Sincronización de procesos con mutexes y variables de condición - Opcional 1

Pablo Mouriño Lorenzo, Diego Pérez Álvarez

Sistemas Operativos II
Grupo 04

Resumen

En este informe se analizará el problema del productor-consumidor con el objetivo de observar carreras críticas y analizar sus posibles soluciones usando mutexes y variables de condición para gestionar el acceso a las variables compartidas, utilizando procesos en lugar de hilos.

Palabras clave: carreras críticas, productor-consumidor, sincronización, mutexes

I. Introducción

Las carreras críticas son errores que se dan por acceder de forma no controlada a memoria compartida por varios procesos. Estos sucesos son muy indeseados e impredecibles, pero pueden evitarse con la sincronización adecuada.

En este trabajo nos centraremos en evitar estos sucesos aplicando una correcta sincronizacón en el acceso de los procesos a la estructura de datos compartida entre sí. Para llevar a cabo esto nos apoyaremos de los mutex y las variables de condición. Los mutex, gracias a su posibilidad de estar en solamente dos estados, nos permitirán gestionar la exclusión mútua sobre los recursos. Sin embargo, en este caso enfocaremos la solución al uso de procesos, en vez de hilos, lo que requerirá un mayor cuidado en la creación y gestion de variables compartidas entre ellos.

En este informe se incluirá un apartado sobre el programa, cómo ejecutarlo y su funcionamiento. También se incluirá un apartado de conclusiones en el que desarrollaremos los resultados obtenidos.

II. Programa

Esta cuestión del productor-consumidor se ha desarrollado en un único programa, escrito en C, de forma que se crean los procesos productores y consumidores a partir del programa. A cada productor se le deberá designar un archivo de texto del cual extraerá únicamente los carácteres alfanuméricos, colocándolos posteriormente en el buffer compartido, el cual tendrá un funcionamiento de cola FIFO. A continuación, los consumidores tomarán los carácteres del array y los colocarán en archivos de texto de salida, creados anteriormente por cada consumidor.

La condición de parada del programa será, para los productores; acabar el archivo de texto del que leen la información, indicando su final añadiendo un asterisco, y para los consumidores, terminar de leer el buffer compartido una vez que se alcanzó un número de asteriscos igual al de productores, indicando que todos estos terminaron.

Los costes generados por este programa son las anteriormente mencionadas regiones críticas. Estas regiones críticas se darán, tanto para productores como consumidores, en el acceso al buffer compartido y a las variables que usaremos como índices de la cola FIFO. Además, en el caso de los consumidores, como se debe llevar la cuenta del número de asteriscos que se han leído, la variable que los contabilice también supondrá zona potencial de regiones críticas. Con el uso de los mutex solucionaremos estos inconvenientes.

A. Ejecución

Para la ejecución de este programa es suficiente con compilarlo de la siguiente manera, enlazando la biblioteca `pthread.h` de ser necesario:

```
gcc -Wall -o opcional1 opcional1.c -lpthread
```

Es necesario pasarle 3 parámetros por línea de comandos para que se ejecute correctamente: `P` (número de procesos que actuarán como Productores), `C` (número de procesos que actuarán como Consumidores) y `N` (número de posiciones que tendrá el buffer). Si alguno de estos parámetros se omite o no se introduce correctamente, el programa no se ejecutará.

B. Funcionamiento

El programa comienza creando la región de memoria que será compartida entre los procesos la cual, a continuación, irá inicializando. Entre estas inicializaciones están los mutexes y variables de condición necesarios:

1. Un mutex para el acceso a la región crítica entre los productores y los consumidores, en este caso la estructura.
2. un mutex para controlar el acceso a la variable que seguirá el número de asteriscos.
3. Una variable de condición `condp` para bloquear los procesos Productores hasta que no se cumpla la condición de que hay una posición libre en el buffer para insertar un nuevo elemento.
4. Una variable de condición `condc` para bloquear los procesos Consumidores hasta que no se cumpla la condición de que hay alguna posición ocupada en el buffer para que consuma el hilo.
5. Una variable de condición `condBarrera` que se encargará de bloquear a los procesos al inicio del programa, mientras el usuario introduce la información solicitada.

En estas inicializaciones, dado que los mutexes están orientados para gestionar los hilos de un mismo proceso, deberemos tener en cuenta que lo que vamos a usar son subprocesos creados por el programa, no hilos. Para poder usar en este caso los mutex y variables deberemos crear dos variables; una para los mutex y otra para las condiciones, de tipo `pthread_mutexattr_t` y `pthread_condattr_t` respectivamente. Con estas, podremos especificar los atributos de los mutex y de las variables de condición, de forma que usaremos la función `pthread_mutexattr_setpshared()` para especificar que en ambos casos que se utilizarán sobre procesos. Debemos también usar la macro `PTHREAD_PROCESS_SHARED`.

