Introducción a Sistemas Operativos: Concurrencia

Clips xxx
Francisco J Ballesteros

1. 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*.

2. 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.