Práctica 5: Sincronización de procesos con paso de mensajes

Pablo Rodríguez Fernández

Mayo 2020

I. Contenido del documento

En este documento , tras una breve introducción al uso de paso de mensajes 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 la herramienta anterior. Tras esto, se explicará un matiz sobre el número de iteraciones que debe realizar el productor y el consumidor. Finalmente, se explicará la diferencia entre las dos versiones que se pidieron implementar: consumir los items siguiendo un orden LIFO y consumirlos siguiendo un orden FIFO.

II. Paso de mensajes: Una breve introducción

El paso de mensajes es un método de comunicación entre procesos basado en dos primitivas: *send* y la *receive*:

- **Send**: Envía un mensaje a un destino especificado
- **Receive**: Recibe un mensaje desde un origen especificado (o de cualquiera, si así lo detalla). Además, puede bloquear al proceso hasta que se reciba un mensaje.

Para asegurarse de que todos los mensajes enviados lleguen a su destino, emisor y receptor pueden acordar el empleo de *ackowledgement*, es decir, que el receptor envíe al emisor un mensaje indicando que un mensaje ha llegado (cuando sea el caso). Si el emisor, pasado un tiempo, no ha recibido este 'acuse de recibo', volverá a transmitir el mensaje.

En algunos sistemas operativos, cuando un proceso recibe un mensaje, lo introduce en el denominado **buffer de recepción**. Cuando un proceso ejecuta un *receive* lo que se hace realmente es comprobar el estado de dicho buffer, y si no está vacío, extraer un mensaje. También puede existir un **buffer de emisión**, en el cual se copia el mensaje del emisor para enviarlo al emisor. El buffer de recepción será de gran importancia en la solución propuesta por Tanenbaum que se verá del problema del productor-consumidor.

III. Ejercicio

En esta práctica, el ejercicio planteado consiste en resolver el problema del productor-consumidor utilizando paso de mensajes. En particular, se debía implementar la solución propuesta por Tanenbaum, pero realizando dos versiones: una en la cual el consumidor consume los items/mensajes en un orden LIFO, y otra en un orden FIFO. La diferencia entre ambas versiones se tratará al final del documento. Como esta diferencia es mínima, se explicará con detalle solamente la implementación FIFO, en lugar de ambas.

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 se muestran en 1 las constantes que se usarán en el código (junto con su significado), definidas así para facilitar su modificación, ya que son parámetros clave del problema. Además se muestran las dos variables globales que se usarán para hacer referencia a las colas de mensajes asociadas al consumidor y al receptor.

```
#define MAX_BUFFER 6  //tamaño del buffer
#define DATOS_A_PRODUCIR 10  //número de datos a producir
#define MSG_CHAR_SIZE 15  //Numero de chars enviados en un mensaje

mqd_t almacen1; //cola de entrada de mensajes para el productor
mqd_t almacen2; //cola de entrada de mensajes para el consumidor
```

Listing 1: *Definición de constantes*

En 2 se puede observar el *main()* del productor, el cuál fue facilitado por los profesores de la asignatura. De esta parte cabe recalcar la importancia de usar *mq_unlink* antes de crear una cola de mensajes, ya que, al igual que sucede con los semáforos, tienen asociado un nombre. Si se intenta crear una cola/semáforo con el mismo nombre que uno existente, simplemente se abrirá el ya existente, no uno nuevo. Si no se tiene en cuenta este hecho se pueden originar graves problemas en ejecuciones posteriores.

```
 struct mq_attr attr;
/* Atributos de la cola */
attr.mq_maxmsg = MAX_BUFFER; //Tamaño de la cola
attr.mq_msgsize = sizeof (char)*MSG_CHAR_SIZE; //Tamaño de cada mensaje enviado
/* Borrado de los buffers de entrada por si existían de una ejecución previa*/
mq_unlink("/ALMACEN1");
mq_unlink("/ALMACEN2");
/* Apertura y creacion de los buffers */
almacen1 = mq_open("/ALMACEN1", O_CREAT|O_RDONLY, 0777, &attr);
almacen2 = mq_open("/ALMACEN2", O_CREAT|O_WRONLY, 0777, &attr);
if ((almacen1 == -1) || (almacen2 == -1)) {
  perror ("mq_open");
  exit(EXIT_FAILURE);
}
//Invocamos la función que realiza la producción de items
productor();
//Cerramos el acceso a las colas de mensajes
mq_close(almacen1);
mq_close(almacen2);

exit(EXIT_SUCCESS);
```

Listing 2: *Main del productor*

Respecto al main del consumidor, el cual se muestra en 3, decir que es más simple que el anterior, pues el encargado de crear las colas es el productor, con lo cual el consumidor solo debe abrirlas. Por esta razón no debe realizar el *mq_unlink*, ya que de hacerlo, se eliminarían las colas creadas por el productor. Simplemente debe abrir las colas con las flags de lectura y edición adecuadas (según sea la cola del productor, en la que introducirá mensajes vacíos, o la suya propia, de la cual extraerá mensajes solamente).

