Introducción a Sistemas Operativos: Concurrencia

Francisco J Ballesteros

1. Threads y procesos

Hemos utilizado `fork` para crear procesos, lo que hace que los procesos hijo *no* compartan recursos con el padre: tienen su propia copia de los segmentos de memoria, descriptores de fichero, etc. Esa es la abstracción, aunque en realidad, el segmento de texto se suele compartir (dado que es de sólo lectura). El resto de segmentos se comportan como si no se compartiesen, pero UNIX hace que el padre y el hijo los compartan tras degradar sus permisos a sólo-lectura. Cuando un proceso intenta escribir en una de las páginas que "no comparten", UNIX comprueba que, en efecto, el proceso puede escribir en ella y hace una copia de la página que comparten padre e hijo. Como en este punto cada proceso tiene su propia copia de la página, los permisos vuelven a dejarse como lectura-escritura, y el proceso puede completar su escritura pensando que siempre ha tenido su propia copia. A esto se lo conoce como **copy on write**.

Un proceso para el kernel es, principalmente, el flujo de control (su juego de registros ya sea en el procesador o salvado en su pila de kernel y su pila, tanto la de usuario como la de kernel). El resto de recursos puede o no compartirlos con el proceso que lo ha creado.

A la vista de esto, resulta posible crear un nuevo proceso que comparta la memoria con el padre. A esto se lo suele denominar **thread**. El nombre procede de los tiempos en que el kernel no sabía nada de threads y éstos eran *procesos ligeros de usuario* o *corutinas* creadas por la librería de C (u otra similar) sin ayuda del kernel. Desde hace años, los threads son procesos como cualquier otro y el kernel los planifica como a cualquier otro proceso. Lo único que sucede es que algunos procesos comparten segmentos de memoria (y otros recursos) con otros procesos. Eso es todo. Si piensas en los threads como en procesos todo irá bien.

Resulta más cómodo (y es más portable) crear un proceso que comparte los recursos con el padre utilizando la *librería de pthreads* que está instalada en prácticamente todos los sistemas UNIX que utilizando la llamada al sistema involucrada (que suele además variar mucho de unos UNIX a otros).

El siguiente programa crea tres threads que imprimen 5 mensajes cada uno.

[thr.c]:

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

```
static void*
tmain(void *a)
{
    int i;
    char *arg, *sts;

    arg = a;
    for(i = 0; i < 5; i++) {
        write(2, arg, strlen(arg));
    }
    asprintf(&sts, "end %s", strchr(arg, 't'));
    free(arg);
    return sts;
}


int
main(int argc, char* argv[])
{
    int i;
    pthread_t thr[3];
    void *sts[3];
    char tabs[10], *a;

    for(i = 0; i < 3; i++) {
        memset(tabs, '\t', sizeof tabs);
        tabs[i] = 0;
        asprintf(&a, "%st %d\n", tabs, i);
        if(pthread_create(thr+i, NULL, tmain, a) != 0) {
            err(1, "thread");
        }
    }

    for(i = 0; i < 3; i++) {
        pthread_join(thr[i], sts+i);
        printf("join %d: %s\n", i, sts[i]);
        free(sts[i]);
    }
    exit(0);
}
```

El programa crea un nuevo thread utilizando código como

```
pthread_t thr;
...
if (pthread_create(&thr, NULL, func, funcarg) != 0) {
    err(1, "thread");
}
```

La llamada crea un proceso que comparte los recursos con el que hace la llamada y arregla las cosas para que el nuevo flujo de control ejecute `func(funcarg)`. Donde `func` es el *programa principal* o el punto de entrada del nuevo thread y ha de tener una cabecera como

```
void *func(void *arg)
```

El argumento que pasamos al final a `pthread_create` es el argumento con que se llamará a dicho punto

de entrada. Además, la función principal del thread devuelve un `void*` que hace las veces de *exit status* para el thread.

El primer argumento es un puntero a un **tid** o identificador de thread que utiliza la librería de threads para identificar al thread en cuestión. Utilizarás este valor en sucesivas llamadas que se refieran al thread que has creado. Es similar al *pid* de un proceso. Piensa que aunque cada thread tiene un proceso, la librería mantiene más información sobre cada thread y desea utilizar sus propios identificadores.

El programa puede después llamar a

```
void *sts;
...
pthread_join(thr, &sts);
```

para (1) esperar a que el thread termine y (2) obtener su estatus. Dicho de otro modo, `pthread_join` hace las veces de *wait(2)*.

Cuando no deseamos llamar a `pthread_join` para un thread, debemos informar a la librería *pthread* de tal cosa (¡Igual que sucedía con `fork` y `wait`!). Una buena forma de hacerlo es hacer que la función principal del thread llame a

```
pthread_t me;
me = pthread_self();
pthread_detach(me);
```

Pero observa que este código ejecuta en el nuevo thread, no en el código del proceso padre. La función `pthread_self` es como *getpid(2)*, pero devuelve el *thread id* y no el *process id*.

Si ahora vuelves a leer el programa seguramente entiendas las partes que antes te parecían oscuras. Cuando lo ejecutamos, podemos ver una salida parecida a esta:

```
unix$ thr
        t 2
        t 2
        t 2
        t 2
    t 1
t 0
    t 1
t 0
t 0
t 0
t 0
    t 1
    t 1
    t 1
        t 2
join 0: end t 0
join 1: end t 1
join 2: end t 2
unix$
```

Siendo procesos distintos... ¡No sabemos en qué orden van a ejecutar! Luego la salida seguramente difiera si repetimos la ejecución. Pero podemos ver que los tres threads ejecutan concurrentemente que que el programa principal puede esperar correctamente a que terminen y recuperar el estatus de salida de cada uno de ellos.

Seguramente resultará instructivo que ahora intentes leer de nuevo el programa trazando mentalmente cómo ha podido producir la salida que hemos visto.

2. Condiciones de carrera

¿De nuevo?... ¡Sí! En cuanto más de un proceso utiliza el mismo recurso... hay condiciones de carrera. Ahora que podemos compartir memoria entre varios procesos vamos a verlo de nuevo.

El siguiente programa incrementa un contador un número dado de veces (10 por omisión) en tres threads distintos:

[race.c]:

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

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


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

    cntp = a;
    for(i = 0; i < nloops; i++) {
        *cntp = *cntp + 1;
    }
    return NULL;
}
```

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

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    cnt = 0;
    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]);
    }
    printf("cnt is %d\n", cnt);
    exit(0);
}
```

Si lo ejecutamos, el contador debiera ser tres veces el número de incrementos, ¿No?. Y parece que es así...

```
unix$ thr
cnt is 30
unix$
```

Pero... ¡Vamos a ejecutarlo para que cada thread haga 1000 incrementos!

```
unix$ thr 1000
cnt is 2302
unix$
```

¡Otra vez!

```
unix$ thr 1000
cnt is 2801
unix$
```

No te gustaría que pasara esto si se tratase del programa que controla ingresos en tu cuenta corriente. Estamos viendo simplemente el efecto de una condición de carrera en el uso de la variable `cnt`, que es compartida por todos los procesos del programa (el proceso que teníamos desde el programa principal y los tres threads que hemos creado).

Podemos verlo fácilmente si simplificamos el programa para que ejecute sólo dos threads y para que la función que ejecutan sea:

[race2]:

```
...
static int cnt;
static void*
tmain(void *a)
{
    int i;
    for (i =0 ; i < 2; i++) {
        cnt++;
    }
    printf("cnt is %d\n", cnt);
    return NULL;
}
...
```

Ahora haremos dos incrementos en dos threads y hemos cambiado la declaración de `cnt` para que sea una variable global, por simplificar más.

Cuando lo ejecutamos vemos:

```
unix$ race2
cnt is 2
cnt is 4
```

Todo bien.

