Practica 4: Sincronización de procesos con mutexes

Manuel José Corral Pazos, Grupo 1 (Jueves)

Notas previas: En esta práctica, se ha creado un único código para el ejercicio propuesto, puesto que los consumidores y productores se ejecutan en hilos. Además, se utiliza un archivo de texto llamado 1txt.txt como apoyo para el ejercicio. En este archivo se guardará una cadena de números. Estos números serán la representación del buffer en el que los productores y consumidores realizarán sus respectivos trabajos. Para la realización de la práctica, se tomará la siguiente notación:

- 0 para los elementos que aún no hayan sido producidos ni consumidos
- 1 para los elementos producidos y que están presentes en el buffer
- 2 en las posiciones en las que un elemento haya sido consumido y todavía no se haya introducido ninguno por parte de los productores.

Código

En primer lugar, se incluyen las librerías necesarias y se crean las variables globales.
Entre ellas, encontramos la variable mutex, utilizada por los productores y consumidores para indicar que están entrando en la región crítica y evitar que otro hilo entre. También encontramos las variables condicionales lleno y vacío, que indicarán respectivamente si el buffer está completo de objetos producidos o si no contiene ningún objeto. Las variables elementos, pos, y texto, son la cantidad de elementos que hay actualmente en el buffer, la posición del último elemento introducido, y la representación del buffer sobre el que se produce/consume.

```
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>

#define N 10 //tamaño del buffer
pthread_mutex_t mutex;
pthread_cond_t lleno, vacio;
int elementos=0;
int pos=0;
char *texto;
```

Lo primero que se encuentra en el main es la inicialización de las primeras variables creadas, el mutex y las variables condicionales lleno y vacío. Dichas inicializaciones tienen sus respectivas comprobaciones para salir en caso de error.

```
int main (int argc, char** argv){
    //antes de nada, creamos el mutex y las
ctivas comprobaciones
    if(pthread_mutex_init(&mutex,NULL)!=0){
        return -1;
    }
    if(pthread_cond_init(&lleno,NULL)!=0){
        return -1;
    }
    if(pthread_cond_init(&vacio,NULL)!=0){
        return -1;
    }
```

Tras esto, se cumple con el primer requisito de la práctica, la realización de la misma para números arbitrarios menores que 5 hilos para productores y menores que 4 hilos para consumidores.

```
    srand(time(NULL));
 productores
    int ncons=0;
    int nprod=0;
    nprod=rand()%5+1;
s) para la cantidad de
    ncons=rand()%4+1;
```

Estos números son utilizados en la creación del vector de hilos, tanto de productores como de consumidores. A continuación, se muestra una captura de la creación de hilos de productores, a los cuales se le asigna la función “producir”. La creación de consumidores es análoga, pero se utiliza la función “consumir”.

```
pthread_t productores[nprod]; //vector de hilos de productores
int comprobacion=0; //esta variable se utiliza para comprobar que se crean los hilos correctamente
for(int i=0; i<nprod; i++){ //creación de nprod productores
    if((comprobacion=pthread_create(&productores[i], NULL, producir, (void *)i))!=0){
       //se crean los hilos de producción y se le asigna la función de producir
        return -1; //en caso de que la función falle, se aborta el programa
    }
}
```

En dichas funciones, se encuentra un bucle infinito, en el cual se encuentra primeramente un sleep con una cantidad aleatoria entre 0 y 4, cumpliendo así uno de los requisitos de la práctica. Este sleep se encuentra antes de la región crítica, y su función es la de forzar situaciones en las que tanto los productores como los consumidores se bloqueen por las variables de condición.

```
double dormir=0;
dormir=drand48()*4.0;
sleep(dormir);
```