Se pasará a analizar ahora las funciones de **productor()** y **consumidor()**, que simulan el comportamiento del productor y el consumidor de este problema. Señalar que estas funciones hacen empleo de dos funciones auxiliares: *consume_item*, que imprime un item que 'consume'

[1] Para ver la implementación completa con todos los detalles comentada, se adjunta el código del ejercicio

```
/* Apertura de los buffers */
  almacen1 = mq_open("/ALMACEN1", O_WRONLY);
  almacen2 = mq_open("/ALMACEN2", O_RDONLY);
  if ((almacen1 == -1) || (almacen2 == -1)) {
    perror ("mq_open");
    exit(EXIT_FAILURE);
  }
  //Invocamos la función que realiza el consumo de items
  consumidor();
  //Cerramos el acceso a las colas de mensajes
  mq_close(almacen1);
  mq_close(almacen2);

  exit(EXIT_SUCCESS);
```

Listing 3: *Main del consumidor*

el consumidor, y *produce_item*, que genera una cadena que 'produce' el productor.[2] Además, ambas funciones incluyen llamadas a la función *sleep()* para desacoplar consumidor y productor y lograr ritmos de producción y consumo diferentes.

Respecto a la función de producción, se puede observar su código en 4.

```
void productor() {
  int i;
  char mensaje[MSG_CHAR_SIZE];
  char vacio[0];
  //Esperamos a que el consumidor llene el buffer del productor con mensajes vacios
  struct mq_attr attr;
  attr.mq_curmsgs=0;
  //Esperamos a que el consumidor llene el buffer del productor con mensajes vacios
  while(attr.mq_curmsgs!=MAX_BUFFER) mq_getattr(almacen1,&attr);
  for(i=0;i<DATOS_A_PRODUCIR;i++){
    produce_item(mensaje);
    mq_receive(almacen1,vacio,sizeof(mensaje),0); //Esperamos a que llegue un mensaje vacio
    mq_send(almacen2,mensaje,sizeof(mensaje),0); //Enviamos el mensaje al consumidor
  }
}
```

Listing 4: *Función de producción*

En primer lugar, se inicializan las variables, incluyendo dos cadenas de caracteres, una de tamaño MSG_CHAR_SIZE utilizada para guardar el item que se envia al consumidor y otra de tamaño 0 para recibir los mensajes 'vacíos' del consumidor. La razón de ser de esta última cadena es que es necesario que a la función *receive* se le pase una cadena donde almacenar el mensaje (en este caso vacío, de ahí el tamaño 0). Tras esto, se comprueba en un bucle while si el buffer de entrada está lleno -obteniendo para ello el atributo *mq_curmgs* del almacen1, que contiene el número mensajes en un momento determinado en el buffer-. Esta espera, si bien no es necesaria, facilita el desacoplamiento entre los ritmos de trabajo de productor y de consumidor.

Una vez se llena el buffer por primera vez, se entra en el bucle de producción, en el cual, tras llamar a la función *produce_item()* para generar un item, se espera a que llegue un mensaje vacío del consumidor (que se puede entender como 'enviame un item'). En este punto cabe destacar que la función *mq_receive()* necesita que el tamaño del mensaje recibido sea mayor o igual que el tamaño de mensaje fijado para la cola. De ahí, en la función *receive* se utilice como parámetro de tamaño el tamaño de la cadena que almacena el mensaje, que se corresponde con el fijado en el main como tamaño de mensaje, en lugar del tamaño de la cadena vacía. Finalmente, se

[2]Para ver la implementación detallada de estas funciones, ver el código del ejercicio

envía al consumidor el item utilizando la función *mq_send*, a través del almacen2.

Respecto al consumidor, su código se puede observar en 5

```
void consumidor(){
  int i;
  char mensaje[MSG_CHAR_SIZE];
  char mensajeVacio[0]; //La solucion de Tanenbaum implica el envio de mensajes vacios

  for(i=0;i<MAX_BUFFER;i++){
    mq_send(almacen1,mensajeVacio,sizeof(mensajeVacio),0); //Enviamos N mensajes vacios
  }
  for(i=0;i<DATOS_A_PRODUCIR-MAX_BUFFER;i++){
    mq_receive(almacen2,mensaje,sizeof(mensaje),0); //Esperamos a que llegue un mensaje
    mq_send(almacen1,mensajeVacio,sizeof(mensajeVacio),0);//Enviamos un mensaje vacio
    consume_item(mensaje);
  }
  for(i=0;i<MAX_BUFFER;i++){
    mq_receive(almacen2,mensaje,sizeof(mensaje),&prioridad); //Esperamos a que llegue un mensaje
    consume_item(mensaje,prioridad);
  }
```

Listing 5: *Función de consumo*

Tras inicializar las variables (de manera similar al productor), envía mensajes 'vacíos' hasta llenar la cola del productor (tal como propone la solución de Tanenbaum). Tras esto entra en el bucle de consumo, en el cual extrae un mensaje del almacen2, al cual el productor envía los items que produce. De estar esta cola vacía, se quedará esperando hasta que llegue un nuevo item. Tras esto, envía un nuevo mensaje vacío al productor, para indicarle que quiere otro item, y consume el item. Tras acabar este bucle, se ejecuta otro bucle, que será explicado en el siguiente apartado.