Una vez que el main termina de crear todos los procesos necesarios, esperará hasta que terminen de ejecutarse todos los productores y los consumidores mediante un `pthread_join`.

Cuando finaliza el programa, liberamos todos los recursos reservados.

C. Procesos Productores

En los procesos que asumen la función de Productor, el funcionamiento es el siguiente. Después de pedir al usuario el nombre del archivo que tendrá que leer y su retardo máximo de espera (usando los mutexes necesarios), espera usando una función que simula la barrera que se emplea en el caso de los hilos para que todos los procesos estén listos.

Una vez que se cumple esta condición, cada productor recorre todo su archivo de caracter en caracter, comprobando si es de tipo alfanumérico. Si lo es, intenta obtener el mutex para acceder a la región crítica, y una vez dentro de ella usa la variable de condición `condp` para comprobar que haya alguna posición libre en el buffer en la que insertar el elemento producido. De no ser así, se bloqueará hasta que se cumpla usando `pthread_cond_wait`. A continuacion, inserta el elemento en la posición final de la cola y envía la señal `condc` a alguno de los consumidores bloqueados por esta variable de condición para que, si estaban bloqueados, puedan continuar con su ejecución ya que ya hay un elemento que consumir. Por último, libera el mutex.

Una vez que termina de leer e insertar en el buffer todo el archivo, es necesario repetir todo este proceso otra vez para introducir en el buffer un asterisco, que señalizará que un Productor ha terminado.

D. Procesos consumidores

Los procesos consumidores, como se ha mencionado anteriormente, se encargan de escribir los elementos del buffer compartido en archivos de texto. Para esto, lo primero que hacen es crear dichos archivos, identificando cada uno en base al id que tiene cada consumidor. Además, cada uno solicitará un entero, el cual tendrá la función de ser el máximo de un rango de tiempo que simulará el tiempo de producción. Todo esto se verá sincronizado gracias al uso de una barrera de tipo `pthread_barrier_t`.

Una vez entramos en el bucle, se comienza bloqueando el mutex, lo que nos permitirá comprobar el número de elementos en el buffer. De esta forma, mientras el número de elementos es nulo y no está finalizado el trabajo de los consumidores (no procesaron todos los asteriscos), los consumidores esperarán a que alguna de las dos condiciones no se cumpla. Al salir de este bucle, se vuelve a comprobar las mismas condiciones, de forma que, si había algún consumidor esperando y otro terminó todo el trabajo, cuando salgan del bucle anterior no ejecutarán el código restante. Una vez pasadas estas condiciones, aprovechando el bloqueo inicial del mutex se consume el elemento del buffer más antiguo, respetando la estructura FIFO, lo que causará que el proceso salga de la memoria compartida y pueda desbloquear el mutex a la vez que avisa a los productores mediante las variables de condición. Ahora es turno de que el consumidor escriba el elemento consumido en su respectivo archivo de salida y, una vez hecho, evalúe si el elemento era un asterisco o no. En caso de no serlo, prosigue con la ejecución simulando un tiempo de consumo, pero si es un asterisco deberá actualizar el contador de asteriscos. Para ello, dado que es una variable a la que tendrán acceso todos los procesos, deberá bloquear el mutex y aumentar en uno el contador. A continuación, comprobará si el número de asteriscos coincide con el número de procesos productores para que, si es así, avise a todos los procesos consumidores de que el trabajo terminó usando `pthread_cond_broadcast(&condc)`, evitando que algún consumidor se quedase bloqueado. Para finalizar, desbloqueará el mutex simulará el tiempo de consumición mediante un `sleep()`. Al terminar su trabajo, cerrará el archivo y ejecutará su `pthread_exit()`.

III. Conclusiones

En este informe se ha abordado una solución de las carreras críticas a través de la implementación del clásico problema del productor-consumidor mediante productores, mutex y variables de condición. A lo largo del desarrollo, demostramos cómo con una sincronización adecuada entre procesos, se puede solucionar estas inconsistencias que se pueden dar en el acceso a la memoria compartida, evitando los problemas indeseados que estas disputas entre procesos pueden ocasionar.

Este trabajo nos demuestra la importancia de los métodos de sincronización entre procesos para evitar estos comportamientos fatídicos que llevan a un resultado erróneo e indeseable.