Practica 5: Sincronización de procesos con paso de mensajes

Manuel José Corral Pazos, Grupo 1 (Jueves)

Notas previas: En esta práctica se han creado un total de 4 códigos, dos por cada versión LIFO/FIFO que se corresponden al productor y al consumidor. Además, se utilizará durante toda la práctica la librería <mqueue.h>, con el fin de simular un buzón para el paso de mensajes entre los procesos. Para la realización del ejercicio, se tomará lo siguiente:

- El productor generará una letra mayúscula aleatoria cada vez que se produzca un objeto.
- Los mensajes “vacíos” enviados por el consumidor al productor serán el símbolo ‘_’, simulando dicho tipo de mensaje.
- Se empleará la función sleep con números aleatorios entre 0 y 4 en los momentos adecuados, para simular la función de producción y consumición de forma más realista. Además, ayudará a que en ciertos momentos el buffer esté lleno de elementos o completamente vacío, debido a las posibles diferencias de velocidades entre los procesos ocasionadas por los sleeps.

Código

En primer lugar, se incluyen las librerías necesarias y se crean las variables globales. Entre ellas, encontramos las variables de cola, las cuales servirán para el envío de mensajes del productor y el consumidor respectivamente. También encontramos la definición de constantes, que servirán como tamaño máximo de buffer y número de datos a producir, dato que se utilizará en el bucle.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

#define MAX_BUFFER 6 //tamaño del buffer
#define DATOS_A_PRODUCIR 100 //numero de datos a producir/consumir

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

Lo primero que aparece en el main es la creación de la estructura que compondrá los atributos de las colas, utilizando la constante definida anteriormente para el número máximo de mensajes que puede albergar la cola, y el tamaño de char para el tamaño máximo de mensaje, puesto que los mensajes serán caracteres de texto.

```
struct mq_attr attr; //atributos de la cola
attr.mq_maxmsg=MAX_BUFFER; //numero maximo de mensajes
attr.mq_msgsize=sizeof(char); //tamaño maximo de mensajes
```

En el consumidor, se añaden dos invocaciones a la función mq_unlink de la cola antes de sus creaciones, con el fin de evitar problemas en caso de que las colas ya hayan sido creadas con anterioridad. Se hace en el consumidor puesto que va a ser el proceso que se va a ejecutar en primer lugar.

```
mq_unlink("/ALMACEN1");
mq_unlink("/ALMACEN2");
```

Para la creación de las colas, se utiliza la función mq_open, la cual toma como parámetros el nombre de la cola, las vlags que controlan las operaciones posibles en sobre la cola, los permisos, y la estructura que definirá sus atributos. En la captura posterior, se muestra el ejemplo del productor, que tendrá permisos de escritura sobre el la variable almacen1, y de lectura sobre almacen2. En caso de consumidor, los permisos se darán al revés, teniendo de escritura sobre almacen2 y de lectura sobre almacen1. Cabe destacar que aquí, cada proceso tiene dos buffers presentes en su kernel, uno de entrada y otro de salida.

```
//Apertura de los buffers
almacen1=mq_open("/ALMACEN1", O_CREAT|O_WRONLY, 0777, &attr);
almacen2=mq_open("/ALMACEN2", O_CREAT|O_RDONLY, 0777, &attr);

if((almacen1==-1) || (almacen2==-1)){
    perror("mq_open");
    exit(EXIT_FAILURE);
}
```

Antes de la función de producir/consumir, se crean las variables necesarias para el correcto funcionamiento de la operación. Hay que destacar que la variable prio solamente existe en el productor de la versión LIFO del programa. Dicha variable se utilizará en la función mq_send en el atributo de prioridad, haciendo así que la cola funcione de forma LIFO tal y como se mostrará en posteriores capturas de pantalla.

```
srand(time(NULL));
char elemento; //el elemento que se producirá
char entrada; //servirá para recibir los mensajes vacíos del consumidor
double dormir=0.0; //variable para el sleep
unsigned int prio=0; //utilizada en la prioridad del send para hacer cola LIFO
```

En primer lugar, es importante destacar que en la solución propuesta por Tanenbaum para este problema se introduce en paso e mensajes en los dos sentidos, tanto desde productor hacia consumidor como a la inversa. Los mensajes enviados desde el consumidor al productor son mensajes vacíos, indicando que el consumidor tiene la capacidad para recibir y leer mensajes. Los mensajes enviados desde el productor al consumidor son los objetos produciros.

