

Sincronización de procesos con mutexes y variables de condición I

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.

Palabras clave: carreras críticas, productor-consumidor, hilos, 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 sincronización en el acceso de los hilos a las variables compartidas 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 mutua sobre los recursos.

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 hilos productores y consumidores a partir del programa. A cada productor se le deberá designar un archivo de texto del cual extraerá únicamente los caracteres alfanuméricos, colocándolos posteriormente en el buffer compartido, el cual tendrá un funcionamiento de cola FIFO. A continuación, los consumidores tomarán los caracteres 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 prod_cons prod_cons.c -lpthread
```

Es necesario pasarle 3 parámetros por línea de comandos para que se ejecute correctamente: P (número de hilos que actuarán como Productores), C (número de hilos 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 e inicializando 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 el buffer compartido.
2. Una variable de condición `condp` para bloquear los hilos Productores hasta que no se cumpla la condición de que hay una posición libre en el buffer para insertar un nuevo elemento.
3. Una variable de condición `condc` para bloquear los hilos Consumidores hasta que no se cumpla la condición de que hay alguna posición ocupada en el buffer para que consuma el hilo.

Además, como se requiere pedir al usuario que introduzca los valores de los retardos para cada uno de los procesos productores y consumidores (para poder visualizar mejor el avance de la ejecución) y los nombres de los archivos de entrada para los productores, necesitamos un mutex adicional para acceder al recurso compartido de la entrada por línea de comandos (ya que solo puede realizarse un `scanf` a la vez sin que se produzcan errores). Como solo se puede comenzar la ejecución del propio problema del productor-consumidor una vez estén todos los hilos correctamente inicializados con su archivo y retardos, es necesario crear también una barrera para que todos los hilos esperen hasta que el último termine de inicializarse antes de continuar con su ejecución normal.

Una vez que el `main` termina de crear todos los hilos 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. Hilos Productores

En los hilos 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 barrera hasta que todos los hilos 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 continuación, 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 señalará que un Productor ha terminado.

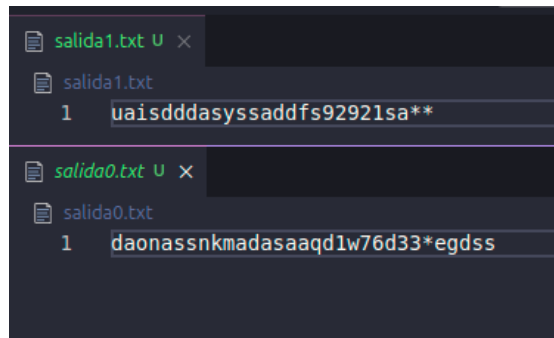
D. Hilos consumidores

Los hilos 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 hilo 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 hilos, deberá bloquear el mutex y aumentar en uno el contador. A continuación, comprobará si el número de asteriscos coincide con el número de hilos productores para que, si es así, avise a todos los hilos consumidores de que el trabajo terminó usando `pthread_cond_broadcast(&condc)`, evitando que algún consumidor se quedase bloqueado. Para finalizar, desbloqueará el mutex simulando el tiempo de consumición mediante un `sleep()`. Al terminar su trabajo, cerrará el archivo y ejecutará su `pthread_exit()`.

III. EJEMPLO DE EJECUCIÓN

Una vez explicado el funcionamiento del programa, podremos ejecutarlo para ver el resultado que producirá. Para esto, una vez compilado el código, lo ejecutaremos, en este caso, con `./prod_cons 3 2 8`, de forma que en esta ejecución tendremos 3 productores, 2 consumidores y un buffer con 8 espacios. Al inicio del programa, como sabemos, introduciremos la información solicitada por los productores y consumidores, que serán los archivos de entrada para los



```
salida1.txt U x
salida1.txt
1 uaisdddasyssaddfs92921sa**

salida0.txt U x
salida0.txt
1 daonassnkmadasaaqd1w76d33*egdss
```

Figura 1: Resultado de la ejecución

productores y los retardos para ambos.

El resultado de esta ejecución lo podemos ver en la figura 1. En esta, observamos como hay dos archivos de salida; `salida0.txt` y `salida1.txt`, que equivalen al número de consumidores especificado. Además, como se especificaba en el enunciado, los tres productores han escrito su respectivo asterisco, indicando el final de la lectura del archivo que se le indicó.

IV. 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 hilos, mutex y variables de condición. A lo largo del desarrollo, se ha demostrado cómo la sincronización adecuada entre hilos puede solucionar estas inconsistencias que se pueden dar en el acceso a la memoria compartida, evitando las inconsistencias y los problemas indeseados que estas disputas entre hilos pueden ocasionar.

Se ha implementado un programa en C que simula la interacción entre un productor y un consumidor, utilizando un buffer compartido. A través de la adición de mutex y variables de condición, se ha resuelto el problema de que ocurran carreras críticas.

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