

Carreras críticas en el problema productor-consumidor

CÉSAR POZA GONZÁLEZ, PABLO MOURIÑO LORENZO

Sistemas Operativos II

{cesar.poza.gonzalez, pablo.mourino.lorenzo}@rai.usc.es

I. INTRODUCCIÓN

El problema del productor-consumidor es un ejemplo clásico en la concurrencia y la sincronización de procesos. Se trata de una situación en la que un proceso productor genera datos y los almacena en un buffer compartido, mientras que un proceso consumidor extrae y procesa estos datos. Si no se gestiona adecuadamente el acceso al buffer, pueden producirse condiciones de carrera, donde la ejecución concurrente de ambos procesos altera el comportamiento esperado del programa.

Este informe presenta la implementación de un productor y un consumidor como procesos independientes que comparten un buffer de tipo LIFO con capacidad de 8 caracteres. Dado que emplearemos el lenguaje C, se utilizan llamadas a `sleep()` estratégicamente ubicadas para aumentar la probabilidad de que ocurran condiciones de carrera, permitiendo así su observación y análisis. Además, se emplea memoria compartida mediante `mmap()` para gestionar el acceso a los datos y se considera el uso de semáforos o señales para la sincronización.

Para la ejecución de ambos códigos, debemos compilar los archivos con `gcc`; `gcc -Wall -o prod prod.c` y `gcc -Wall -o cons cons.c`. Una vez tenemos los archivos ejecutables, abrimos dos terminales para ejecutar en cada una el código correspondiente. En este caso, no importa cuál se ejecuta antes, pero debemos pasar como parámetro el número de segundos que se desean en los `sleep()` que se encargan de alargar el tiempo de la región crítica. Por ejemplo, si queremos que el productor tenga un segundo, haremos `./prod 1`.

II. PROGRAMA PRODUCTOR

Para comenzar, primero programamos el programa que se encargará de realizar la función de productor. Este archivo llevará el nombre **prod.c**. Para realizar el código de esta parte, emplearemos el pseudocódigo proporcionado en las diapositivas del tema 1 de la asignatura, obteniendo un resultado similar al que vemos en la imagen 1.



```
while(TRUE) {
    item = produce_item();
    while(info->numElementos == N) {
    }

    insert_item(info, item);
    info->numElementos++;
    if(info->numElementos == 1) {
        kill(info->pidCons, SIGUSR1);
    }
}
```

Figura 1: pseudocódigo productor traducido a C

Básicamente, este trozo de código se ejecuta siempre que el `bool` de la condición del `while` sea verdadera. En caso de serlo, se genera un elemento con la función `produce_item()` el cual será insertado en el array compartido. Sin embargo, antes de insertarlo, se comprueba que el array no esté lleno; en caso de estarlo, el proceso realiza espera activa. En caso contrario, el elemento será introducido al array compartido y el contador del número de elementos, aumentará. Por último, si el buffer deja de estar vacío, se despierta al consumidor mediante una señal.

III. PROGRAMA CONSUMIDOR

Por el otro lado, debemos realizar el programa que tomará el rol de consumidor que, de

igual forma, lo obtenemos a partir del pseudocódigo mostrado en las transparencias del primer tema. Este archivo llevará el nombre de **cons.c**. Una vez traducimos el pseudocódigo a lenguaje C, obtenemos un código similar al de la figura 2.

```
while(boolean) {
    while(info->numElementos == 0) {
    }

    item = remove_item(info);

    info->numElementos--;

    if(info->numElementos == N - 1) {
        kill(info->pidProd, SIGUSR1);
    }

    consume_item(item, arrayLocal, &indexLocal);
}
```

Figura 2: pseudocódigo consumidor traducido a C

De forma contraria a cómo actúa el productor, el consumidor se encarga de eliminar el elemento más reciente que fue insertado en el array de tipo LIFO. Una vez se ejecuta el bucle, primero hace una comprobación de que el número de elementos no sea cero, en caso de serlo, realizará una espera activa, de la cual salirá en 2 casos:

1. Se actualiza numElemntos: puede que el proceso detecte la actualización de la variable compartida, a lo que salirá del bucle al no cumplirse la condición.
2. Recibe la señal: también puede ocurrir que, el productor, envíe la señal que hay dentro del condicional al final del bucle de su código, avisando así al consumidor de que el array ya no está vacío. Esto es posible debido a que las señales interrumpen cualquier operación bloqueante.

Una vez sale de la espera activa, la función `remove_item()` se encarga de eliminar el objeto más reciente, decrementando así el número de elementos al realizar esta acción. De forma similar a cómo se hace en el productor, si el buffer estaba lleno, pero ya se consumió un elemento, el consumidor avisa al productor mediante una señal para que este salga de la espera activa.

IV. MEMORIA COMPARTIDA

Como se mencionó en la introducción, este problema es posible si se ejecutan dos procesos los cuales hacen modificaciones sobre una variable compartida, siendo en este caso el array. Como ya sabemos de Sistemas Operativos I, dos procesos no comparten memoria, pero sí pueden modificar un mismo archivo. Teniendo conocimiento sobre esto, la compartición de memoria se puede conseguir usando diferentes funciones que nos proporciona C.

