Pŕactica 4: Sincronización de procesos con mutexes

Pablo Rodríguez Fernández

Abril 2020

I. Contenido del documento

En este documento , tras una breve introducción al uso de mutexes y variables de condición como herramienta de sincronización de procesos, se explicará como está realizado y las conclusiones obtenidas del ejercicio propuesto,el cuál trata una solución al problema del productor-consumidor empleando las herramientas anteriores.

II. Mutexes y variables de condición: Una breve introducción

Un mutex es una variable que toma solo dos valores, 0 o 1, regida por dos operaciones básicas:

- **Lock**: Si el mutex se encuentra abierto (es decir, si su valor es 0), el proceso continúa y pone el mutex a 1 (es decir, lo cierra). Si por la contra, el mutex es 1 (está cerrado), el hilo se bloquea hasta que este habra.

- **Unlock**: Abre un mutex (es decir, pone su valor a 0). En caso de que haya varios hilos se hallen bloqueados esperando a obtener el acceso al mutex, se selecciona uno de ellos al azar para que lo adquiera.

Se puede observar que un mutex es, en cierta manera, una simplificación de un semáforo, ya que administra la exclusión mútua en cierto fragmento de código sin utilizar la habilidad de conteo de los semáforos (ya que solo pued e estar abierto o cerrado). Debido a que se implentan con una mayor facilidad y eficiencia, son especialmente útiles en paquetes de hilos, que es exáctamente como se va a utilizar en esta práctica (en la pŕactica anterior se utilizaban semáforos con procesos).

En combinación con los mutexes, se utiliza en esta práctica otro mecanismo de sincronización, las **variables de condición**. Estas permiten que los hilos se bloqueen debido a que cierta condición no se esté cumpliendo. Utilizando este mecanismo y mutexes se obtiene una herramienta de sincronización de gran potencia, como se verá en la explicación del ejercicio realizado.

III. Ejercicio

En esta praćtica, el ejercicio planteado consiste en resolver el problema del productor-consumidor utilizando mutexes y variables de condición. En particular, se debía implementar la solución propuesta por Tanenbaum, pero realizando la generalización para n productores y m consumidores, y un buffer de tamaño arbitrario fijado previamente.

Se comenzará explicando las partes más relevantes de la iniciación de variables y del main(), para comentar luego las funciones que realizan la tarea de productor y consumidor específicamente. [1]

En primer lugar,1 muestra la declaración del mutex (el cual controla el acceso a la región crítica y asegura la exclusión mútua) y las variables de condición utilizadas en el programa..

```
pthread_mutex_t mutex; //Declaracion del mutex
pthread_cond_t condConsumer; //Declaracion de la variable de condicion asociada al consumidor
pthread_cond_t condProducer; //Declaracion de la variable de condicion asociada al productor
```

Listing 1: *Declaración del mutex y variables de condición*

En 2 se puede observar la declaración del buffer, el cual se pide que sea una pila LIFO. Para eso se utilizarán tres variables auxiliares:

- **Cuenta**: Numero de posiciones ocupadas en el buffer
- **posConsumir**: Posición del buffer en la cual se encuentra el elemento que se debe consumir
- **posProducir**: Posición del buffer en la cual se debe introducir un elemento tras ser producido

El empleo de estas variables permite 'crear' una cola FIFO sin necesidad de reorganizar todos los elementos cuando se consume un elemento, ya que, en lugar de eso, las variables anteriores funcionan como 'punteros' (en el sentido de indicar donde se deben introducir y eliminar elementos). Además, como se verá más adelante, las modificaciones en estas variables se realizan dentro de la región protegida por mutexes, con lo cual no existirán carreras críticas a la hora de acceder a ellas.

```
int buffer[N]; //Declaracion del buffer de tamaño N
//Las siguientes variables permiten que el buffer funcione como una cola FIFO
int cuenta=0;//Indica el numero de elementos validos en el buffer (numero de posiciones ocupadas)
int posConsumir=0;//Indica la posicion en la cual se debe consumir
int posProducir=0;//Indica la posicion en la cual se debe producir
```

Listing 2: *Declaración de la cola LIFO*

Como último fragmento a destacar antes del main(), se muestra 3, el cual contiene una struct que se le pasa como parámetro a la función consumidor(). Su uso simplemente se debe a que para invocar estas funciones en hilos diferentes se utilizar pthread_create(), que permite pasar un único parámetro a la función que va a ejecutar un hilo , y en este caso es necesario pasar dos. [2]

```
//Struct que almacena el color y el numero de iteraciones asociadas a cada consumidor
struct consum_struct {
  char *color;
  int numeroIteraciones;
};
```

Listing 3: *Declaración de la struct*

Una vez se está en el main(), tras la lectura de parámetros y la comprobación de que son correctos (un requisito de la práctica es que el número de productores fuese menor que 5 y

[1] Para ver la implementación completa con todos los detalles comentada, se adjunta el código del ejercicio
[2] El empleo de colores es opcional, pero facilita mucho la interpretación de resultados, por eso se incluye

el de consumidores menor que 4), y la inicialización de las posiciones del buffer a valor -1 (que se interpreta como 'posición vacía' simplemente para facilitar la lectura de resultados), se encuentra el fragmento 4, que se describe a continuación.