Cambiemos otra vez el código para que sea:

[race3]:

```
static void*
tmain(void *a)
{
    int i, loc;
    for (i =0 ; i < 2; i++) {
        loc = cnt;
        loc++;
        cnt = loc;
    }
    printf("cnt is %d\n", cnt);
    return NULL;
}
```

Si lo ejecutamos, vemos que todo sigue bien:

```
unix$ race3
cnt is 2
cnt is 4
```

Pero si hacemos el siguiente cambio:

[race4]:

```
static void*
tmain(void *a)
{
    int i, loc;
    for (i =0 ; i < 2; i++) {
        loc = cnt;
        loc++;
        sleep(1);
        cnt = loc;
    }
    printf("cnt is %d\n", cnt);
    return NULL;
}
```

¡La cosa cambia!

```
unix$ race4
cnt is 2
cnt is 2
```

Ambos threads escriben `2` como valor final para `cnt`. Hemos provocado que la condición de carrera se manifieste. Esto quiere decir que, aunque no seamos conscientes, todas las versiones anteriores de este programa están mal y no pueden utilizarse.

El problema de la condición de carrera procede en realidad de la *ilusión* implementada por los procesos: *ejecución secuencial independiente*. Resulta que si tenemos un sólo procesador, la ejecución no es ni secuencial ni independiente. UNIX multiplexa (reparte) el procesador entre los procesos, y aun así pensamos que nuestros programas ejecutan secuencialmente y sin interferencias.

Lo que sucede en realidad es que las instrucciones de los procesos se *mezclan* en un único flujo de control implementado por el procesador. Esto es, ejecutará determinado número de instrucciones de un proceso, luego tendremos un cambio de contexto y ejecutará otro, luego otro, etc. No sabemos cuándo sucederán los cambios de contexto y por tanto no sabemos en qué orden se mezclaran las instrucciones. Se suele llamar **interleaving** (entrelazado) al mezclado de instrucciones, por cierto.

Las primeras veces que hemos utilizado el programa resulta que todas las instrucciones involucradas en

```
cnt++
```

han ejecutado. Cuando hemos cambiado el código para que sea más parecido a las instrucciones que realmente ejecutan

```
loc = cnt;
loc++;
cnt = loc;
```

hemos seguido teniendo (mala) suerte y dichas instrucciones han ejecutado sin interrupción.

Así pues, la ilusión de ejecución secuencial y sin interferencia se ha mantenido. Cuando incrementamos un registro para incrementar la variable el mundo seguía como lo dejamos en la línea anterior y la variable global (en la memoria) seguía teniendo el mismo valor. Cuando actualizamos en la siguiente línea la variable global nadie había consultado ni cambiado la variable mientras ejecutamos.

Al introducir la llamada a `sleep` hemos provocado un cambio de contexto justo en el punto en que tenemos la condición de carrera (durante la consulta e incremento de la global). El efecto puede verse en la figura 1.

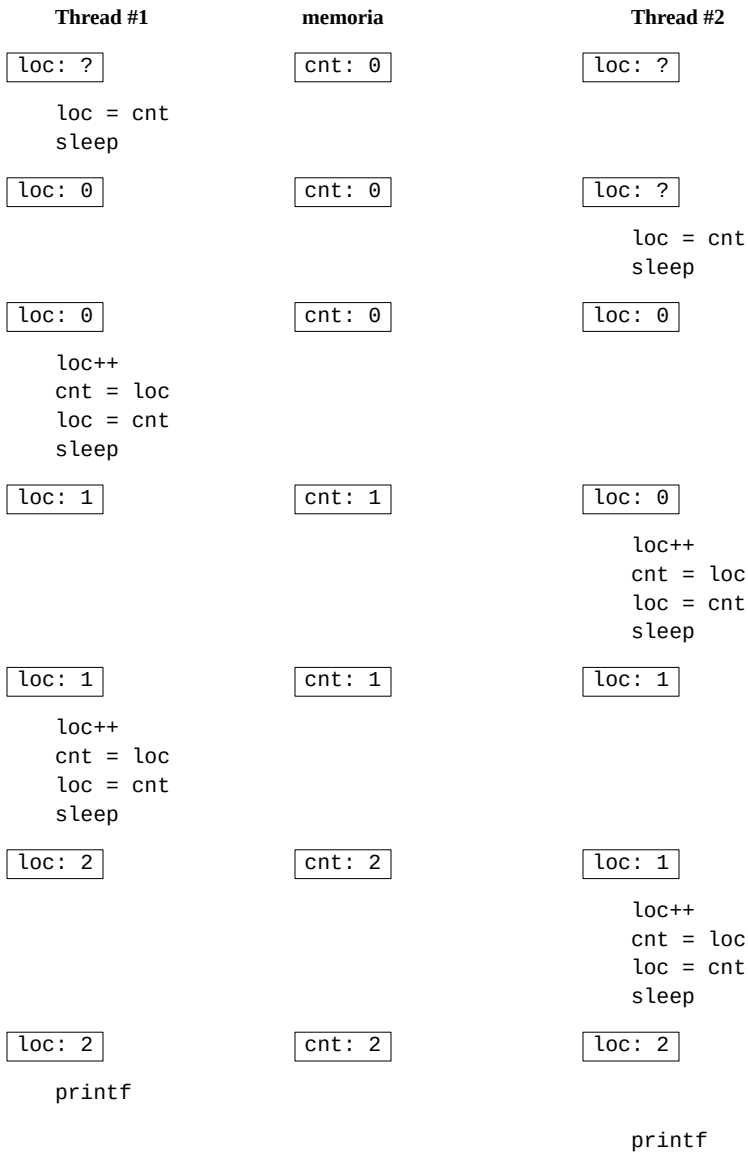

Figura 1: *Un entrelazado de sentencias en ambos procesos que da lugar a una condición de carrera en la última versión del programa.*

El problema consiste en que, después de haber consultado el valor del contador en la local `loc` y antes de que actualicemos el valor del contador, *otro* proceso accede al contador y puede que incluso lo cambie. En la figura podemos ver que el entrelazado ha sido:

1. El thread 1 consulta la variable
2. El thread 2 consulta la variable
3. El thread 1 incrementa su copia de la variable (registro o variable local)
4. El thread 1 incrementa su copia de la variable (registro o variable local)
5. El thread 1 actualiza la variable
6. El thread 2 actualiza la variable

Este *interleaving* pierde un incremento. El problema es que toda la secuencia de código utilizando la variable no ejecuta de forma indivisible (se "cuela" otro proceso que utiliza la misma variable). Si este código

fuese **atómico**, esto es, ejecutase de forma indivisible, no habría condición de carrera; pero no lo es.

Por ello es crítico que no se utilice la variable global desde ningún otro proceso mientras la consultamos, incrementamos y actualizamos y llamamos **región crítica** a dicho fragmento de código. Una *región crítica* es simplemente código que accede a un recurso compartido y que plantea condiciones de carrera si no las evitamos haciendo que ejecute de forma atómica (indivisible con respecto a otros que comparten el recurso).

¿Qué sucedía en el programa inicial que hacía *n* incrementos en 3 threads? Simplemente que hay cierta probabilidad de tener un cambio de contexto en la región crítica. La probabilidad aumenta cuantas más veces ejecutemos la región crítica. Al ejecutar mil veces el incremento en cada thread, *algunos* de los incrementos sufrieron un cambio de contexto en mal sitio, eso es todo.

3. 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 2 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.

