PRACTICA 3

ANTONIO GARCIA CASTILLO

PRODUCTOR CONSUMIDOR MULTIPLE

Para el ejercicio del productor consumidor múltiple lo primero que voy a comentar en mi solución es son las constantes definidas globales.

```
const int
  id_productor_min = 0,
  id_productor_max = 3,
  id_buffer = 4,
  id_consumidor_min = 5,
  id_consumidor_max = 9,
  num_productores = 4,
  num_consumidores = 5,
  num_procesos_esperado = 10,
  num_items = 20,
  num_items_productores = num_items/ num_productores,
  tag_productor = 100,
  tag_consumidor = 101,
  num_items_consumidores = num_items / num_consumidores,
  tam_vector = 10;
```

He usado un rango de ids para los procesos productores y un rango de ids para los procesos consumidores, con lo que en el main bastará preguntar si el id_propio esta dentro de ese rango o es exactamente 4, con lo que sería el id_buffer. Dependiendo de que rango se trate llamaremos a una función u otra.

```
  // ejecutar la operación apropiada a 'id_propio'
  if ( id_propio <= id_productor_max )
     funcion_productor(id_propio);

  else if ( id_propio == id_buffer )
     funcion_buffer();
  else if ( id_propio > id_buffer && id_propio <= id_consumidor_max)
     funcion_consumidor(id_propio - id_consumidor_min);
```

Con respecto a las demás constantes definidas, observamos que tenemos dos tags, uno para productor y otro para consumidor, las cuales nos ayudaran en el proceso buffer para poder hacer la correcta operación.
Primero, vamos a ver las funciones productor y consumidor.

```
void funcion_productor(int num_orden)
{
  num_orden = num_orden * num_items_productores;

  for ( unsigned int i= 0 ; i < num_items_productores ; i++ )
  {
     // producir valor
     int valor = producir(num_orden);
     num_orden++;
     // enviar valor
     cout << "Productor va a enviar valor " << valor << endl << flush;
     MPI_Ssend( &valor, 1, MPI_INT, id_buffer, tag_productor, MPI_COMM_WORLD );
  }
}
```

PRACTICA 3

ANTONIO GARCIA CASTILLO

PRODUCTOR CONSUMIDOR MULTIPLE

*Lo primero que hacemos es calcular el ítems desde el que queremos que empiece a producir, para ello basta con multiplicar el num_orden * num_items_productores. Esto establecerá el primer elemento a producir, luego observamos que el for hará num_items_productores iteraciones, con lo que nos aseguramos el rango a producir.*

```
void funcion_consumidor(int num_orden)
{
   int         peticion,
               valor_rec = 1 ;
   MPI_Status  estado ;

   for( unsigned int i=0 ; i < num_items_consumidores; i++ )
   {
      MPI_Ssend( &peticion,  1, MPI_INT, id_buffer, tag_consumidor, MPI_COMM_WORLD);
      MPI_Recv ( &valor_rec, 1, MPI_INT, id_buffer, tag_consumidor, MPI_COMM_WORLD, &estado );
      cout << "Consumidor ha recibido valor " << valor_rec << endl << flush ;
      consumir( valor_rec );
   }
}
```

En la función consumidor, básicamente lo que hacemos es un bucle que repetimos num_items_consumidores veces, en el cual enviamos una petición de 1 int del bufer de manera síncrona, a continuación lo recibimos y consumimos.

Con respecto a la función buffer, la gran diferencia ahora radica en que filtramos según el tag del emisor, con lo que con esto controlamos la escritura y lectura del buffer.

```
for( unsigned int i=0 ; i < num_items*2 ; i++ )
{
   // 1. determinar si puede enviar solo prod., solo cons, o todos

   if ( num_celdas_ocupadas == 0 )                 // si buffer vacío
      tag_emisor_aceptable = tag_productor ;       // $\sim$ solo prod.
   else if ( num_celdas_ocupadas == tam_vector )   // si buffer lleno
      tag_emisor_aceptable = tag_consumidor ;      // $\sim$ solo cons.
   else                                            // si no vacío ni lleno
      tag_emisor_aceptable = MPI_ANY_TAG ;         // $\sim$ cualquiera

   // 2. recibir un mensaje del emisor o emisores aceptables

   MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE,tag_emisor_aceptable, MPI_COMM_WORLD, &estado );

   // 3. procesar el mensaje recibido

   switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
   {
      case tag_productor: // si ha sido el productor: insertar en buffer
         buffer[primera_libre] = valor ;
         primera_libre = (primera_libre+1) % tam_vector ;
         num_celdas_ocupadas++ ;
         cout << "Buffer ha recibido valor " << valor << endl ;
         break;

      case tag_consumidor: // si ha sido el consumidor: extraer y enviarle
         valor = buffer[primera_ocupada] ;
         primera_ocupada = (primera_ocupada+1) % tam_vector ;
         num_celdas_ocupadas-- ;
         cout << "Buffer va a enviar valor " << valor << endl ;
         MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE ,tag_consumidor, MPI_COMM_WORLD);
         break;
   }
}
```

PRACTICA 3

ANTONIO GARCIA CASTILLO

FILOSOFOS CON INTERBLOQUEO

El problema que nos encontramos cuando rellenamos la plantilla facilitada de los filósofos, es que puede llegar a interbloqueo si todos piden su tenedor izquierdo a la vez, todos estarían esperando a su otro tenedor al mismo tiempo, con lo que nunca conseguirán su tenedor derecho.

