Introducción a Sistemas Operativos: Concurrencia

Clips xxx
Francisco J Ballesteros

1. 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`.

2. 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 1.

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

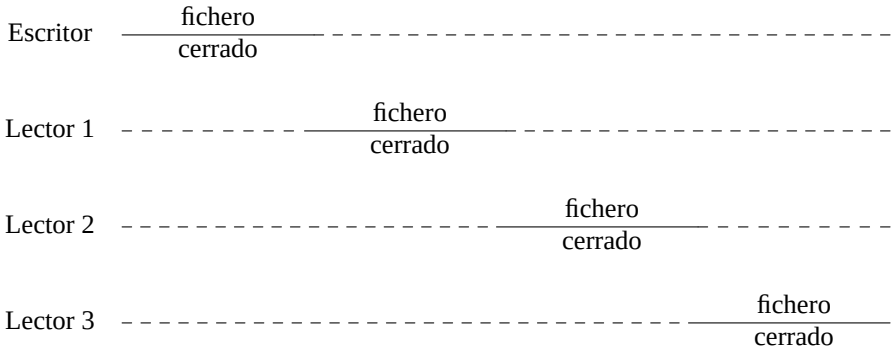

Figura 1: *Múltiples lectores y escritores de un fichero con un cierre exclusivo. Sólo puede ejecutar uno cada vez dentro de la región crítica.*

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 2. Como puede verse, los procesos han de esperar en menos y, conjuntamente, terminamos antes.

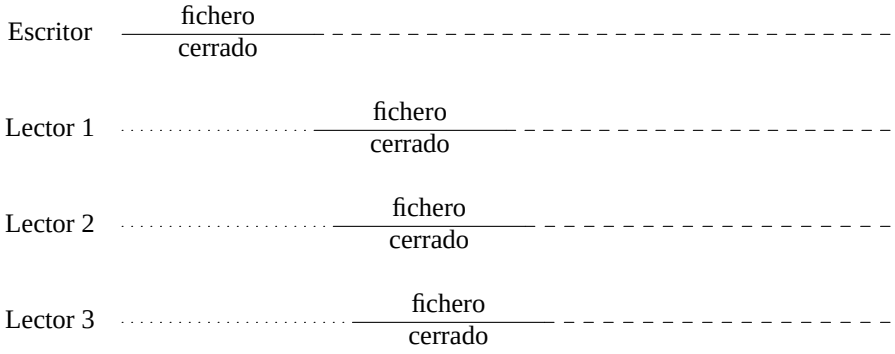

Figura 2: *Con un cierre de tipo lectores/escritores permitimos múltiples lectores concurrentes manteniendo la exclusión mutua de los escritores.*

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*).

3. 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.