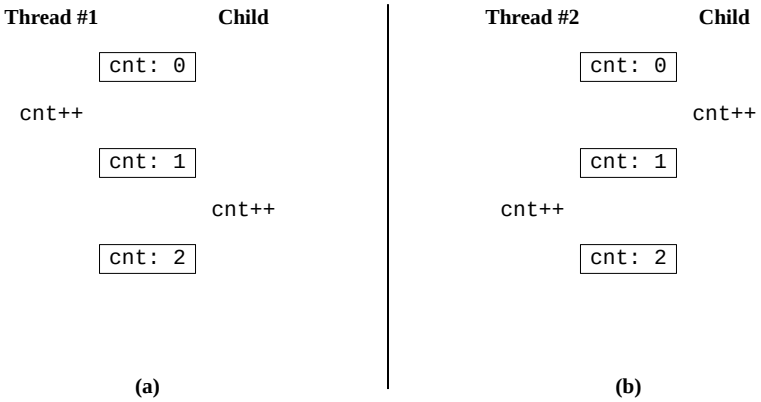

Figura 2: *Incrementando el contador de forma atómica no hay condiciones de carrera, ya sea (a) o (b) lo que suceda en realidad.*

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;
```

4. Cierres en ficheros

El siguiente programa incrementa un contador que tenemos escrito dentro de un fichero. Esto sucede en la realidad en aplicaciones que deben numerar secuencialmente recursos cada vez que ejecuta determinado programa, por ejemplo. Si tenemos un fichero `datafile` que contiene "`3`" y ejecutamos el programa, el fichero pasará a contener `4`. El código del programa es simple:

`[incr.c]`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("datafile", O_RDWR);
    if(fd < 0) {
        err(1, "open");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);

    nb++;
    fprintf(stderr, "set to %d\n", nb);
    snprintf(buf, sizeof buf, "%d", nb);
    lseek(fd, 0, 0);
    if(write(fd, buf, strlen(buf)) != strlen(buf)) {
        err(1, "write");
    }
    close(fd);
    exit(0);
}
```

Hemos incluído multiples prints que no tendríamos normalmente para que veamos qué sucede al ejecutarlo. Vamos a hacerlo:

```
unix$ cat datafile
3
unix$ incr
nb is 3
set to 4
unix$ cat datafile
4
unix$
```

Todo bien.

Pero pongamos un `sleep` de tal forma que el código use ahora

```
nb++;
sleep(5);
```

en lugar de tan sólo incrementar `nb`. Y ahora ejecutemos dos veces el programa:

```
unix$ incr &
[1] 47846
nb is 4
unix$ incr
nb is 4
set to 5
set to 5
unix$ cat datafile
5
unix$
```

¡Hemos perdido un incremento!

Naturalmente, dos procesos que acceden al mismo fichero producen una condición de carrera por compartir el fichero. Necesitamos un cierre. Si ambos procesos compartiesen memoria podríamos utilizar un cierre como los que hemos antes sin ningún problema (¡Aunque el recurso que cierran sea un fichero y esté fuera de la memoria del proceso!). Recuerda que todo esto es un convenio. Hemos quedado en adquirir un cierre antes de entrar en la región crítica, pero es tan sólo un acuerdo.

En este caso, por desgracia, los procesos no comparten memoria. Pero aún podemos solucionar el problema con la ayuda de UNIX. En UNIX es posible adquirir un cierre sobre un fichero e incluso sobre un rango de bytes dentro de un fichero. La llamada al sistema para conseguirlo es *flock(2)*. Concretamente,

```
flock(fd, LOCK_EX);
```

echa el cierre en `fd` de modo exclusivo (como en los cierres que vimos antes) y

```
flock(fd, LOCK_UN);
```

libera el cierre. Si el proceso muere o cierra el descriptor de fichero, el cierre se libera.

Sabiendo esto, el siguiente programa es la versión correcta del programa anterior, sin condiciones de carrera.

[safeincr]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("datafile", O_RDWR);
    if(fd < 0) {
        err(1, "open");
    }
    if(flock(fd, LOCK_EX) != 0){
        err(1, "lock");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);

    nb++;

    fprintf(stderr, "set to %d\n", nb);
    snprintf(buf, sizeof buf, "%d", nb);
    lseek(fd, 0, 0);
    if(write(fd, buf, strlen(buf)) != strlen(buf)) {
        err(1, "write");
    }
    if(flock(fd, LOCK_UN) != 0){
        err(1, "lock");
    }
    close(fd);
    exit(0);
}
```

Recuerda que todo esto es un convenio. ¡Podríamos echar el cierre en un fichero para trabajar en otro! Así pues, tenemos ya la forma de trabajar con ficheros compartidos. Basta pensar dónde y cómo disponemos ficheros que usamos como cierre. Por ejemplo, muchos programas de correo utilizan un fichero llamado `.LOCK` en el directorio que contiene los buzones de correo (que son ficheros). Para utilizar los ficheros en dicho directorio, estos programas llaman a `flock` sobre `.LOCK` y luego trabajan con los buzones. Cuando terminan de trabajar, sueltan el cierre. No es muy diferente a lo que hicimos nosotros utilizando una variable `lock` para tener exclusión mutua en el acceso a un contador `cnt`.

5. Cierres de lectura/escritura

Como en ocasiones nos preguntamos qué valor tendrá el contador que incrementamos en el programa anterior, vamos a realizar un programa para imprimirlo. Podríamos utilizar *cat(1)*, naturalmente, pero vamos a hacer un programa que pueda ver el valor sin condiciones de carrera.

`[safecat]`:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("datafile", O_RDONLY);
    if(fd < 0) {
        err(1, "open");
    }
    if(flock(fd, LOCK_EX) != 0){
        err(1, "lock");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);
    if(flock(fd, LOCK_UN) != 0){
        err(1, "lock");
    }
    close(fd);
    exit(0);
}
```

Podemos usarlo sin condiciones de carrera:

```
unix$ safecat
nb is 5
unix$
```

¿Qué sucede si ejecutamos tres `safecat` y un `safeinr` simultáneamente? Todos ellos adquieren el cierre sobre el fichero para trabajar en él con exclusión mutua y, aunque el orden en que consigan ejecutar y echar el cierre variará, podríamos tener una ejecución como la que vemos en la figura 3.

Si hay muchos procesos leyendo (ejecutando `safecat`) tardaremos mucho en ejecutarlo todo. Pero hay una posibilidad de mejorar las cosas: puesto que los lectores sólo leen el fichero, es posible ejecutar más de un lector a la vez sin exclusión mutua respecto a otros lectores. Dicho de otro modo, podríamos tener un

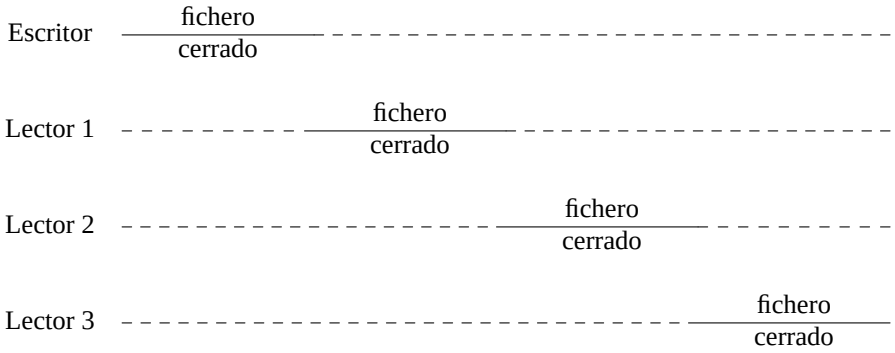

Figura 3: *Múltiples lectores y escritores de un fichero con un cierre exclusivo. Sólo puede ejecutar uno cada vez dentro de la región crítica.*

escritor o cualquier número de lectores, pero no ambas cosas. Existe un tipo de cierre que permite exclusión mutua entre lectores y escritores. Permite lectores concurrentes en exclusión con escritores. Naturalmente, los escritores excluyen otros escritores también. Si utilizamos este tipo de cierre, la ejecución podría ser como se ven la figura 4. Como puede verse, los procesos han de esperar en menos y, conjuntamente, terminamos antes.

