

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, al cual controlarán los accesos mediante semáforos. Dado que emplearemos el lenguaje C, se utilizan llamadas a funciones como `sem_open()`, `sem_wait()` o `sem_post()` para realizar la interacción de los procesos con los semáforos que controlarán la aparición de carreras críticas. Además, se emplea memoria compartida mediante `mmap()` para gestionar el acceso a los datos y se considera el uso de señales para la sincronización.

Para la ejecución de ambos códigos, debemos compilar los archivos con `gcc`; `gcc -Wall -o prod_sem prod_sem.c -pthread` y `gcc -Wall -o cons_sem cons_sem.c -pthread`. Una vez tenemos los archivos ejecutables, abrimos dos terminales para ejecutar en cada una el código correspondiente. En este caso, deberemos ejecutar antes el productor que el consumidor. Esto debido a que es el encargado de producir los elementos, el que crea los semáforos que se usarán.

II. PROGRAMA PRODUCTOR

Para comenzar, primero programamos el programa que se encargará de realizar la función de productor. Este archivo llevará el nombre **prod_sem.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_sem.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

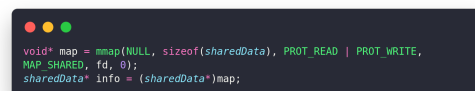
Como se mencionó en la introducción, este problema es posible si se ejecutan dos procesos los cuales hacen modificaciones sobre una variable compartida, siendo en este caso el array. Como ya sabemos de Sistemas Operativos I, dos procesos no comparten memoria, pero sí pueden modificar un mismo archivo. Teniendo conocimiento sobre esto, la comparación 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 ??, proyectamos el archivo en memoria.



```
typedef struct {
    char array[N];
    int numElementos;
} sharedData;
```

Figura 3: Struct con las variables a compartir



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

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 en la asignación de la variable `info`, 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. USO DE SEMÁFOROS

A diferencia del ejercicio anterior, en el cual buscábamos forzar las carreras críticas entre los procesos, en este experimento tratamos de evitarlas con el uso de semáforos. Para poder usarlos, deberemos incluir en nuestro programa la biblioteca `<semaphore.h>`. Con esta, crearemos los semáforos en el productor con la función `sem_open()`, la cual usará el consumidor para abrir estos semáforos recién creados. De esta forma, se crean 3 semáforos distintos; `mutex`, `vacías` y `llenas`, teniendo cada uno su propósito.

El semáforo `mutex` tiene la finalidad de sincronizar los procesos, siendo inicializado en el con `sem_open()` a valor 1, indicando que el buffer tiene el acceso permitido. Este semáforo es abierto y cerrado tanto por el productor como por el consumidor.

El semáforo `vacías` se encarga de controlar el número de espacios libres que quedan en el buffer. Además, es inicializado a `N`, ya que en un inicio, está completamente vacío. El productor se encarga de decrementarlo con `sem_wait()` cuando introduce un elemento al array compartido. Permitiendo así que el proceso productor se bloquee cuando el array esté lleno. Por otro lado, el consumidor se encarga de aumentar el semáforo, usando `sem_post()`, consiguiendo así que, si el array está lleno, cuando el consumidor elimine un elemento, el productor se desbloquee.

Por último, el semáforo `llenas` es creado e inicializado a 0, ya que lleva la cuenta de las posiciones llenas en el array. Tal como hemos explicado el semáforo `vacías`, sobre el semáforo `llenas` se hacen las mismas operaciones, pero con los roles intercambiados. El productor se encargará de incrementarlo, mientras que el consumidor lo decrementará.

array y la variable que contabiliza el número de elementos. Esta evasión de las carreras críticas se puede apreciar con una ejecución de los programas, en la cual podremos ver cómo los mensajes que van imprimiendo los programas, respetan el orden en que se introducen y eliminan los elementos del buffer compartido.

VI. RESULTADOS

Una vez realizada la experimentación, podemos observar la **no** presencia de carreras críticas en la actualización de la información compartida entre los procesos, es decir, en el