Tras el sleep, está el lock de la variable mutex, indicando que se entra en la región crítica y que la función de, en este caso producir, va a comenzar. Tras entrar en la región crítica, se imprime el contenido del buffer antes de producir un elemento. En caso de que el número de elementos sea igual al tamaño total del buffer, se utiliza la función wait sobre la variable condicional lleno, indicando así que el buffer está lleno y por tanto el productor se bloquea hasta que algún consumidor utilice la función signal sobre la variable condicional lleno.
Para completar otro de los objetivos de la práctica, se encuentra un bucle que “mueve” todos los elementos una posición a la derecha, y la última la desecha. De esta forma, se simula el funcionamiento de una cola en el buffer. Esto se hace porque se buscará que el buffer funcione como una estructura FIFO, de forma que el objeto que lleve más tiempo sea el primero en ser consumido, y al hacer el desplazamiento de objetos, ayuda a que el funcionamiento sea más intuitivo tanto visual como funcionalmente. Tras esto, se “produce” un objeto, poniendo la primera posición del buffer a 1 y aumentando la cantidad de elementos. Además, la posición del primer elemento introducido se incrementa, puesto que se han movido todos los elementos una posición “a la derecha”.
A continuación, se comprueba la cantidad de elementos presentes en el buffer tras la producción. Si sólo hay un elemento, significará que el buffer estaba vacío antes de producir, por lo que los consumidores estarán eventualmente bloqueados. Por ello, se utiliza la función signal sobre “vacío”, la variable de condición que bloquea los consumidores, para hacer que se desbloqueen ya que hay objetos producidos dispuestos a ser consumidos. Finalmente, se imprime el buffer después de la producción y se desbloquea el mutex, indicando que se sale de la región crítica y la operación está finalizada.

```
pthread_mutex_lock(&mutex);
printf("\n\tP%d Buffer antiguo: ", (int)tid);
for(int i=0; i<N; i++){
    printf("%c ", texto[i]);
}
while(elementos==N){
    pthread_cond_wait(&lleno, &mutex);
}
for(int i=N; i>0; i--){
    texto[i]=texto[i-1]; //se mueven todos los objetos una posicion a la derecha
}
texto[0]='1';
elementos++;
if(pos<N){
    pos=(pos+1);
}
//pos=(pos+1)%N;
if(elementos==1){
    pthread_cond_signal(&vacio);
}
printf("\n\tP%d Buffer nuevo:   ", (int)tid); //se imprime de nuevo el buffer tras la consumición
for(int i=0; i<N; i++){
    printf("%c ", texto[i]);
}
pthread_mutex_unlock(&mutex);
```

Por otro lado, la función de consumir presenta un comportamiento similar al principio, con el sleep y el bloqueo de la variable mutex al entrar a la región crítica. Se imprime el contenido del buffer antes de producir y se comprueba que el buffer no esté vacío. En caso de estarlo, se ejecuta wait sobre la variable “vacío”, para bloquearse y esperar a que haya algún objeto disponible, en cuyo caso el productor desbloqueará la variable.

Para saber qué objeto hay que consumir, basta con seguir la lógica de las variables y consumir el objeto que se encuentra en la posición pos-1. Por tanto, se pone un 2 en dicha posición. Tras esto, se disminuyen las variables pos y elementos, puesto que se ha consumido un elemento, concretamente el primero (en orden temporal) de los producidos. Finaliza de forma análoga a la función de producir.

```
double dormir=0;
dormir=drand48()*4.0;
sleep(dormir);
pthread_mutex_lock(&mutex);
printf("\n\tC%d Buffer antiguo: ", (int) tid);
for(int i=0; i<N; i++){
    printf("%c ", texto[i]);
}
while(elementos==0){
    pthread_cond_wait(&vacio, &mutex);
}
texto[pos-1]='2';
if(pos>0){
    pos=pos-1;
}
//pos=(pos-1)%N;
elementos--;
if(elementos==(N-1)){
    pthread_cond_signal(&lleno);
}
printf("\n\tC%d Buffer nuevo:   ", (int) tid); //se imprime de nuevo el buffer tras la consumición
for(int i=0; i<N; i++){
    printf("%c ", texto[i]);
}
//printf("\thay %d elementos", elementos);
pthread_mutex_unlock(&mutex);
```

Ejecución

En este apartado, se ejecutará el código varias veces, puesto que se está tratando con un número aleatorio de consumidores y productores. De esta forma, se contemplarán varios casos.

En primer lugar, se ejecuta el programa y el número de productores resulta ser 3 y el de consumidores 4. En primera instancia, se puede ver el funcionamiento de los productores 0 y 1 produciendo objetos. Aunque no se aprecie visualmente, a medida que se producen elementos, el resto se mueven una posición “hacia la derecha”, debido al bucle introducido en la función de producir. Una vez que empiezan a trabajar los consumidores, en este caso el consumidor 0, se consume el objeto más a la derecha de la cola, que resulta ser el primero en producirse puesto que se han ido moviendo los objetos una posición en cada producción. Tras esto, ya se puede apreciar la dinámica FIFO del buffer, puesto que al producir un objeto, el 2 que representa el objeto consumido se mueve una posición a la derecha.

