resulta correcto.

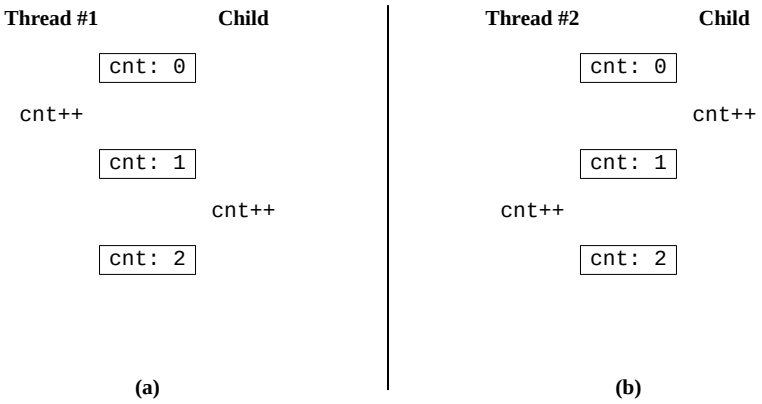

Figura 1: *Incrementando el contador de forma atómica no hay condiciones de carrera, ya sea (a) o (b) lo que suceda en realidad.*

Dependiendo del recurso al que accedamos, tenemos diversas abstracciones para conseguir exclusión mutua en la región crítica. Lo que necesitamos es una abstracción como mecanismo de **sincronización** para que unos procesos se pongan de acuerdo con otros.

En el caso de procesos que comparten memoria podemos utilizar una abstracción llamada **cierre** para conseguir tener exclusión mutua. La idea es rodear la región crítica por código similar a...

```
lock(cierre);
... región crítica...
unlock(cierre);
```

La variable `cierre` y sus dos operaciones se comportan como una "llave" (de ahí el nombre). Sólo un proceso puede tener el cierre. Cuando un proceso llama a `lock`, si el cierre está abierto (libre), el proceso echa el cierre y continúa. Si el cierre está echado (ocupado) el proceso espera (dentro de `lock`) y, cuando esté libre, el proceso echa el cierre y continúa. La operación `unlock` suelta o libera el cierre.

Existen instrucciones capaces de consultar y actualizar una posición de memoria de forma atómica y pueden utilizarse para implementar `lock` y `unlock`. Basta usar un entero y suponer que si es cero el cierre está libre y si no lo es está ocupado.

Dependiendo del tipo de cierre que utilicemos es posible que el proceso esté en un `while` esperando a que el cierre se libere. En tal caso el proceso utiliza el procesador para esperar y decimos que tenemos **espera activa**. A estos cierres se los conoce como **spin locks**. Igualmente, es posible que los cierres cooperen con UNIX y que el sistema pueda bloquear al proceso mientras espera. Esto último es lo deseable, pero has de consultar el manual para ver qué cierres tienes disponibles en tu librería de threads.

Cuando un cierre (u otra abstracción) se utiliza para dar exclusión mutua se lo denomina **mutex**. Suelen usarse las expresiones *coger el mutex* y *soltar el mutex* para indicar que se echa el cierre y se libera.

Cuando utilizamos *pthreads* tenemos a nuestra disposición el tipo de datos `pthread_mutex_t` que representa un mutex y funciones para adquirir y liberar el mutex. Para implementar exclusión mutua basta con usar

```
pthread_mutex_lock(&lock);
...región crítica...
pthread_mutex_unlock(&lock);
```

Donde la variable `lock` puede declararse e inicializarse según

```
pthread_mutex_t lock;
...
pthread_mutex_init(&lock, NULL);
```

Una vez deje de ser útil el mutex, hay que liberar los recursos que usa llamando a

```
pthread_mutex_destroy(&lock);
```

Un aviso. Todas estas llamadas devuelven en realidad una indicación de error. Normalmente `0` is hacen el trabajo y algún otro número si no. Hemos optado por no comprobar estos errores en los programas que mostramos en esta sección para no distraer del código que requiere la programación concurrente. Pero **hay que comprobar los errores** cuando se utilicen en la práctica. Muchas veces sólo fallan si el proceso se queda sin memoria disponible, pero eso no es una excusa para no comprobar errores.

El siguiente programa es similar al del epígrafe anterior pero no presenta condiciones de carrera.

`[safe.c]`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <err.h>
#include <string.h>

typedef struct Cnt Cnt;
struct Cnt {
    int n;
    pthread_mutex_t lock;
};

enum { Nloops = 10 };
static int nloops = Nloops;
```

```
static void*
tmain(void *a)
{
    int i;
    Cnt *cntp;

    cntp = a;
    for(i = 0; i < nloops; i++) {
        pthread_mutex_lock(&cntp->lock);
        cntp->n = cntp->n + 1;
        pthread_mutex_unlock(&cntp->lock);
    }
    return NULL;
}


int
main(int argc, char* argv[])
{
    int i;
    Cnt cnt;
    pthread_t thr[3];
    void *sts[3];

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    cnt.n = 0;
    if(pthread_mutex_init(&cnt.lock, NULL) != 0) {
        err(1, "mutex");
    }
    for(i = 0; i < 3; i++) {
        if(pthread_create(thr+i, NULL, tmain, &cnt) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], sts+i);
        free(sts[i]);
    }
    pthread_mutex_destroy(&cnt.lock);
    printf("cnt is %d\n", cnt.n);
    exit(0);
}
```

Hemos seguido la costumbre de situar el cierre cerca del recurso al que cierra, si ello es posible. En este caso, situamos tanto el cierre como el contador dentro de la misma estructura:

```
typedef struct Cnt Cnt;
struct Cnt {
    int n;
    pthread_mutex_t lock;
};
```

Cuando los threads intentan incrementar el contador, ejecutan

```
pthread_mutex_lock(&cntp->lock);
cntp->n = cntp->n + 1;
pthread_mutex_unlock(&cntp->lock);
```

y uno de ellos llegará a `pthread_mutex_lock` antes que los demás. Ese proceso adquiere el cierre y continúa. Si llegan otros a `pthread_mutex_lock` antes de que el proceso que tiene el mutex lo suelte, quedarán bloqueados dentro de `pthread_mutex_lock` hasta que el cierre quede libre. Si ahora editamos el código y añadimos un `sleep` a mitad del incremento, como hicimos antes, veremos que no tenemos condiciones de carrera. Podemos ver esto además en la siguiente ejecución:

```
unix$ safe 10000
cnt is 30000
unix$
```

¡No se pierden incrementos!

Por cierto, los mutex de *pthread(3)* suelen bloquear el proceso para hacer que espere, por lo que no son *spin locks* y no desperdician procesador.

Otro detalle importante es que es posible inicializar un cierre usando un valor inicial, en lugar de usar `pthread_mutex_init`. Para hacerlo, basta usar código como:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```