Ahora, se mostrará cómo el consumidor envía MAX_BUFFER mensajes vacíos a la cola del productor. La razón de esto es llenar el buffer del productor antes de empezar a consumir, indicando que está preparado para leer MAX_BUFFER mensajes.

```
for(int i=0; i<MAX_BUFFER; i++){
    mq_send(almacen2, &elemento, sizeof(elemento), 0);
    printf("Consumidor\n\tMensaje vacio enviado: %c\n", elemento);
}
```

Tras el envío de dichos mensajes, el consumidor entra en un bucle que no dura lo mismo que el productor. La razón es que el consumidor ya envió MAX_BUFFER mensajes al productor, por lo que en el siguiente bucle ha de mandar DATOS_A_PRODUCIR-MAX_BUFFER mensajes para no romper el balance entre los objetos enviados y los objetos pedidos. Dentro del bucle en sí, lo primero es una función de mq_receive, que coge el mensaje enviado por el productor (que está en almacen 1) y lo guarda en la variable entrada. Después, se envía el mensaje vacío al productor indicando que ya está preparado para recibir otro elemento; se imprime el elemento recibido y se entra en el sleep, simulando la consumición del elemento.

```
for(int i=0; i<DATOS_A_PRODUCIR-MAX_BUFFER; i++){
//en este caso, el bucle va desde 0 hasta datosAProducir-maxBuffer
//la razon es porque el consumidor y el productor tienen que
// enviar y recibir el mismo número de mensajes
//el consumidor, ya mandó un total de maxBuffer mensajes al productor
//por tanto, ha de enviar menos mensajes en el siguiente bucle for
    mq_receive(almacen1, &entrada, attr.mq_msgsize, 0);
    printf("Consumidor\n\tMensaje recibido: %c\n", entrada);
    //elemento=mensaje[1];
    mq_send(almacen2, &elemento, sizeof(elemento), 0);
    printf("\tElemento consumido: %c\n", entrada);
    dormir=drand48()*4.0;
    sleep(dormir);
}
```

Ahora veamos la función de producir. En este caso el bucle sí que dura DATOS_A_PRODUCIR iteraciones, ya que no hay ninguna operación previa como sí la había en el caso del consumidor. Se produce el objeto (con el sleep para simular dicha producción) y se recibe el mensaje vacío del consumidor. Por tanto, se envía el mensaje. En caso de la cola FIFO, el último argumento de la función mq_send será 0, puesto que no hay que establecer ningún tipo de modificación en el orden de consumición de los mensajes y el funcionamiento usual de la función ya cumple las características buscadas. Por otra parte, el la cola LIFO se utiliza el atributo de prioridad de la función para simular el funcionamiento. Por ello, cada vez que se envía un mensaje se aumenta en uno la prioridad con la que se envía el mensaje de forma que como los mensajes son dispuestos en la cola en orden de prioridad descendente, se obtiene el comportamiento LIFO buscado.

```
    for(int i=0; i<DATOS_A_PRODUCIR; i++){
        //producir elemento
        elemento='A' + (rand()%(90-65)); //genera una letra aleatoria
        dormir=drand48()*4.0;
        sleep(dormir);
        mq_receive(almacen2, &entrada, attr.mq_msgsize,0);
        printf("Productor\n\tMensaje recibido: %c\n", entrada);
        //en este caso se indica la cola de la que se hace el recieve, dónde se alma
cena, el tamaño, y la prioridad
        printf("\tElemento introducido: %c\n", elemento);
        mq_send(almacen1, &elemento, sizeof(elemento), prio);
        printf("\tMensaje enviado: %c\n", elemento);
        prio++;
    }
```

Para terminar, se cierran ambas colas con la función mq_close.

```
mq_close(almacen1);
mq_close(almacen2);
```

Ejecución Lifo:

Primero, el consumidor ejecuta el lazo en el que envía MAX_BUFFER mensajes vacíos al productor. Tras esto, se queda bloqueado a la espera de la recepción de algún mensaje del productor.

```
Consumidor
        Mensaje vacio enviado: _
Consumidor
        Mensaje vacio enviado: _
Consumidor
        Mensaje vacio enviado: _
Consumidor
        Mensaje vacio enviado: _
Consumidor
        Mensaje vacio enviado: _
Consumidor
        Mensaje vacio enviado: _
```

