Práctica 3: Sincronización de procesos con semáforos

Pablo Rodríguez Fernández

Marzo 2020

I. Contenido del documento

En este documento , tras una breve introducción al uso de semáforos como herramienta de sincronización de procesos, se explicará como están realizados y las conclusiones obtenidas de los ejercicios propuestos. Estos tratan sobre el problema del productor-consumidor, y su posible solución empleando semáforos.

II. Semáforos: Una breve introducción

Un semáforo es una variable que toma valores enteros positivos, regida por dos operaciones básicas:

- **Down**: Si el semáforo es mayor que 0, se decrementa y el proceso continúa. En el caso de que sea 0, el proceso queda bloqueado.
- **Up**: Incrementa el semáforo. En caso de que algún otro proceso se encuentre bloqueado debido al semáforo, se despierta y ejecuta su down.

La gran ventaja de los semáforos es que las acciones que realizan las funciones up y down se realizan en conjunto como una sola acción atómica indivisible. Por tanto, ningún proceso puede modificar un semáforo si otro lo está haciendo en ese mismo momento. Este hecho permite resolver problemas de sincronización y evitar condiciones de carrera crítica, y por eso es indicado para resolver el problema del productor-consumidor, como se verá en los diferentes ejercicios.

III. Ejercicios

Cada uno de estos ejercicios consta de 2 códigos, representando uno al consumidor y otro al productor. En estes códigos se incluye un main() en el cual se realizan diversas tareas previas (reserva de memoria, inicialización de variables), y que luego llama a las funciones *producer()* o *consumer()*, que contienen la codificación del problema del productor-consumidor. Al acabar estas funciones, y antes de finalizar la ejecución, cada programa realiza las liberaciones y cierres (ficheros, memoria, semáforos) que sean necesarios.

La separación en dos ejecutables distintos permite ejecutar los códigos en terminales diferentes y ver con claridad como evolucionan el consumidor y el productor.

El nombre de los códigos referentes a cada ejercicio se pueden encontrar en el **cuadro 1**.

En los siguientes subapartados se comentarán los dferentes ejercicios, incluyendo fragmentos destacables junto con su utilidad de cara a lograr la solución al ejercicio. Los códigos completos de cada ejercicio se encuentran en la misma carpeta comprimida donde se entrega este documento.

Ejercicio	Productor	Consumidor
1	3_producer.c	3_consumer.c
2	3_semProducer.c	3_semConsumer.c
3	3_generalProducer.c	3_generalConsumer.c

Cuadro 1: *Códigos asociados a cada ejercicios*

A. Ejercicio 1

En este primer ejercicio se nos pide estudiar el problema del **productor-consumidor** sin incluir ninguna medida de prevención de carreras críticas, con la intención de constatar la ocurrencia de estas.

Por ser este el primer ejercicio, se explicará que realiza el main antes de invocar a producer(), ya que en los ejercicios posteriores se realizarán tambien estas acciones

El programa recibe como argumento de entrada la ruta a un fichero, el cual se mapeará en memoria para crear una zona de memoria compartida. Antes de realizar este mapeado, se modifica su tamaño (en el código productor únicamente) para que la región sea lo suficientemente grande como para almacenar lo siguiente:

- La variable **count**, que almacena el número de espacios ocupados del buffer.
- El **buffer** (de un tamaño prefijado N), que será utilizado por el consumidor y el productor para introducir y sacar ítems.

En 1 se puede observar como se realiza esta modificación del tamaño.

```
//Tomamos como tamanho del fichero N+1 enteros, para almacenar el buffer y el count
if(ftruncate(fichero,(N+1)*sizeof(int))==-1){
  perror("Ampliacion del fichero erronea: ejecucion abortada\n");
  exit(-1);
}
```

Listing 1: *Modificación del tamaño del fichero*

Tras esto se realiza el mapeado del fichero en memoria utilizando la función *mmap()* [1] y se inicializa la zona de memoria compartida, como se ve en 2

```
proyeccion[0]=0;//Inicializamos count a 0
for(int i=1;i<=N;i++){
        proyeccion[i]=-1; //-1 indica que el buffer esta vacio.
}
```

Listing 2: *Inicialización de la memoria compartida*

Todas las acciones descritas también se realizan en el código del consumidor, salvo el cambio del tamaño del fichero.