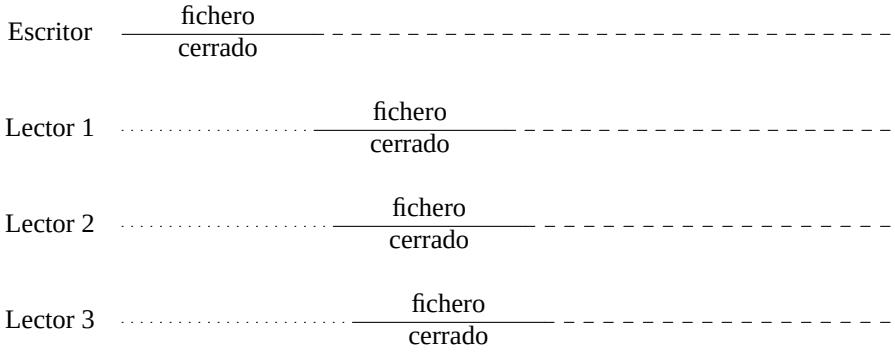

Figura 4: *Con un cierre de tipo lectores/escritores permitimos múltiples lectores concurrentes manteniendo la exclusión mutua de los escritores.*

Otra forma de verlo (que en realidad coincide con la implementación) es pensar que los escritores comparten el mutex con otros lectores cuando lo adquieren.

En nuestro programa podemos conseguir este efecto haciendo que `safecat` adquiera el cierre en modo *lector*. Si hacemos tal cosa, estamos utilizando el cierre del fichero como un cierre de tipo lectores/escritores, también llamado **read/write lock**.

[safecatr]:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <err.h>
#include <string.h>
#include <sys/file.h>

int
main(int argc, char* argv[])
{
    int fd, n, nb;
    char buf[100];

    fd = open("afile", O_RDONLY);
    if(fd < 0) {
        err(1, "open");
    }
    if(flock(fd, LOCK_SH) != 0){
        err(1, "lock");
    }
    n = read(fd, buf, sizeof(buf)-1);
    if(n < 0){
        err(1, "read");
    }
    buf[n] = 0;
    nb = atoi(buf);
    fprintf(stderr, "nb is %d\n", nb);
    if(flock(fd, LOCK_UN) != 0){
        err(1, "lock");
    }
    close(fd);
    exit(0);
}
```

Ahora tenemos algunos (lectores) que adquieren un cierre usando `LOCK_SH` (*shared*) y otros (escritores) que lo hacen usando `LOCK_EX` (*exclusive*).

6. Cierres de lectura/escritura para threads

La librería de *pthreads* dispone de cierres de tipo lectura/escritura. Su uso es similar al que hemos visto para usar los mutex de *pthread*, pero las variables de tipo cierre se declaran e inicializan como en

```
pthread_rwlock_t rwlock;
...
pthread_rwlock_init(&rwlock);
```

o bien

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Un lector usaría estas llamadas para proteger su región crítica:

```
pthread_rwlock_rdlock(&rwlock);
    ... región crítica ...
pthread_rwlock_unlock(&rwlock);
```

Y un escritor estas otras:

```
pthread_rwlock_wrlock(&rwlock);
    ... región crítica ...
pthread_rwlock_unlock(&rwlock);
```

Una vez deja de ser necesario el cierre hay que liberar sus recursos con

```
pthread_rwlock_destroy(&rwlock);
```

Si un proceso necesita un cierre como lector y posteriormente convertirse en escritor, suele ser mejor mantener el cierre como escritor todo el tiempo.

7. Deadlocks

Rara vez necesitaremos un único cierre. O, dicho de otro modo, rara vez necesitaremos un único recurso compartido. Lo normal es compartir diversos recursos. En principio cada recurso puede utilizar un cierre distinto y los procesos que necesitan acceder al recurso pueden adquirir su cierre durante la región crítica. Pero piensa lo que sucede si un proceso ejecuta

```
lock(a);
lock(b);
...
```

y otro en cambio ejecuta

```
lock(b);
lock(a);
```

Si ejecutamos el primer `lock` de cada proceso, ninguno podrá continuar jamás. Cada uno tiene un cierre y necesita el que tiene el otro. El efecto neto es que los procesos se quedan bloqueados de por vida y no terminan. A esto lo denominamos **deadlock** o **interbloqueo**. ¿Recuerdas qué sucedía si un proceso deja abierto un descriptor de escritura en un pipe y se pone a leer del mismo? Efectivamente, es un deadlock.

En el caso de cierres, para evitar el problema suele bastar fijar un **orden** a la hora de adquirir los recursos. Si todos adquieren los cierres que necesitan siguiendo el orden acordado, no tenemos interbloqueos. Por ejemplo, podemos acordar que el cierre de `a` ha de tomarse antes que el de `b` y el de `b` antes que el de `c`. De ser así, podríamos tener código para un proceso que ejecute

```
lock(a);
lock(c);
...
```

otro que ejecute

```
lock(a);
lock(b);
```

y otro

```
lock(b);
lock(c);
```

y en ningún caso tendríamos un bloqueo. Si alguien necesita un cierre nadie puede tenerlo y además estar esperando los que ya tenemos nosotros: no hay deadlocks.

Esto puede resultar complicado y en ocasiones se opta por usar un único cierre para todos los recursos, a lo que se suele denominar *giant lock* o "cierre gordo". Pero naturalmente se limita la concurrencia y se reduce el rendimiento. No obstante, hay menos posibilidades de tener condiciones de carrera por olvidar echar un cierre en el momento adecuado.

Hay veces en que se opta por dejar que suceda el deadlock y, tras detectarlo o comprobar que estamos ante un posible deadlock, soltar todos los cierres y volverlo a intentar. Esto no es realmente una solución dado que podríamos volver a caer en el interbloqueo una vez tras otra (aunque no es 100% seguro que siempre suceda tal cosa). A este tipo de bloqueos (que podrían romperse con cierta probabilidad) se los denomina **livelock**.

Lo que es más, puede que algún proceso tenga mala suerte y, si aplica esta estrategia, nunca consiga los cierres que necesita porque siempre gane otro proceso al obtener uno de los cierres que necesita. El pobre proceso continuaría sin poder trabajar, a lo que se denomina **starvation**, o **hambruna**.

8. Semáforos

Quizá la abstracción más conocida para sincronizar procesos en programación concurrente y controlar el acceso a los recursos sean los **semáforos**. Un semáforo es en realidad un contador que indica cuántos *tickets* tenemos disponibles para acceder a determinado recurso. No son muy diferentes de un contador de entradas para acceder al cine. Si no hay entradas, no se puede entrar al cine. Del mismo modo, si un semáforo está a `0`, no se puede acceder al recurso de que se trate.

La parte interesante del semáforo es que dispone de dos operaciones:

```
down(sem);
```

y

```
up(sem);
```

que (respectivamente) adquieren un *ticket* y lo liberan.

- La primera, `down`, toma un ticket del semáforo (cuyo valor se decrementa) cuando existen tickets (el valor es positivo). De no haber tickets disponibles (cuando el semáforo es cero), `down` *espera* a que existan tickets libres y entonces toma uno.
- La llamada `up` simplemente libera un ticket. Si alguien estaba esperando en un `down`, lo adquiere y continúa. Si nadie estaba esperando en un `down`, el ticket queda en el semáforo, que pasa a incrementarse.

A la abstracción se la denomina semáforo puesto que su inventor pensó en semáforos ferroviarios que controlan el acceso a las vías. Por cierto que en muchas implementaciones se utilizan los nombres

```
P(sem);
```

y

```
V(sem);
```

en lugar de `down` y `up`, y en otras implementaciones se utiliza

