Introducción a Sistemas Operativos: Concurrencia

Clips xxx
Francisco J Ballesteros

1. 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.

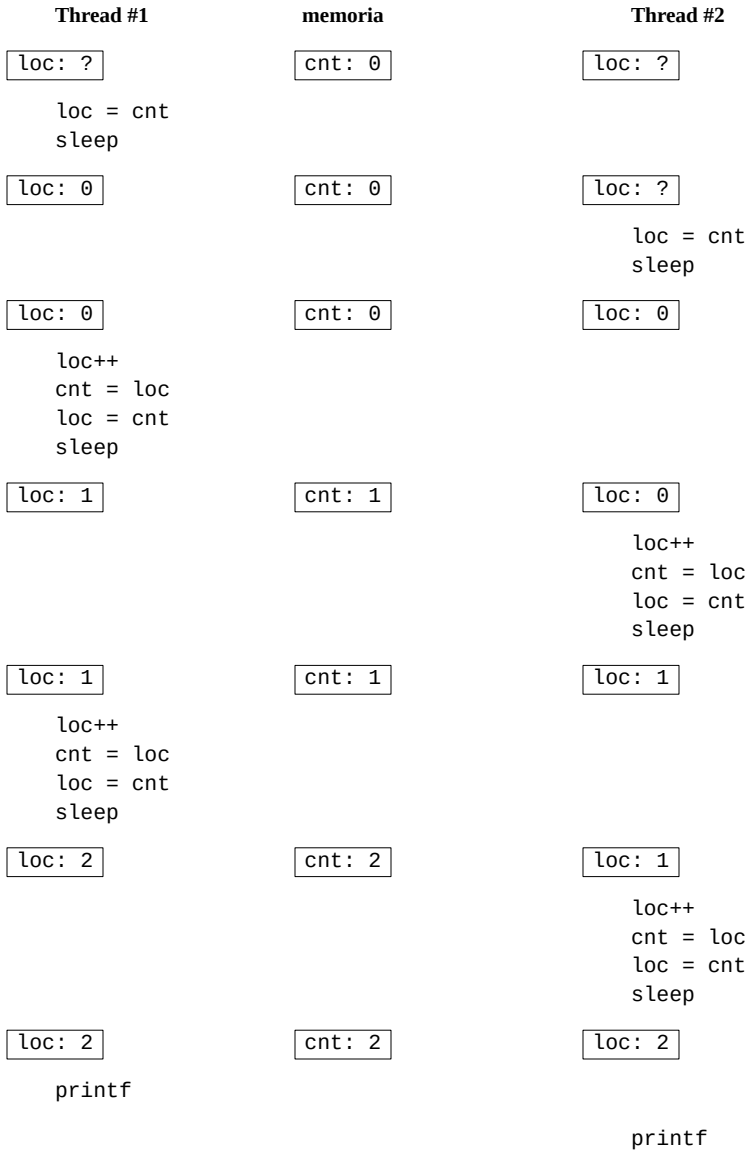

Figura 1: *Un entrelazado de sentencias en ambos procesos que da lugar a una condición de carrera en la última versión del programa.*

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.