Ahora se analizará el código de las funciones *consumer* y *producer*, que es la verdadera razón de ser del ejercicio.

En 3 se encuentra el código de producer(), sin incluir la inicializacion de variables. Se realiza un bucle infinito que, en primer lugar, produce un item. Luego comprueba si el buffer está lleno, de ser así espera hasta que quede un hueco libre, realizando comprobaciones continuamente: es lo que se denomina **espera activa**. En cuanto detecta que existe un hueco, inserta en el buffer el item en la posición que le indica count (ya que el buffer funciona como una estructura LIFO) e incrementa count para indicar que hay un elemento más en el buffer. Finalmente ejecuta un

[1] Ver código adjunto a este documento para más detalles sobre esta función y su utilización

sleep, cuya función es desacoplar el productor y el consumidor para que interactúen con el buffer a ritmos diferentes, buscando que se den condiciones de carrera.

```
void producer(){
        while(TRUE){
                item=produce_item();
                while(*count==N); //Esperamos mientras en buffer este lleno
                insert_item(item,count);
                ++(*count);//Incrementamos el contador
                printf("  Count:%d \n",*count);
                sleep(rand()%3);//Sleep aleatorio entre 0 y 2 segundos
        }
}
```

Listing 3: *Código de producer*

En 4 se encuentra el código de consumer(). Comienza comprobando si el buffer está vacío, y de ser así, realiza espera activa como en el caso anterior. Luego, 'saca' del buffer el item en la posición indicada por count y realiza un sleep para desacoplarlo del productor y además alargar la duración de la región crítica (ya que se sitúa en media de ella). Finalmente, decrementa el valor de count y consume el item que se retiró del buffer (lo imprime).

```
void consumer(){
        while(TRUE){
                while(*count==0); //Esperamos mientras el buffer este vacio
                item=remove_item(count);
                sleep(rand()%3); //Sleep aleatorio entre 0 y 2 segundos
                --(*count);
                consume_item(item);
                //Si count==10->El productor a incrementado contador antes de poder imprimirlo
                printf("  Count:%d \n",*count);
        }
}
```

Listing 4: *Código de consumer*

A.1. Conclusiones del ejercicio 1

Esta implementación 'pura' del problema del productor-consumidor es propensa a producir carreras críticas, ya que la espera activa no es un buen mecanismo de sincronización de procesos. La razón es que no se bloquea de manera alguna el acceso a las variables mientras un proceso se encuentra en la región crítica, pudiendo modificarlas el otro. Esto puede provocar que un mismo item sea consumidor más de una vez, lo cual al ejecutar el programa se puede observar cuando un mismo item es consumido varias veces para un valor de count concreto, aún cuando el productor haya producido un item con valor distinto).

B. Ejercicio 2

En este ejercicio se pide programar la solución al problema del **productor-consumidor** utilizando **semáforos** para el caso de un único consumidor y un único productor.

Comenzaremos comentando cómo se usan los semáforos en este caso.

En primer lugar, se declaran 3 semáforos, los cuales tiene diferentes funciones, como se ve en 5 Tras realizar las mismas acciones de reserva de memoria y manejo de ficheros que fueron explicadas anteriormente, se deben crear estos semáforos (ya que de momento solo se encuentran declarados). Esta acción solo se debe realizar una vez, en este caso se realizará en el código que ejecuta el productor, el cual se puede ver en 6. Cabe resaltar el uso de la funcion

```
//Semaforos utilizados en la solucion de Tanenbaum
sem_t *full;//Toma valores entre 0 y N. Indica los espacios ocupados del buffer
sem_t *empty;//Toma valores entre 0 y N. Indica los espacios libres del buffer
sem_t *mutex;//Toma los valores 0 o 1.Garantiza la exclusion mutua
```