```
wait(sem);
```

y

```
signal(sem);
```

en su lugar. Nosotros usaremos `down` y `up` para evitar confusiones con otras operaciones.

Dado un semáforo, podemos implementar un mutex utilizando código como

```
Sem sem = 1;
...
down(sem);
...región crítica...
up(sem);
```

Puesto que sólo hay un ticket en el semáforo, sólo un proceso puede adquirirlo, con lo que tenemos exclusión mutua.

9. Semáforos en UNIX

Existen diversas implementaciones de semáforos en UNIX. Normalmente tienes disponible una denominada *posix semaphores*, que puedes combinar con los threads de la librería de *pthread(3)*.

Así pues... ¡Cuidado! Es posible que si usas determinado tipo de semáforos estos sólo existan en tu sistema y no en otros UNIX. Por ejemplo, Linux dispone de semáforos con nombre y sin nombre, pero OS X suministra semáforos con nombre y no dispone del otro tipo de semáforos. Nosotros usaremos sólo semáforos con nombre que tenemos disponibles en gran parte de los sistemas UNIX hoy en día.

Veamos el siguiente programa:

[semcnt.c]:

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

static int cnt;
static sem_t *sem;

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


static void*
tmain(void *a)
{
    int i;

    for(i = 0; i < nloops; i++) {
        if (sem_wait(sem) < 0) {
            err(1, "down");
        }
        cnt++;
        if (sem_post(sem) < 0) {
            err(1, "up");
        }
    }
    return NULL;
}
```

```
int
main(int argc, char* argv[])
{
    int i;
    pthread_t thr[3];
    void *sts;
    char name[1024];

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    snprintf(name, sizeof name, "/sem.%s.%d", argv[0], getpid());
    sem = sem_open(name, O_CREAT, 0644, 1);
    if (sem == NULL) {
        err(1, "sem creat");
    }
    printf("sem '%s' created\n", name);
    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);
    }
    if (sem_close(sem) < 0) {
        warn("sem close");
    }
    sem_unlink(name);

    printf("cnt is %d\n", cnt);
    exit(0);
}
```

Se trata una vez más de nuestro programa para incrementar un contador en tres threads el número indicado de veces, y podemos comprobar que funciona sin condiciones de carrera:

```
unix$ semcnt 10000
sem '/sem.semcnt.49029' created
cnt is 30000
unix$
```

Lo primero que hemos de tener en cuenta es que estos semáforos *tienen nombre* y existen como una abstracción que suministra el kernel. UNIX mantendrá un record dentro con la implementación del semáforo. Eso quiere decir que hemos de destruirlo cuando no sean útil.

Puesto que tienen nombre, hemos de tener cuidado de no utilizar nombres que usen otros programas. En nuestro caso optamos por construir un nombre a partir de "`sem`" (el nombre de nuestra variable), `argv[0]` (el nombre de nuestro programa) y `getpid()`. Por ejemplo,

```
/sem.semcnt.49029
```

es el nombre del semáforo que hemos creado en la ejecución anterior. Haciéndolo así, es imposible que colisionemos con otros nombres de semáforo de nuestro programa o de otros.

El semáforo lo ha creado la llamada

```
sem = sem_open(name, O_CREAT, 0644, 1);
if (sem == NULL) {
    err(1, "sem creat");
}
```

que es similar a *open(2)* cuando se usa para crear fichero. Esta vez creamos un semáforo y no un fichero. Tras los permisos se indica *qué número de tickets* queremos inicialmente en el semáforo. En este caso `1` dado que es un mutex.

Cuando todo termine, necesitamos cerrar y destruir el semáforo utilizando

```
if (sem_close(sem) < 0) {
    warn("sem close");
}
sem_unlink(name);
```

Una vez creado, podemos compartir el semáforo con otros procesos que no compartan memoria. Basta abrir el semáforo en ellos utilizando

```
sem = sem_open(semname, 0);
if (sem == NULL) {
    // no existe!
}
```

después de haberlo creado. No obstante, es mejor utilizar en ellos la misma llamada que usamos en nuestro programa, indicando `O_CREAT` para que el semáforo se cree si no existe.

10. ¿Y si algo falla?

Un problema con este tipo de semáforos es que siguen existiendo hasta que se llame a `sem_unlink` para borrarlos. Si nuestro programa falla, el semáforo seguirá existiendo en el kernel hasta que rearranquemos o detengamos el sistema. Lamentablemente, no disponemos de comando alguno para listar los semáforos que existen y es posible que dejemos semáforos perdidos si nuestro programa falla.

Un remedio paliativo es usar como nombre del semáforo uno que no incluya el *pid* del proceso y tan sólo use el nombre de nuestra aplicación y escribir un programa que borre el semáforo (o borrarlo antes de crearlo por si habíamos dejado alguno en ejecuciones anteriores.

Otro problema importante es que si un proceso muere mientras tiene un ticket el ticket se pierde.

En conclusión, si se desea un mutex es mucho mejor utilizar un cierre en un fichero si hemos de sincronizar procesos distintos que no forman parte del mismo programa.

Este problema afecta a muchos otros semáforos y abstracciones disponibles en UNIX. Presta atención a qué hace el sistema ante una muere prematura de un proceso que tiene un mutex. Lo deseable es que el mutex se libere, pero posiblemente no suceda tal cosa. Los cierres en ficheros con `flock` si que se comportan correctamente y es por ello que son populares a la hora de conseguir un mutex para compartir recursos entre procesos totalmente distintos.

Cuando los procesos forman parte del mismo programa basta con que los mutex que creamos se liberen si el programa muere. Si uno de los procesos del programa muere prematuramente, esto se debe a un bug y dado que es el mismo programa no afecta a otras aplicaciones, por lo que no es un problema en realidad: habrá que depurar el error y listo.

11. Semáforos con pipes

Vamos a implementar un semáforo que podremos usar en aplicaciones que compartan un proceso padre (o ancestro) común. La idea es utilizar un pipe como semáforo y guardar tantos bytes en el semáforo como tickets queramos tener: Para hacer un `down` leeremos y byte y para hacer un `up` escribiremos un byte.

Con esta idea, el único problema es que es preciso crear el semáforo antes de llamar a `fork` (para que todos lo procesos lo compartan), si es que usamos `fork`. Si usamos threads, dado que comparten los descriptores de fichero, no tenemos problemas.

Este es el interfaz para nuestros semáforos

[sem.h]:

```
typedef struct Sem Sem;
struct Sem {
    int fd[2];
};

int semdown(Sem *s);
int semup(Sem *s);
int semcreat(Sem *s, int val);
void semclose(Sem *s);
```

y esta es la implementación:

[sem.c]:

```
#include <stdio.h>
#include <unistd.h>
#include "sem.h"

int
semdown(Sem *s)
{
    char c;

    if (read(s->fd[0], &c, 1) != 1) {
        return -1;
    }
    return 0;
}


int
semup(Sem *s)
{
    if (write(s->fd[1], " ", 1) != 1) {
        return -1;
    }
    return 0;
}
```

```
void
semclose(Sem *s)
{
    close(s->fd[0]);
    close(s->fd[1]);
    s->fd[0] = s->fd[1] = -1;
}


int
semcreat(Sem *s, int n)
{
    int i;

    if (pipe(s->fd) < 0) {
        return -1;
    }
    for (i = 0; i < n; i++) {
        if (semup(s) < 0) {
            semclose(s);
            return -1;
        }
    }
    return 0;
}
```

Una vez los tenemos, podemos utilizarlos como en nuestro programa de ejemplo, que hemos adaptado para utilizar estos semáforos:

[semcnt.c]:

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

enum { Nloops = 10 };
static int nloops = Nloops;
static int cnt;
static Sem sem;
```

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

    for(i = 0; i < nloops; i++) {
        if (semdown(&sem) < 0) {
            err(1, "down");
        }
        cnt++;
        if (semup(&sem) < 0) {
            err(1, "up");
        }
    }
    return NULL;
}


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

    if(argc > 1) {
        nloops = atoi(argv[1]);
    }
    if (semcreat(&sem, 1) < 0) {
        err(1, "sem creat");
    }
    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);
    }
    semclose(&sem);
    printf("cnt is %d\n", cnt);
    exit(0);
}
```

Ahora podemos compilarlo y ejecutarlo sin problemas:

```
unix$ cc -c sem.c
unix$ cc -c semcnt.c
unix$ cc -o semcnt semcnt.o sem.o
unix$ semcnt 10000
cnt is 30000
unix$
```

Un problema con estos semáforos es que no podemos hacer que el número de tickets supere el número de bytes que caben en el buffer del pipe. ¿Qué pasaría si lo hacemos? Una ventaja es que cuando nuestro programa termine su ejecución se cerrarán sus descriptores de fichero incluso si hemos olvidado llamar a `semclose` y el semáforo (el pipe) se destruirá sin dejar recursos perdidos en el sistema. Otra ventaja es que podemos usarlos tanto si creamos procesos que llamen a `fork` como si usamos `pthread_create`.

12. Buffers compartidos: el productor/consumidor

Un problema que aparece a todas horas en programación concurrente es el del **productor-consumidor**, también conocido como el del **buffer acotado**. Se trata de tener procesos que producen cosas y procesos que las consumen, como puede verse en la figura 5. Cuando el buffer no tiene límite se denomina *productor-consumidor* al problema y en otro caso se le conoce como el problema del *buffer acotado*. ¡Los pipes son un caso de dicho problema!

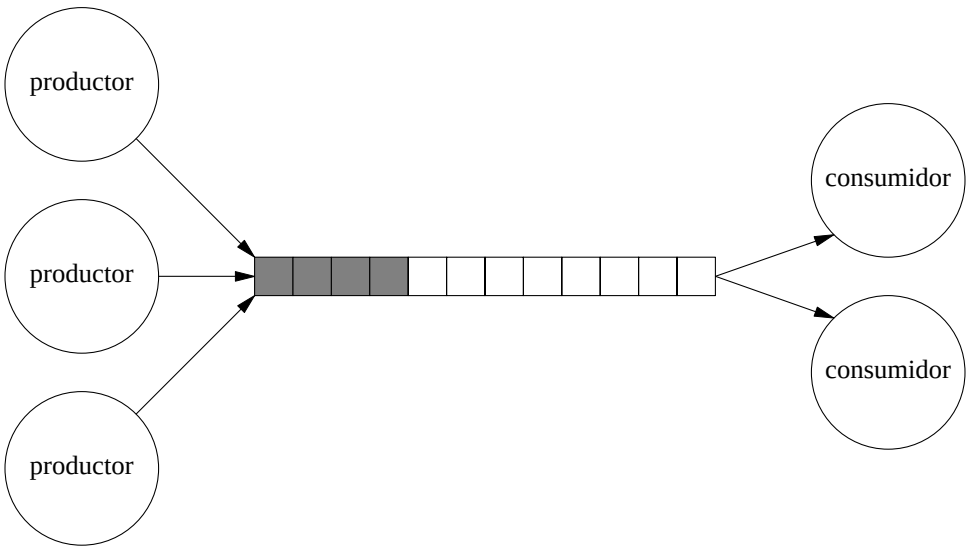

Figura 5: *El problema del productor consumidor con un buffer acotado.*

La solución es sencilla si pensamos en lo siguiente:

- Para producir un ítem es preciso tener un hueco en el buffer
- Para consumir in ítem es preciso tener un ítem en el buffer
- Para operar con el buffer es preciso utilizar un mutex.

La idea es tener un semáforo que represente los ítems en el buffer, otro que represente los huecos en el buffer y otro que represente el mutex:

- Cuando queramos un hueco basta pedirlo: `down(huecos)`.
- Cuando queramos un ítem basta pedirlo: `down(ítems)`.

El mutex ya sabes manejarlo. Naturalmente, si alguien produce un ítem deberá llamar a `up(ítems)` e, igualmente, si alguien produce un hueco deberá llamar a `up(huecos)`. Si piensas en los semáforos como en cajas con tickets, todo esto te resultará natural.

Primero vamos a implementar un buffer compartido que, en nuestro caso, será una cola de caracteres. Vamos a mostrar y discutir el código en el mismo orden en que aparece en el fichero `pc.c`, secuencialmente de arriba a abajo.

[pc.c]:

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

enum {QSIZE = 4};

typedef struct Queue Queue;
struct Queue {
    Sem mutex, nchars, nholes;
    char buf[QSIZE];
    int hd, tl;
};
```

En la cola `Queue` guardamos los tres semáforos que hemos mencionado, el buffer para guardar los caracteres y la posición de la cabeza (`hd`) y cola (`tl`) en el buffer. Insertaremos caracteres al final de la cola (en `tl`) y los tomaremos del principio, de `hd`.

Inicializar la cola requiere crear los semáforos y poco más:

```
static int
qinit(Queue *q)
{
    memset(q, 0, sizeof *q);
    if (semcreat(&q->mutex, 1) < 0) {
        return -1;
    }
    if (semcreat(&q->nchars, 0) < 0) {
        semclose(&q->mutex);
        return -1;
    }
    if (semcreat(&q->nholes, QSIZE) < 0) {
        semclose(&q->mutex);
        semclose(&q->nchars);
        return -1;
    }
    return 0;
}
```

Tomamos la precaución de dejar todos los campos a valores nulos llamando a `memset` para que todo esté bien inicializado.

Terminar de usar la cola requiere cerrar nuestros semáforos:

```
static void
qterm(Queue *q)
{
    semclose(&q->mutex);
    semclose(&q->nchars);
    semclose(&q->nholes);
    q->hd = q->tl = 0;
}
```

¡Ahora hay que hacer la parte interesante! Para poner un carácter en la cola necesitamos y hueco, luego ponerlo mientras tenemos el mutex (para evitar condiciones de carrera en el uso de la cola) y además hemos de echar un ticket al semáforo que indica cuántos ítems hay en la cola:

```
static int
qput(Queue *q, int c)
{
    if (semdown(&q->nholes) < 0) {
        return -1;
    }
    if (semdown(&q->mutex) < 0) {
        return -1;
    }
    q->buf[q->tl] = c;
    q->tl = (q->tl+1)%QSIZE;
    if (semup(&q->mutex) < 0) {
        return -1;
    }
    if (semup(&q->nchars) < 0) {
        return -1;
    }
    return 0;
}
```

Tomar un carácter de la cola es simétrico con respecto a ponerlo. Esta vez pedimos ítems y generamos huecos:

```
static int
qget(Queue *q)
{
    int c;

    if (semdown(&q->nchars) < 0) {
        return -1;
    }
    if (semdown(&q->mutex) < 0) {
        return -1;
    }
    c = q->buf[q->hd];
    q->hd = (q->hd+1)%QSIZE;
    if (semup(&q->mutex) < 0) {
        return -1;
    }
    if (semup(&q->nholes) < 0) {
        return -1;
    }
    return c;
}
```

Ya podemos declarar la cola:

```
static Queue q;
```

Un *productor* será un proceso que pone caracteres en la cola. En nuestro caso cada productor pondrá 10:

```
static void*
tput(void *a)
{
    char *s;
    int i;

    s = a;
    for (i = 0; i < 10; i++) {
        if (qput(&q, *s) < 0) {
            err(1, "qput");
        }
    }
    return NULL;
}
```

Hemos pasado como parámetro un puntero a un `char` para que cada productor ponga un carácter distinto en la cola.

El consumidor va a sacar de la cola todo lo que pueda hasta que obtenga un `0`, con lo que marcaremos el final de los datos:

```
static void*
tget(void *a)
{
    int c;
    char buf[1];

    for (;;) {
        c = qget(&q);
        if (c == 0) {
            break;
        }
        if (c < 0) {
            err(1, "qget");
        }
        buf[0] = c;
        if (write(1, buf, 1) != 1) {
            err(1, "write");
        }
    }
    return NULL;
}
```

Por último, nuestro programa principal inicializará la cola y creará dos productores y un consumidor.