A. Diferencia entre las iteraciones del consumidor y del productor

En el productor, como se puede ver en 4, el bucle principal realiza un número de iteraciones igual al número de datos a producir. Sin embargo, en el consumidor este número es menor, correspondiéndose esta diferencia al número de elementos del buffer. La razón de esto es que, inicialmente, el consumidor envía MAX_BUFFER mensajes vacíos (es decir, peticiones de producción) al productor. Tras esto, el consumidor entra en un bucle en el cual, por cada item que consume, solita uno al productor. Por tanto, si el bucle principal del consumidor se ejecutase un número de veces igual a DATOS_A_PRODUCIR, se estarían solicitando ese número de items a mayores de los que se solicitaron antes. Por tanto, para que no se soliciten items de más -ya que se requiere que el productor solo produzca un número de items prefijado-, debemos 'recortar' el bucle principal del consumidor. Por esa razón el bucle final de 5 realiza MAX_BUFFER iteraciones ,en las cuales solo se extraen elementos para consumirlos, sin realizar peticiones de nuevos items.

B. Diferencia entre las versiones LIFO y FIFO

En este ejercicio se pedía realizar dos versiones. En una de ellas, el consumidor debería consumir los items en el orden LIFO, y en la otra, consumirlos en el orden FIFO. En este caso, esto significa que, en una de las versiones, el consumidor debería retirar de su cola de mensajes el más antiguo (FIFO), y en la otra, el último mensaje en llegar (LIFO). Por como está hecha la función *receive*, a priori esto no es posible, ya que extrae de una cola de mensajes el más antiguo. Por tanto, por defecto se tiene un comportamiento FIFO.

Sin embargo, si se profundiza en esa función, se podrá ver como realmente, lo que extrae, es el **mensaje más antiguo** que tenga la **prioridad más alta**. Por defecto, esta prioridad es 0

(dando el comportamiento FIFO). Sin embargo, utilizando esta prioridad, es posible que el consumidor extraiga sus mensajes según un orden LIFO. Basta con que el productor envíe sus items asignándoles una prioridad creciente. De esta manera, el último mensaje será también el que tenga una prioridad más alta, y por tanto, será el que se retire. Cabe decir que esta solución es limitada, ya que cardinal de la prioridad debe ser inferior a 32767, que es el valor de sysconf(_SC_MQ_PRIO_MAX) - 1, y que en LINUX se corresponde con el máximo cardinal que puede tomar el campo de prioridad. [3]. Por tanto, no sería posible utilizar la versión LIFO si la cantidad de elementos a producir va a ser mayor que ese número. Sin embargo, al realizarse las pruebas en este ejercicio con 100-200 items a producir, se decidió utilizar esta implementación.

A nivel de código, los cambios que son necesarios son mínimos, pudiéndose observar los del productor en 6 y los del consumidor en 7.

```
mq_send(almacen2,mensaje,sizeof(mensaje),prioridad);
prioridad++;
```

Listing 6: *Cambios productor LIFO*

```
mq_receive(almacen2,mensaje,sizeof(mensaje),&prioridad); //Esperamos a que llegue un mensaje
```

Listing 7: *Cambios consumidor LIFO*

Los cambios del consumidor son totalmente optativos, ya que la función *receive* extrae el de mayor prioridad siempre, como se dijo anteriormente. Sin enbargo, se decidió realizar una pequeña variación para facilitar la lectura de resultados durante la ejecución.

En ambos casos se introduce una variable de tipo entero para almacenar la prioridad, que se utiliza en las funciones de envío y recepción. En el caso del productor, esta se incrementa en cada iteración del bucle (para lograr que sea siempre creciente). Respecto al consumidor, esta se utiliza para obtener la prioridad del mensaje que se extrae de la cola. En ambos casos también se incluyó la impresión del campo prioridad [4], para que fuese más fácil ver que se cumple la extracción LIFO.

IV. Cierre del documento

Tras realizar este ejercicio queda comprobado que el uso de paso de mensajes es totalmente válido para implementar la solución del problema del productor-consumidor propuesta por Tanenbaum. Por tanto, puede ser utilizado como mecanismo de sincronización, siendo especialmente útil en aquellos casos en los cuales se opera en dispositivos sin memoria común, incluso con distintos sistemas operativos.

[3] Para más información, consultar el manual de la función *mq_overview*
[4] Puede verse en el código que acompaña al informe