Listing 5: *Declaración de los semáforos*

sem_unlink() antes de crear los semaforos con *sem_open()*. Esta función eliminar semáforos que existan con el mismo nombre que el pasado como parámetro, y evita problemas que puedan aparecer, ya que el nombre de un semáforo es único. En cuanto el programa consumidor, este

```
//Eliminamos posibles semaforos con el mismo nombre que existan en el kernel
sem_unlink("FULL");
sem_unlink("EMPTY");
sem_unlink("MUTEX");
//Creamos los semaforos: sem_open. Argumentos:
/*1.-Nombre        2.-Flag (en este caso de creacion)
3.-Modo de creacion 4.-Valor de inicializacion*/
full= sem_open("FULL",O_CREAT,0600,0);//Inicialmente hay 0 espacios ocupados en el buffer
empty= sem_open("EMPTY",O_CREAT,0600,N);//Inicialmente hay N espacios libres en el buffer
mutex= sem_open("MUTEX",O_CREAT,0600,1);//Inicialmente el buffer esta a 1
```

Listing 6: *Creación de los semáforos en el productor*

no crea los semáforos: simplemente los abre. Para ello utiliza la misma función utilizada para crearlos, pero con un menor número de argumentos, como se puede ver en 7. Antes de finalizar

```
//Abrimos los semaforos creados por el productor
//sem_open. Argumentos:
/*1.-Nombre 2.-Flags*/
full= sem_open("FULL",0);
empty= sem_open("EMPTY",0);
mutex= sem_open("MUTEX",0);
```

Listing 7: *Abertura de los semáforos en el consumidor*

el programa es necesario cerrar cada uno de los semáforos desde cada uno de los programas (al igual que se cierra un fichero), utilizando la función *sem_close(name)*, donde *name* es el nombre que se le asignó al crearlo al semáforo.

Ahora se analizarán los cambios que se introducen en las funciones que realizan de consumidor y de productor. Cabe decir en primer lugar que solo varía la función principal, es decir, solucionar le problema del productor-consumidor utilizando semáforos no implica realizar ningún cambio en las funciones *consume_item()*, *produce_item()*, etc. En la figura 8 se puede observar como queda la función *producer()*, y en 9 como queda *consumer()*.

Se pueden destacar 3 cambios reseñables en estos códigos respecto a la versión del ejercicio anterior:

- **Uso de sleep()**: En este caso las funciones sleep se sitúan fuera de la región crítica, para lograr que productor y consumidor vayan a ritmos diferentes y en ocasiones alguno de ellos se bloquee.

- **Impresion del buffer**: Para facilitar la compresión de los resultados, se ha incluído una impresión del buffer antes de modificarlo en cada iteración. El código de esta encuentra entre los semáforos que garantizan la exclusión mutua para garantizar que el estado del buffer no se vea modificado mientras se muestra.

- **Inclusión de semáforos:** Se incluyen llamadas a las funciones *sem_wait()* y *sem_post()*,

```
void producer(){
        while(iteraciones-- > 0){
                item=produce_item();
                sleep(rand()%5);//Sleep aleatorio entre 0 y 4 segundos
                sem_wait(empty); //down
                sem_wait(mutex); //down
                printf("Productor en región crítica\n");
                printf("        Buffer antes de producir: ");
                for(i=1;i<=N;i++){
                        printf(" %d ",*(proyeccion+i));
                }
                //Incrementamos en primer lugar pues el buffer comienza en la posicion 1, no en la 0
                ++(*count);
                insert_item(item,*count);
                printf("\n        -Count: %d",*count);
                printf("                -Item producido: %d\n",item);
                sem_post(mutex); //up
                sem_post(full); //up
        }
}
```

Listing 8: *Código del productor utilizando semáforos*

```
void consumer(){
        while(iteraciones-- > 0){
                sleep(rand()%5);//Sleep aleatorio entre 0 y 4 segundos
                sem_wait(full); //Down
                sem_wait(mutex); //Down
                printf("Consumidor en región crítica\n");
                printf("Buffer antes de consumir: ");
                for(int i=1;i<=N;i++){
                        printf(" %d ",*(proyeccion+i));
                }
                item=remove_item(*count);
                --(*count);
                printf("\n-Count: %d",*count);
        sem_post(mutex); //Up
                sem_post(empty); //Up
                consume_item(item);
        }
}
```