```
int
main(int argc, char* argv[])
{
    pthread_t p1, p2, g;
    void *sts;

    if (qinit(&q) < 0) {
        err(1, "qinit");
    }
    if (pthread_create(&p1, NULL, tput, "a") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&p2, NULL, tput, "b") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&g, NULL, tget, NULL) != 0) {
        err(1, "thread");
    }
    pthread_join(p1, &sts);
    pthread_join(p2, &sts);
    if (qput(&q, 0) < 0) {
        err(1, "qput");
    }
    pthread_join(g, &sts);
    write(1, "\n", 1);
    qterm(&q);
    exit(0);
}
```

¡Listos para ejecutarlo!

```
unix$ pc
bbaabaaaaababababbabbb
unix$
```

Este problema es muy importante. En casi todos los programas utilizarás algún tipo de buffer compartido y necesitarás código que resuelva el problema del productor-consumidor. Además, es un buen problema para terminar de entender cómo se utilizan los semáforos y cómo funcionan. Observa que si el buffer se llena, el productor espera a tener hueco. Igualmente, si el buffer se vacía, el consumidor espera a tener algo que consumir. ¡Los pipes funcionan de este modo!

13. Monitores

Existe otra abstracción básica para conseguir la sincronización entre procesos a la hora de acceder a recursos compartidos. Se trata del **monitor**.

Un monitor es una abstracción que, en teoría, suministra el lenguaje de programación y presenta un aspecto similar a un módulo o paquete. La diferencia con respecto a un módulo (o paquete) radica en que se garantiza que *sólo un proceso puede ejecutar dentro del monitor en un momento dado*. Dicho de otro modo, tenemos exclusión mutua en el acceso al monitor.

Los datos compartidos se declararían dentro del monitor y sólo pueden ser utilizados llamando a operaciones del monitor.

Así pues, podríamos utilizar un contador compartido sin tener condiciones de carrera si escribimos algo como:

```
monitor sharedcnt;
static int cnt;
void
incr(int delta)
{
    cnt += delta;
}
int
get(void)
{
    return cnt;
}
```

La idea es que desde fuera del monitor, podemos utilizar llamadas del estilo a

```
sharedcnt c;
c.incr(+3);
printf("val is %d\n", c.get());
```

sin tener que preocuparlos por la programación concurrente.

Esta abstracción es tan simple de usar y suele entenderse tan bien que habitualmente siempre se programa pensando en ella. ¡Tengamos monitores o no! De hecho, rara vez tenemos monitores de verdad. Normalmente tenemos las herramientas para implementarlos y para programar pensando en ellos, y ese es el caso en UNIX.

Para implementar un monitor basta con declarar un mutex para el monitor y

- adquirir el mutex al principio de cada operación del monitor y
- soltar el mutex al final de cada operación del monitor.

Tan sencillo como eso.

Por ejemplo, para el caso del contador compartido podríamos implementar el monitor como una estructura que contiene el contador y el mutex del monitor: Una vez más, vamos a describir el código en el mismo orden en que aparece en el fichero en C del programa.

[cntmon.c]:

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

typedef struct Cnt Cnt;
struct Cnt {
    Sem mutex;
    int val;
};
static Cnt cnt;
```

En este caso, `cnt` es nuestro monitor. Pero necesitamos programar las operaciones del monitor. Primero, una para inicializarlo

```
static int
cntinit(Cnt *c)
{
    memset(c, 0, sizeof *c);
    if (semcreat(&c->mutex, 1) < 0) {
        return -1;
    }
    return 0;
}
```

y otra para terminar su operación

```
static void
cntterm(Cnt *c)
{
    semclose(&c->mutex);
}
```

Después, una función para cada operación del monitor teniendo cuidado de mantener el mutex del monitor cerrado durante toda la función.

```
static int
cntincr(Cnt *c, int delta)
{
    if (semdown(&c->mutex) < 0) {
        return -1;
    }
    c->val += delta;
    if (semup(&c->mutex) < 0) {
        return -1;
    }
    return 0;
}
[verb
static int
cntget(Cnt *c, int *valp)
{
    if (semdown(&c->mutex) < 0) {
        return -1;
    }
    *valp = c->val;
    if (semup(&c->mutex) < 0) {
        return -1;
    }
    return 0;
}
```

Y ya tenemos implementado el monitor. Ahora podemos declararlo:

```
static Cnt c;
```

Un thread que desee incrementar el contador utiliza la operación del monitor correspondiente para hacerlo:

```
static void*
tincr(void *a)
{
    if (cntincr(&c, 1) < 0) {
        err(1, "cntintr");
    }
    return NULL;
}
```

Nótese como podemos utilizar `cntintr` casi como en un programa secuencial, sin pensar mucho en la programación concurrente.

El programa principal va a inicializar el monitor, crear varios threads que incrementen el contador que contiene y por último imprimir cuánto vale dicho contador utilizando la operación `cntget` del monitor.

```
int
main(int argc, char* argv[])
{
    pthread_t thr[10];
    void *sts;
    int i, val;

    if (cntinit(&c) < 0) {
        err(1, "cntinit");
    }
    for (i = 0; i < 10; i++) {
        if (pthread_create(&thr[i], NULL, tincr, NULL) != 0) {
            err(1, "thread");
        }
    }
    for(i = 0; i < 10; i++) {
        pthread_join(thr[i], &sts);
    }
    if (cntget(&c, &val) < 0) {
        err(1, "cntget");
    }
    printf("val %d\n", val);
    cntterm(&c);
    exit(0);
}
```

Si ejecutamos el programa podemos utilizar el contador compartido sin condiciones de carrera:

```
unix$ cntmon
val 10
unix$
```

¿Podríamos entonces programar

```
if (cntget(&c, &val) < 0) {
    err(1, "cntget");
}
if (val > 0) {
    if (cntincr(&c, -1) < 0) {
        err(1, "cntincr");
    }
}
```

para decrementar un contador sólo si es positivo? En teoría el monitor nos permite solucionar las condiciones de carrera... ¿No? ¡Lo que no impide es que cometamos estupideces!

Pensemos. La llamada a `cntget` funciona correctamente, y deja en `val` el valor del contador. Igualmente, la llamada a `cntincr` incrementa el contador (en `-1` en este caso). Lo que ocurre es que entre una y otra llamadas podría ser que otro proceso entre y decremente el contador. Este es el problema de no mantener cerrado el recurso durante *toda la región crítica*.

En este ejemplo, todo el código debería formar parte de una operación del monitor: *decrementar-si-podemos*.

14. Variables condición

Vamos a implementar el problema del buffer acotado utilizando monitores. Al contrario que cuando utilizamos semáforos, aquí la idea es poder tener operaciones en el monitor que podamos llamar olvidando la concurrencia.

En principio tendríamos una operación para poner un ítem en el buffer y otra para consumirlo. Ambas utilizarían un buffer declarado *dentro* del monitor y ambas tendrían el cierre (o el mutex) del monitor mientras ejecutan. Luego no habría condiciones de carrera en el acceso al buffer.