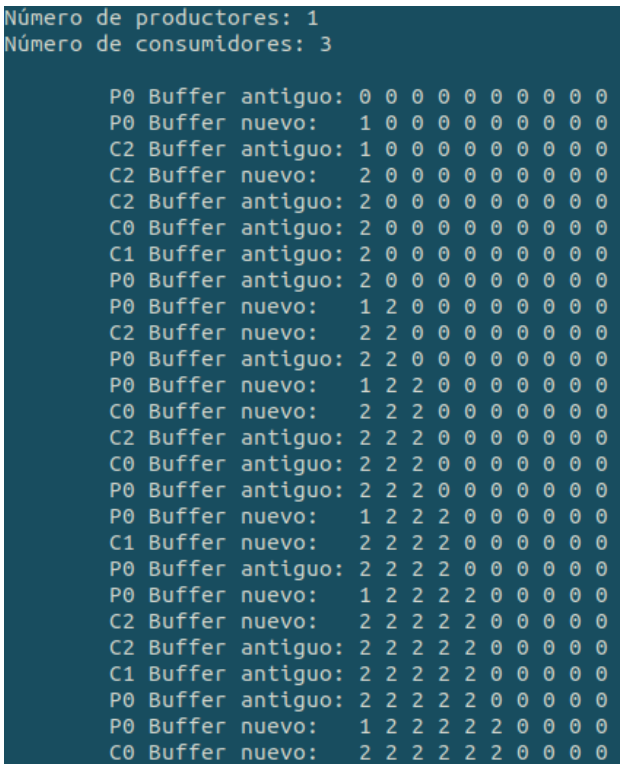
```
Número de productores: 3
Número de consumidores: 4

        P0 Buffer antiguo: 0 0 0 0 0 0 0 0 0 0
        P0 Buffer nuevo:   1 0 0 0 0 0 0 0 0 0
        P0 Buffer antiguo: 1 0 0 0 0 0 0 0 0 0
        P0 Buffer nuevo:   1 1 0 0 0 0 0 0 0 0
        P1 Buffer antiguo: 1 1 0 0 0 0 0 0 0 0
        P1 Buffer nuevo:   1 1 1 0 0 0 0 0 0 0
        P0 Buffer antiguo: 1 1 1 0 0 0 0 0 0 0
        P0 Buffer nuevo:   1 1 1 1 0 0 0 0 0 0
        P0 Buffer antiguo: 1 1 1 1 0 0 0 0 0 0
        P0 Buffer nuevo:   1 1 1 1 1 0 0 0 0 0
        C0 Buffer antiguo: 1 1 1 1 1 0 0 0 0 0
        C0 Buffer nuevo:   1 1 1 1 2 0 0 0 0 0
        P2 Buffer antiguo: 1 1 1 1 2 0 0 0 0 0
        P2 Buffer nuevo:   1 1 1 1 1 2 0 0 0 0
        P1 Buffer antiguo: 1 1 1 1 1 2 0 0 0 0
        P1 Buffer nuevo:   1 1 1 1 1 1 2 0 0 0
        P0 Buffer antiguo: 1 1 1 1 1 1 2 0 0 0
        P0 Buffer nuevo:   1 1 1 1 1 1 1 2 0 0
        P0 Buffer antiguo: 1 1 1 1 1 1 1 2 0 0
        P0 Buffer nuevo:   1 1 1 1 1 1 1 1 2 0
        C1 Buffer antiguo: 1 1 1 1 1 1 1 1 2 0
        C1 Buffer nuevo:   1 1 1 1 1 1 1 2 2 0
        C1 Buffer antiguo: 1 1 1 1 1 1 1 2 2 0
        C1 Buffer nuevo:   1 1 1 1 1 1 2 2 2 0
        C0 Buffer antiguo: 1 1 1 1 1 1 2 2 2 0
        C0 Buffer nuevo:   1 1 1 1 1 2 2 2 2 0
        C2 Buffer antiguo: 1 1 1 1 1 2 2 2 2 0
        C2 Buffer nuevo:   1 1 1 1 2 2 2 2 2 0
```