Listing 9: *Código del consumidor utilizando semáforos*

que realizan un down y un up sobre el semáforo sobre el que actúan, respectivamente. En primer lugar se realiza un down() sobre el semáforo *empty* o *full* (en función de si se encuentra en el productor, el cual va a introducir un item en un espacio libre, o en el consumidor, que va a 'consumir' un espacio ocupado). Tras eso se realiza un down sobre el semáforo *mutex*. De esta manera se asegura que el otro proceso no va a poder entrar en la región crítica, ya que si lo intenta, se encontrará con que dicho semáforo está a 0 y deberá esperar. Tras esto, se realiza un up de este semáforo (para que el otro proceso pueda entrar en la región crítica) y otro up del semáforo que aún no había sido utilizado en este fragmento (para actualizar el número de huecos ocupados o libres, según corresponda).

B.1. Conclusiones del ejercicio 2

La razón de que en este caso no se produzcan carreras críticas como en el ejercicio anterior es que el uso de los diferentes semáforos garantiza que , en cada instante, solo uno de los procesos se encuentre en región crítica, estando en ella el tiempo que necesite para realizar las

actualizaciones de la variable *count* y del buffer. Por tanto, estos tendrán siempre los valores correctos, y no se darán situaciones como consumir dos veces un mismo item, como sucedía en el ejercicios 1.

C. Ejercicio 3

En este último ejercicio se pide generalizar el ejercicio anterior a m productores y n consumidores simultáneos, es decir , realizar una **generalización** del problema del **productor-consumidor** utilizando **semáforos**.

Debido a como se ha realizado el apartado anterior, teniendo separado un código que se encarga del consumidor y otro que se encarga del productor, la generalización utilizando semáforos es relativamente sencilla. Las funciones *consumer()* y *producer()* no es necesario modificarlas[2], al igual que la mayoría del código. Los únicos cambios son en la lectura de argumentos (para incluir la lectura del número de productores y consumidores que se quieran crear, en cada caso), y un bucle que crea tantos procesos como productores/consumidores se hayan indicado. A modo de ejemplo, en 10 se encuentra ese fragmento de código para el caso del productor (el del consumidor es análogo). Simplemente aclarar que el array colors[] contiene 15 colores que se asignan a los diferentes procesos "productores".

```
for(i=0;i<m;i++){//Se crearan m productores
    if((hijo=fork())<0){ //Creamos los procesos
      printf("Error en la creacion de uno de los procesos: Ejecucion abortada\n");
      exit(-1);
    }
    else if(hijo==0){ //Proceso productor
      producer(colors[i%15]); //Hay 15 colores disponibles. Si hay mas productores, se repetiran
      exit(1); //Salimos do proceso
    }
    else //Proceso padre
      ; //Continuamos con el bucle de creacion de procesos productores
  }
  //Esperamos por los m productores
  for(i=0;i<m;i++){
    wait(&estado);
  }
```

Listing 10: *Creación de varios productores*

C.1. Conclusiones del ejercicio 3

La razón de que la solución siga funcionando a pesar de existir varios procesos que consumen y producen en el buffer (el cual es común para todos) es que los semáforos también son comunes: todos utilizan los mismos. Véase en el caso del productor (análogo con el consumidor): Si un productor no puede entrar en la región crítica por estar el semáforo *mutex* a 0, ningún productor podrá , ya que todos verán ese semáforo a 0. Por otro, si un consumidor acaba de salir de la región crítica y pone dicho semáforo a 1, los distintos procesos productores competirán entre sí por ver quién entra en la región crítica. Una vez entre uno y el semáforo *mutex* esté otra vez a 0, el resto deberán esperar como en el caso anterior.

[2] Aún así, se ha realizado una pequeña modificación: se ha incluído la impresión de todo el texto en el terminal en colores, para facilitar saber que productor o consumidor está trabajando en un determinado momento. Se puede ver en el código que se anexa con este documento