El problema viene en cuanto vemos que para implementar `put` necesitamos esperar a tener un hueco para poder continuar. De tener monitores, nos gustaría poder escribir

```
void
put(int item)
{
    if(buffer lleno) {
        wait until(tenemos hueco)
    }
    buffer[nitems++] = item;
}
```

El código se entiende, ¿No? Dentro de `put` tenemos el mutex del monitor, no te preocupes por condiciones de carrera. Habría bastado

```
buffer[nitems++] = item;
```

si el buffer nunca se llena. El problema es que si el buffer puede llenarse hay que esperar a tener un hueco antes de consumirlo.

Igualmente, el consumidor podría utilizar una operación como la que sigue:

```
int
get(void)
{
    if(buffer vacio) {
        wait until(tenemos un item)
    }
    return buffer[--nitems];
}
```

Primero esperamos si es que el buffer está vacío a que deje de estarlo y luego consumimos uno de los elementos del buffer.

Para poder esperar hasta que se cumpla determinada condición que necesitamos *dentro de un monitor* tenemos **variables condición**. Son variables que representan una condición y tienen dos operaciones:

- `wait`: espera incondicionalmente a que se cumpla la condición
- `signal`: avisa de que la condición se cumple.

Nótese que estamos comprobando con un `if` si la condición se cumple o no antes de esperar a que se cumpla. Dicho de otro modo, `wait` en una variable condición duerme al proceso incondicionalmente. En cambio, `wait` en un semáforo (recuerda que era un posible nombre para `down`) sólo duerme al proceso si el semáforo está sin tickets.

Otra forma de verlo es que con los semáforos es el semáforo el que se ocupa de si tenemos que dormir o no. Nosotros tan sólo pedimos un ticket y cuando lo tengamos `down` (o `wait`) nos dejará continuar. Pero con las variables condición somos nosotros los que decidimos esperar cuando vemos que no podemos continuar.

Para que el código de nuestro problema esté completo falta llamar a `signal` cuando se cumplan las condiciones. Si le damos un repaso, este sería nuestro código por el momento:

```
Cond hayhuecos, hayitems;
void
put(int item)
{
    if(ntimes == SIZEOFBUFFER) {
        wait(hayhuecos);
    }
    buffer[nitems++] = item;
    signal(hayitems);
}


int
get(void)
{
    int item;
    if(nitems == 0) {
        wait(hayitems);
    }
    item = buffer[--nitems];
    signal(hayhuecos);
    return item;
}
```

Un buen nombre para una variable condición es el nombre de la condición. Cuando se llama a `wait`, el proceso se duerme soltando el mutex del monitor para que otros procesos puedan utilizar el monitor mientras dormimos. Cuando se llama a `signal`, uno de los procesos que duermen despierta. Si no hay procesos dormidos esperando, `signal` no hace nada.

Esto quiere decir que deberíamos llamar a `signal` justo al final de la operación, de tal modo que nosotros terminamos y el proceso que hemos despertado es el único que continúa ejecutando dentro del monitor.

Las variables condición forman parte de la implementación del monitor y se ocupan de soltar el mutex mientras duermen. Dependiendo de la implementación, lo normal es que cuando un proceso despierta en un `signal`, el que lo despierta le ceda el mutex.

En otras ocasiones (lamentablemente) el proceso que despierta compite por el mutex del monitor con todos los demás, lo que quiere decir que otro proceso podría ganarle y hacer que la condición de nuevo sea falta. Eso implica que el código en este caso debería ser

```
while(nitems == 0) {
    wait(hayitems);
}
```

y no

```
if(nitems == 0) {
    wait(hayitems);
}
```

puesto que cuando despertamos es posible que tengamos que volver a dormir si otro se nos adelante antes de que podamos adquirir el mutex del monitor. Java es un notable ejemplo de esta **pésima** implementación de monitores.

En UNIX disponemos de llamadas en la librería *pthread(3)* para usar variables condición. Dichas llamadas utilizan tanto una variable condición como un mutex que tenemos que crear para que lo use el monitor. Esto es, podemos implementar monitores pero los estamos programando casi a mano.

Sigue el código de nuestro productor consumidor utilizando los mutex y variables condición de *pthread(3)*, en lugar de utilizar nuestros semáforos como en la implementación que vimos antes.

[pcmon.c]:

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

enum {QSIZE = 4};

typedef struct Queue Queue;
struct Queue {
    pthread_mutex_t mutex;
    pthread_cond_t notempty, notfull;
    char buf[QSIZE];
    int hd, tl, sz;
};


static int
qinit(Queue *q)
{
    memset(q, 0, sizeof *q);
    if (pthread_mutex_init(&q->mutex, NULL) != 0) {
        return -1;
    }
    if (pthread_cond_init(&q->notfull, NULL) < 0) {
        pthread_mutex_destroy(&q->mutex);
        return -1;
    }
    if (pthread_cond_init(&q->notempty, NULL) < 0) {
        pthread_cond_destroy(&q->notfull);
        pthread_mutex_destroy(&q->mutex);
        return -1;
    }
    return 0;
}
```

```
static void
qterm(Queue *q)
{
    pthread_cond_destroy(&q->notfull);
    pthread_cond_destroy(&q->notempty);
    pthread_mutex_destroy(&q->mutex);
    q->hd = q->tl = 0;
}


static void
qput(Queue *q, int c)
{
    if (pthread_mutex_lock(&q->mutex) != 0) {
        err(1, "mutex");
    }

    while (q->sz == QSIZE) {
        if (pthread_cond_wait(&q->notfull, &q->mutex) != 0) {
            err(1, "cond wait");
        }
    }

    q->buf[q->tl] = c;
    q->tl = (q->tl+1)%QSIZE;
    q->sz++;

    if (pthread_cond_signal(&q->notempty) != 0) {
        err(1, "cond signal");
    }
    if (pthread_mutex_unlock(&q->mutex) != 0) {
        err(1, "mutex");
    }
}
```

```
static int
qget(Queue *q)
{
    int c;

    if (pthread_mutex_lock(&q->mutex) != 0) {
        err(1, "mutex");
    }

    while (q->sz == 0) {
        if (pthread_cond_wait(&q->notempty, &q->mutex) != 0) {
            err(1, "cond wait");
        }
    }

    c = q->buf[q->hd];
    q->hd = (q->hd+1)%QSIZE;
    q->sz--;

    if (pthread_cond_signal(&q->notfull) != 0) {
        err(1, "cond signal");
    }
    if (pthread_mutex_unlock(&q->mutex) != 0) {
        err(1, "mutex");
    }
    return c;
}


static Queue q;


static void*
tput(void *a)
{
    char *s;
    int i;

    s = a;
    for (i = 0; i < 10; i++) {
        qput(&q, s[0]);
    }
    return NULL;
}
```

```
static void*
tget(void *a)
{
    int c;
    char buf[1];

    for (;;) {
        c = qget(&q);
        if (c == 0) {
            break;
        }
        if (c < 0) {
            err(1, "qget");
        }
        buf[0] = c;
        if (write(1, buf, 1) != 1) {
            err(1, "write");
        }
    }
    return NULL;
}


int
main(int argc, char* argv[])
{
    pthread_t p1, p2, g;
    void *sts;

    if (qinit(&q) < 0) {
        err(1, "qinit");
    }
    if (pthread_create(&p1, NULL, tput, "a") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&p2, NULL, tput, "b") != 0) {
        err(1, "thread");
    }
    if (pthread_create(&g, NULL, tget, NULL) != 0) {
        err(1, "thread");
    }
    pthread_join(p1, &sts);
    pthread_join(p2, &sts);
    qput(&q, 0);
    pthread_join(g, &sts);
    write(1, "\n", 1);
    qterm(&q);
    exit(0);
}
```

Si lo ejecutamos, veremos que funciona de un modo similar a nuestra última implementación.

```
unix$ pcmon
bababaaababaabbbaabb
unix$
```

Te resultará útil en este punto comparar la implementación con semáforos con la implementación con un monitor.