Se ejecuta el productor y se introducen los elementos V, P, E y L.

```
Productor
        Mensaje recibido: _
        Elemento introducido: V
        Mensaje enviado: V
Productor
        Mensaje recibido: _
        Elemento introducido: P
        Mensaje enviado: P
Productor
        Mensaje recibido: _
        Elemento introducido: E
        Mensaje enviado: E
Productor
        Mensaje recibido: _
        Elemento introducido: L
        Mensaje enviado: L
```

```
Consumidor
        Mensaje recibido: V
        Elemento consumido: V
Consumidor
        Mensaje recibido: E
        Elemento consumido: E
Consumidor
        Mensaje recibido: L
        Elemento consumido: L
Consumidor
        Mensaje recibido: P
        Elemento consumido: P
```

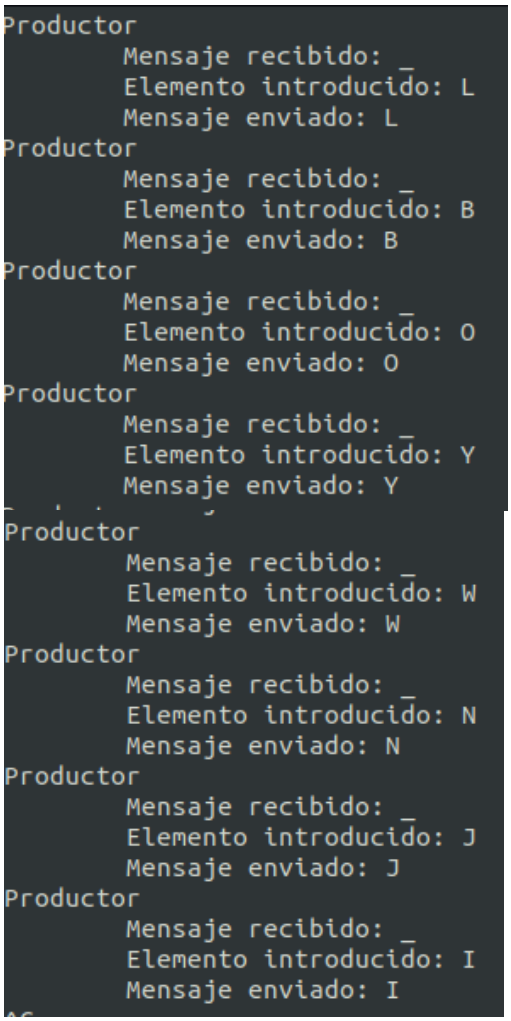
Si vamos a la salida del consumidor, nos encontramos con que el consumidor consume los elementos en orden V, E, L, P. Esto se debe a que el orden en el que sucedieron las cosas fue el siguiente:

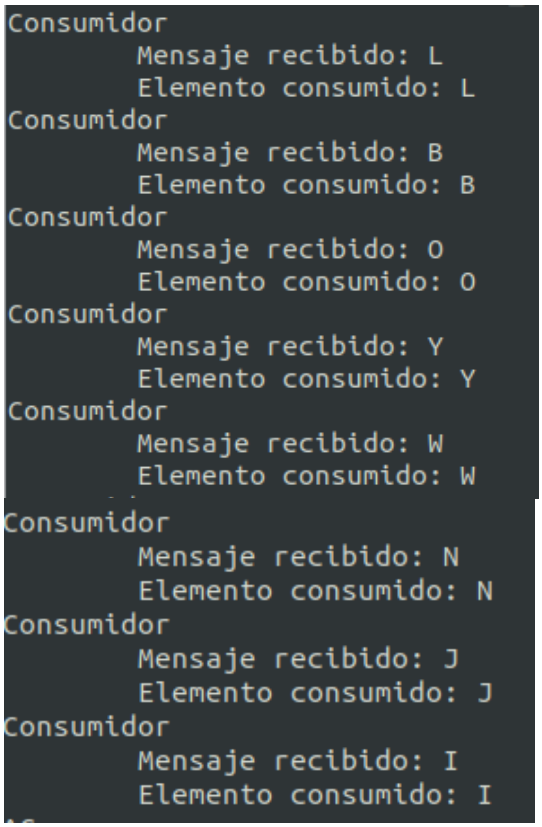
1. Productor envía V
2. Consumidor consume V
3. Productor envía P
4. Productor envía E
5. Consumidor consume E
6. Productor envía L
7. Consumidor consume L
8. Consumidor consume P

Como el orden es ese, se ve el correcto funcionamiento de la aplicación cuando se trata de cola Lifo.

Ejecución FIFO:

En este caso, lo esperado es que los elementos salgan y entren e los buffers en el mismo orden. Se ejecuta, y los resultados son los siguientes:





Después de que el consumidor envíe los mensajes vacíos, comienza el proceso de producción-consumición. El resultado es el esperado, los objetos son consumidos en el mismo orden en el que se producen, siguiendo la estrategia FIFO.