Introducción a Sistemas Operativos: Concurrencia

Clips xxx
Francisco J Ballesteros

1. 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.

2. 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.

3. ¿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.