En primer lugar, se crean dos arrays que se usarán para crear los n hilos productores y los m consumidores. Tras esto, se eliminan el mutex y las variables de condición, para posteriormente inicializarlas (el mutex a 0, y las variables de condición con las opciones por defecto). La razón de eliminar antes de crear es evitar que interfieran posibles mutexes o variables de condición que existiesen de otros programas o ejecuciones anteriores con el mismo nombre (al igual que se hace con los semáforos). De hecho, realizar esto está considerado una buena práctica de programación.

```
pthread_t productores[nPr]; //Array de productores
pthread_t consumidores[nCo]; //Array de consumidores
//Eliminamos los mutex antes de inicializarlos
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&condProducer);
pthread_cond_destroy(&condConsumer);
//Inicilizacion del mutex y de las variables de condicion, todos a valor 0
pthread_mutex_init(&mutex,0);
pthread_cond_init(&condProducer,0);
pthread_cond_init(&condConsumer,0);
```

Listing 4: *Inicialización de mutex y variables de condición*

Tras esto, se crean los hilos productores, cada uno ejecutando la función consumer(), a la cual se le pasa como argumento el número de iteraciones que debe realizar, como se puede ver en 5. El número de iteraciones de cada productor es una constante definida anteriormente, Tras

```
//Creacion de productores (y reparto de colores)
 for(i=0;i<nPr;i++){
   pthread_create(&productores[i],0,producer,colors[i]);
 }
```

Listing 5: *Creación de los productores*

esto deberían crearse los consumidores se una manera similar. Sin embargo, en la prácticas se nos pide asegurar que todos los item producidos sean consumidos. Por tanto, en lugar de tener un número de iteraciones fijo, las iteraciones que realizan los consumidores dependen de las que realicen los productores. A grandes rasgos, lo que se hace es repartir el total de iteraciones entre los consumidores y , en caso de que el reparto no sea exacto, asignar las restantes al primer consumidor que se crea. Debido a la restricción que se comentó anteriormente sobre el número de consumidores y productores del ejercicio, este número 'extra' de iteraciones será, en el peor de los casos, 3, de ahí que se decidiese no realizar repartos mas 'elaborados' (como repartir estas iteraciones extra entre todos los consumidores). Esto se puede observar en 6.

```
//Reparto de iteraciones y colores entre los diferentes consumidores
struct consum_struct color_iteraciones[nCo];//A cada consumidor se le pasa una struct
int iteracionesConsumidores = (MAX*nPr)/(float)nCo; //Division exacta:
//Las asignaciones al primer consumidor se hacen fuera pues realiza mas iteraciones que el resto
color_iteraciones[0].numeroIteraciones=iteracionesConsumidores+(MAX*nPr-iteracionesConsumidores*nCo);
color_iteraciones[0].color=colors[nPr];
for(i=1;i<nCo;i++){//Asignamos las iteraciones y el color al resto de consumidores
  color_iteraciones[i].numeroIteraciones=iteracionesConsumidores;
  color_iteraciones[i].color=colors[nPr+i];//Sumamos nPr para no repetir colores
}
```

Listing 6: *Reparto de iteraciones entre los consumidores y asignación de colores*

Tras esto, se crean los consumidores de manera análoga a 5, pero se pasa como parámetro *struct[i]* en lugar de solamente el color.

Para finalizar, esperamos a que terminen todos los hilos productores y consumidores (el orden en el que acaben es indiferente, se realiza el bucle simplemente porque no existe un equivalente a *wait()* para hilos, es necesario indicar un hilo específico al que se espera) y se destruye el mutex y las variables de condición, como se muestra en 7. Tras comentar el main(),

```
  //Esperamos a que terminen los consumidores y los productores
 for(i=0;i<nPr;i++){
   pthread_join(productores[i],0);
 }
 for(i=0;i<nCo;i++){
   pthread_join(consumidores[i],0);
 }
 //Eliminamos las variables de condicion y el mutex
 pthread_cond_destroy(&condConsumer);
 pthread_cond_destroy(&condProducer);
 pthread_mutex_destroy(&mutex);
```

Listing 7: *Reparto de iteraciones entre los consumidores y asignación de colores*

se comentarán las funciones producer() y consumer(), que son las que que ejecuta cada hilo para realizar el papel de productor y consumidor, respectivamente. De estas se omite la declaración de variables.