En la siguiente imagen ofrezco una salida de este problema.

```
mpirun -np 10 --oversubscribe ./filosofos-interb_exe
Filósofo 8 solicita ten. izq.9
Filósofo 4 solicita ten. izq.5
Filósofo 0 solicita ten. izq.1
Ten. 1 ha sido cogido por filo. 0
Filósofo 2 solicita ten. izq.3
Ten. 5 ha sido cogido por filo. 4
Filósofo 6 solicita ten. izq.7
Ten. 9 ha sido cogido por filo. 8
Filósofo 8 solicita ten. der.7
Filósofo 4 solicita ten. der.3
Filósofo 0 solicita ten. der.9
Ten. 3 ha sido cogido por filo. 2
Filósofo 2 solicita ten. der.1
Ten. 7 ha sido cogido por filo. 6
Filósofo 6 solicita ten. der.5
```

Observamos que todos los filósofos han cogido su tenedor izquierdo, con lo cual nunca van a poder coger su tenedor derecho.

PRACTICA 3

ANTONIO GARCIA CASTILLO

SOLUCIÓN ENCONTRADA

Para solucionar este problema he optado por cambiar el orden en el que cogen los tenedores la mitad de los filósofos, si su id es mayor que 5. Con esto se consigue que nunca se llegue a esperar todos por el mismo tenedor.

```
if ( id <5){

cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
MPI_Ssend( &valor,  1, MPI_INT, id_ten_izq, etiq_comer, MPI_COMM_WORLD);

cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
MPI_Ssend( &valor,  1, MPI_INT, id_ten_der, etiq_comer, MPI_COMM_WORLD);

}
else if (id > 5){

  cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
  MPI_Ssend( &valor,  1, MPI_INT, id_ten_der, etiq_comer, MPI_COMM_WORLD);

  cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
  MPI_Ssend( &valor,  1, MPI_INT, id_ten_izq, etiq_comer, MPI_COMM_WORLD);
```

PRACTICA 3

ANTONIO GARCIA CASTILLO

SOLUCIÓN PROPUESTA

Comenzamos viendo las constantes definidas para la solución con el proceso camarero.

```
const int
    num_filosofos = 5 ,
    num_procesos  = 2*num_filosofos + 1 ,
    etiq_com = 11,
    etiq_levantarse = 13,
    etiq_sentarse = 14,
    id_camarero = 10;

int contador_filosofos = 0;
```

Las dos primeras son las constantes definidas para el numero de filósofos y el numero de procesos mientras que las tres siguientes nos sirven para filtrar los procesos. El contador de filósofos nos sirve para que no haya mas de 4 filosofos sentados a la mesa en el mismo instante de tiempo, evitando asi que se produzca un posible interbloqueo.

```
void funcion_camarero ()
{
  int valor, tag_emisor_aceptable;
  MPI_Status estado;

  while (true)
  {
     if ( contador_filosofos == 4){
       tag_emisor_aceptable = etiq_levantarse;
     }
     else if (contador_filosofos == 0){
       tag_emisor_aceptable = etiq_sentarse;
     }
     else {
       tag_emisor_aceptable = MPI_ANY_TAG;
     }

   MPI_Recv (&valor, 1, MPI_INT, MPI_ANY_SOURCE, tag_emisor_aceptable ,MPI_COMM_WORLD, &estado);

   switch (estado.MPI_TAG)
   {
     case etiq_sentarse:
       if ( contador_filosofos <4)
       {
         contador_filosofos++;
         cout << " Camarero ha sentado a filosofo " << estado.MPI_SOURCE << endl;

       }
       break;
     case etiq_levantarse:
       cout << " Camarero ha levantado a filosofo "<< estado.MPI_SOURCE << endl;
       contador_filosofos--;
       break;
   }
  }

}
```

El proceso camarero es la función clave de este problema, controla el numero de camareros sentados a la mesa y según ellos filtramos el tag aceptable, si son 4 solo levantarse, y si son 0 solo sentarse.

La función filósofos no merece gran mención, simplemente se limita a seguir el orden de acciones establecidas, enviando peticiones a los procesos correspondientes.

ANTONIO GARCIA CASTILLO

La función tenedores espera a recibir la llamada del proceso filosofo, y acto seguido espera su liberación

```
void funcion_tenedores( int id )
{
  int valor, id_filosofo ;  // valor recibido, identificador del filósofo
  MPI_Status estado ;       // metadatos de las dos recepciones

  while ( true )
  {
    // ..... recibir petición de cualquier filósofo (completar)

    MPI_Recv (&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_com ,MPI_COMM_WORLD, &estado);
    // ..... guardar en 'id_filosofo' el id. del emisor (completar)
    id_filosofo = estado.MPI_SOURCE;

    cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

    // ..... recibir liberación de filósofo 'id_filosofo' (completar)
    MPI_Recv (&valor, 1, MPI_INT, id_filosofo, etiq_com ,MPI_COMM_WORLD, &estado);
    cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
  }
}
```