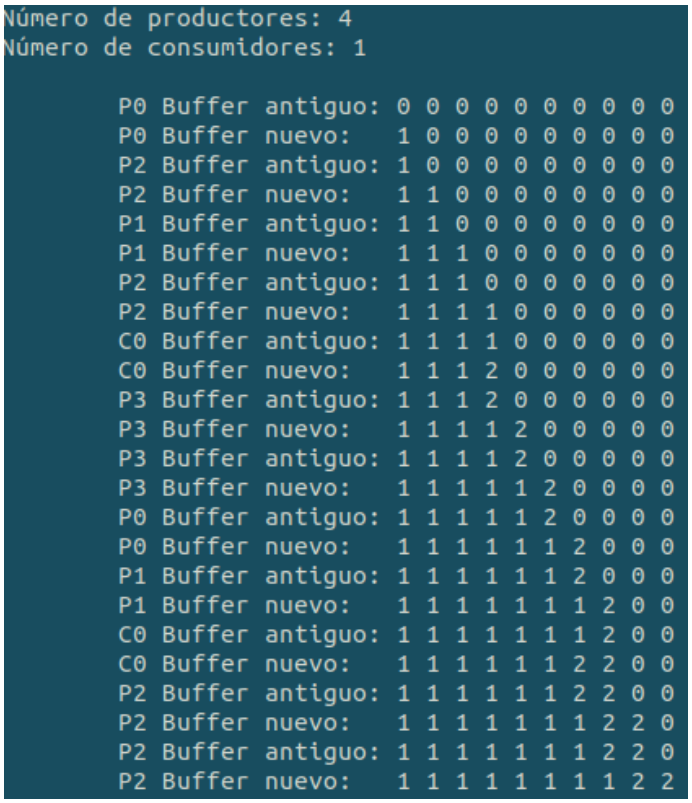
Durante esta ejecución, se llegó a una situación en la que el buffer estaba completamente vacío porque todos los objetos habían sido consumidos. En primer lugar, el productor 2 introdujo un objeto y el consumidor 0 lo consumió. Tras esto, el consumidor 3 trató de consumir, pero se encontró con un buffer vacío. Por tanto, se bloqueó y dejó la región crítica disponible para el siguiente hilo. Dicho siguiente hilo resultó ser también el consumidor 0, el cual también se bloqueó porque no había ningún objeto, dejando paso al siguiente hilo, el cual también fue un consumidor. Tras esto, actuó el productor 2 e introdujo un elemento, haciendo que finalmente se pueda consumir algo del buffer. Luego, se encuentra la impresión del buffer nuevo por parte del consumidor 3 y no del buffer antiguo. Esto es porque el consumidor 3 estaba bloqueado anteriormente, esperando a que hubiese algún elemento a consumir. En este caso, el consumidor 3 consiguió entrar a la región crítica y se desbloqueó, consumiendo el objeto.
Algo similar se puede ver al final de la imagen, cuando el consumidor 1 realiza un comportamiento similar.

```
P2 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
P2 Buffer nuevo:   1 2 2 2 2 2 2 2 2 2
C0 Buffer antiguo: 1 2 2 2 2 2 2 2 2 2
C0 Buffer nuevo:   2 2 2 2 2 2 2 2 2 2
C3 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
C0 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
C1 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
P2 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
P2 Buffer nuevo:   1 2 2 2 2 2 2 2 2 2
C3 Buffer nuevo:   2 2 2 2 2 2 2 2 2 2
P1 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
P1 Buffer nuevo:   1 2 2 2 2 2 2 2 2 2
C2 Buffer antiguo: 1 2 2 2 2 2 2 2 2 2
C2 Buffer nuevo:   2 2 2 2 2 2 2 2 2 2
P2 Buffer antiguo: 2 2 2 2 2 2 2 2 2 2
P2 Buffer nuevo:   1 2 2 2 2 2 2 2 2 2
P2 Buffer antiguo: 1 2 2 2 2 2 2 2 2 2
P2 Buffer nuevo:   1 1 2 2 2 2 2 2 2 2
C1 Buffer nuevo:   1 2 2 2 2 2 2 2 2 2
```

En una segunda ejecución del programa, se obtiene un caso con 1 productor y 3 consumidores. El programa sigue el comportamiento intuitivo de que el buffer va a estar vacío la mayoría del tiempo, puesto que hay un único hilo produciendo y tres consumiendo. De hecho, tras producir un elemento y ser consumido, llegan a bloquearse los tres hilos consumidores por falta de objetos en el buffer, hasta que llega el productor e introduce uno. Esto hace que uno de los consumidores se desbloquee y lo consuma. Poco a poco y a medida que el productor va introduciendo objetos, los hilos consumidores anteriormente bloqueados se van desbloqueando.



Por último, se hace una ejecución en la que la cantidad de productores es 4 y la de consumidores es 1. Al igual que en la ejecución anterior, en esta se sigue la idea intuitiva de que el buffer se llenará de forma muy rápida y que pasará la mayoría del tiempo lleno.



Además, llega el momento en el que el buffer está completamente lleno y el que entra en la región crítica resulta ser un productor. Por ejemplo, en la siguiente imagen el productor 0 intenta producir después de que el productor 3 haya llenado el buffer. Por esta razón, se bloquea después de mostrar el buffer antiguo, sin llegar a modificarlo. La próxima vez que aparece dicho productor 0, simplemente muestra el buffer nuevo tras modificarlo, saliendo del bloqueo en el que se encontraba. En la imagen se puede ver este comportamiento con varios consumidores en varias situaciones distintas. Por ejemplo, cuando tanto el productor 2 entra en la región crítica, se encuentra el buffer lleno y se bloquea, y justo después entra el productor 3 y también se bloquea por la misma razón.