En nuestro caso, decidimos hacer una variable `info` de tipo `struct sharedData*` la cual contiene diferentes variables que compartirán los procesos. Para conseguir esto, primero crearemos un archivo y le hacemos lo necesario, es decir, abrirlo en el programa y truncarlo para ajustarle el tamaño. Una vez hecho esto, tal y como vemos en la figura 4, proyectamos el archivo en memoria.

```
typedef struct {
    char array[N];
    int numElementos;
    int pidProd, pidCons;
} sharedData;
sharedData* info;
```

Figura 3: Struct con las variables a compartir

```
info = (sharedData*)mmap(NULL, sizeof(sharedData),
    PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Figura 4: Proyección del archivo con `mmap`

De esta forma, le especificamos a la función `mmap()` que queremos proyectar el archivo almacenado en `fd` con un tamaño `sizeof(sharedData)`. Además, le especificamos que las páginas pueden ser escritas y leídas y que los cambios son visibles para otros procesos. Por último, es importante destacar el casteo a `sharedData*` que se realiza, ya que `mmap()` devuelve la dirección de la región de memoria compartida como un `void*`. Con el cast, lo que conseguimos es que, a pesar de

proyectar el archivo, lo que compartimos realmente es la estructura `sharedData`, usando el archivo como un respaldo", ya que quedará en él los valores finales de la memoria compartida.

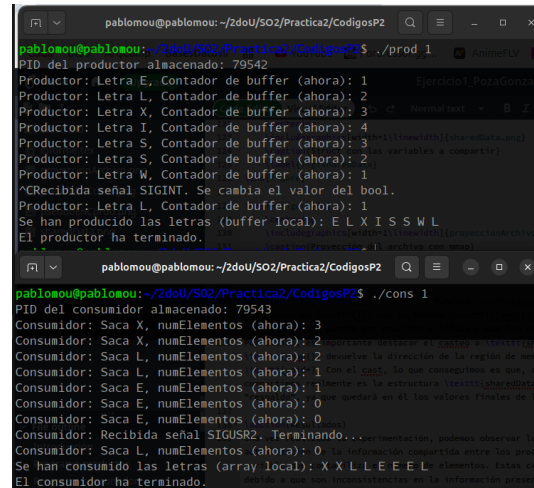
V. FORZADO DE CARRERAS CRÍTICAS

Por último, antes de realizar la ejecución, es importante conocer que podemos "forzar" las carreras críticas. Esto lo podemos conseguir si aumentamos el tiempo que un proceso se encuentra en la región crítica, por lo que, si posicionamos correctamente la función `sleep()`, conseguiremos aumentar significativamente la probabilidad de carrera crítica. En el caso del productor; el primer `sleep()` que colocamos, lo situamos entre la producción del elemento y la inserción en el array, produciendo así una pausa antes de que acceda al array compartido. El segundo, lo situaremos justo antes de modificar el valor de la variable `numElementos`, ya que, una vez inserta el item generado, al pausar antes de la actualización de la variable compartida, se puede dar el caso en el que el consumidor lea el array con el índice desactualizado, provocando inconsistencias. Por otro lado, en el código del consumidor, decidimos poner un `sleep()` en la misma posición que el segundo del productor, justo antes de la actualización de la variable, consiguiendo así el mismo efecto, pero en este caso, generándole la inconsistencia al productor.

VI. RESULTADOS

Una vez realizada la experimentación, podemos observar la presencia de carreras críticas en la actualización de la información compartida entre los procesos, es decir, en el array y la variable que contabiliza el número de elementos. Estas carreras críticas se pueden identificar debido a que son inconsistencias en la información presentada. Un ejemplo de ellas, se puede apreciar en la figura 5.

En esta ejecución, podemos ver como los items producidos, son {E, L, X, I, S, S, W, L}, mientras que los consumidos son {X, X, L, L, E, E, E, L}. Esto se debe a carreras críticas en la actualización de la variable que conta-



```
pablomou@pablomou: ~/2doU/SO2/Practica2/CodigosP2
PID del productor almacenado: 79542
Productor: Letra E, Contador de buffer (ahora): 1
Productor: Letra L, Contador de buffer (ahora): 2
Productor: Letra X, Contador de buffer (ahora): 3
Productor: Letra I, Contador de buffer (ahora): 4
Productor: Letra S, Contador de buffer (ahora): 5
Productor: Letra S, Contador de buffer (ahora): 6
Productor: Letra W, Contador de buffer (ahora): 7
^CRecibida señal SIGINT. Se cambia el valor del bool.
Productor: Letra L, Contador de buffer (ahora): 1
Se han producido las letras (buffer local): E L X I S S W L
El productor ha terminado.

pablomou@pablomou: ~/2doU/SO2/Practica2/CodigosP2
PID del consumidor almacenado: 79543
Consumidor: Saca X, numElementos (ahora): 3
Consumidor: Saca X, numElementos (ahora): 2
Consumidor: Saca L, numElementos (ahora): 2
Consumidor: Saca L, numElementos (ahora): 1
Consumidor: Saca E, numElementos (ahora): 1
Consumidor: Saca E, numElementos (ahora): 0
Consumidor: Saca E, numElementos (ahora): 0
Consumidor: Recibida señal SIGUSR2. Terminando...
Consumidor: Saca L, numElementos (ahora): 0
Se han consumido las letras (array local): X X L L E E E L
El consumidor ha terminado.
```

Figura 5: Ejecución de ambos programas

biliza el número de elementos, por ejemplo; cuando el consumidor comienza su ejecución, el array ya tiene 3 elementos, siendo el tope y el consumido 'X'. Una vez realiza esto, se dispone a consumir el siguiente elemento, es decir; 'L'. Sin embargo, en ese tiempo el productor modificó la variable incrementándola en uno, ya que introduce otro elemento más, por lo que el consumidor repetirá el elemento consumido.