En 8 se puede ver el código de producer(). Inicialmente, se produce un item en la función

```
for(i=0;i<MAX;i++){
    item=produce_item();
    sleep(rand()%5);//Sleep aleatorio entre 0 y 4 segundos
    pthread_mutex_lock(&mutex); //El productor obtiene acceso exclusivo a la region critica
    //Esperamos a que se reciba la señal de que se puede producir
    while(cuenta>=N) pthread_cond_wait(&condProducer,&mutex);
    insert_item(item);
    ++cuenta;//Incrementamos el numero de elementos para consumir que contiene el buffer
    //Fijamos el color de este productor.
    //Lo hacemos dentro de la region de acceso exclusivo para que ningun otro proceso cambie el color
    printf("%s",color);
    //Imprimimos el buffer para que sea mas sencillo de interpretar el resultado
    printf("Productor: ");
    for(int j=0;j<N;j++){
        printf(" %d ",buffer[j]);
    }
    printf("\n");
    printf("                -Item producido: %d\n",item);
    pthread_cond_signal(&condConsumer);//Enviamos una señal al consumidor para despertarlo
    pthread_mutex_unlock(&mutex); //Salimos de la region critica
 }
 pthread_exit(0);
```

Listing 8: *Código de la función producer()*

produce_item() y posteriormente se ejecuta un lock del mutex. Si se consigue el mutex, se comprueba si el número de elementos producidos es mayor o igual que N, es decir, si el buffer está lleno. De ser así, se espera a que la variable de condición asociada al productor reciba una señal. Esto se debe ejecutar continuamente (dentro de un while) ya que las variables de condición **no tienen memoria**: si llega una señal que no se estaba esperando, se pierde. Tras esto, se inserta un item en el buffer usando la función insert_item(), cuyo código se puede ver en 9,en la cual, además de 'añadir' un elemento al buffer, se reajusta el puntero para indicar la siguiente posición en la que se debe consumir. Después se aumenta la variable cuenta para

indicar que se añadió un elemento al buffer y se imprime éste (en el color correspondiente a ese hilo). Finalmente se envía una señal a la variable de condición del consumidor y se abre el mutex. Respecto a las dos últimas instrucciones, es importante que el envío de la señal sea lo más cercano posible a la salida de la región crítica (a la apertura del mutex).

```
buffer[posProducir]=item;//"producimos" un item en la posicion count
//Indicamos que el siguiente elemento a producir sea en la posicion posterior del buffer
//(o en la primera si estamos en la ultima)
posProducir=(posProducir+1)%N; //Nos aseguramos que, al llegar a N, la cuenta vuelva a 0
```

Listing 9: *Código de la función insert_item()*

En cuanto el consumidor, cuyo código se muestra en 10, el código es a grandes rasgos similar al del productor, y se señalarán las diferencias.

```
//El consumidor realiza las iteraciones que le marca el campo del struct, segun el reparto hecho en el main
for(i=0;i<color_iteraciones->numeroIteraciones;i++){
    pthread_mutex_lock(&mutex); //El productor obtiene acceso exclusivo a la region critica
    //Esperamos a que se reciba la señal de que se puede producir
    while(cuenta==0) pthread_cond_wait(&condConsumer,&mutex);
    --cuenta; //Decrementamos el numero de elementos para consumir que contiene el buffer
    item=remove_item();
    printf("%s",color_iteraciones->color);/
    printf("Consumidor: ");
    for(int j=0;j<N;j++){ //Imprimimos el buffer para que sea mas sencillo de interpretar
        printf(" %d ",buffer[j]);
    }
    printf("\n");
    pthread_cond_signal(&condProducer);//Enviamos una señal al productor para despertarlo
    pthread_mutex_unlock(&mutex); //Salimos de la region critica
    consume_item(item);
    sleep(rand()%5);//Sleep aleatorio entre 0 y 4 segundos
  }
  pthread_exit(0);
```

Listing 10: *Código de la función consumer()*

En este caso, lo que se comprueba es si el buffer está vacío (en ese caso, cuenta=0), y de ser así, se espera la señal a la variable de condición asociada a los consumidores. Además, en este caso se envía la señal a la variable de condición del productor. En 11 puede consultarse en código de la función remove_item, en la cual, además de 'eliminar' un elemento del buffer, se reajusta el puntero para indicar la siguiente posición a consumir.

```
item=buffer[posConsumir];//"Consumimos" el item que se encuentra en la posicion count
buffer[posConsumir]=-1; //-1 indica que esta vacio el buffer
//Indicamos que el siguiente elemento a consumir sea en la posicion posterior del buffer
//(o al inicio si estamos en la ultima)
posConsumir=(posConsumir+1)%N; //Nos aseguramos que, al llegar a N, la cuenta vuelva a 0
return item;
```

Listing 11: *Código de la función remove_item()*

Finalmente, destacar que en ambos códigos se añaden sleep() de duración aleatoria entre 0 y 4 segundos para intentar desacoplar lo máximo posible consumidores y productores, con el fin de lograr diferentes ritmos de trabajo y que en ocasiones se llegue a bloquear algún consumidor por estar buffer vacío o algún productor por estar lleno.

Si se ejecuta este código se podrá ver que funciona perfectamente, quedando patente que la utilización de mutex y variables de condición es una manera efectiva para solucionar el problema del productor-consumidor.