Práctica 3. Envío de Mensajes con MPI

Jesús Sánchez de Lechina Tejada

Contents

Ejercicio 1. Múltiples productores y consumidores.

Cambios realizados

Respecto al programa original, `prodcons2.cpp` se han realizado las siguientes modificaciones:

- Se han añadido un conjunto de variables globales, constantes, que indican el número de productores, consumidores, items, IDs que pueden tomar productores/consumidores y buffer.
- En la función `main`, se han modificado las condiciones de lanzamiento de hebras, ahora se lanzará una función productora/consumidora por cada proceso que tenga el ID correspondiente.
- En las funciones `funcion_productor` y `funcion_consumidor` se les ha añadido un parámetro que corresponde con la ID (numero_orden).
- La función de producir ha sido modificada de modo que cada proceso produzca un intervalo de valores comprendido entre el primer valor correspondiente a su ID y el anterior al primer valor de la ID siguiente, de tamaño num-items/num-procesos. Para esto se han añadido una constante al ámbito global que indica el tamaño de estos intervalos.
- Para dar solución al problema de elegir un elemento aleatorio dentro de un subconjunto de un comunicador, al no poder crear varios comunicadores, usaremos la estrategia del establecimiento de etiquetas. Para ello, en primer lugar crearemos las etiquetas productor (1), consumidor (2) y buffer (3). Estas se nombrarán con el prefijo `etiq_`.
- Este último apartado nos lleva a realizar modificaciones en la función del buffer. En primer lugar ahora no esperamos una ID en concreto, sino que recibe el mensaje de cualquier fuente (MPI_ANY_SOURCE) y, en función de la disponibilidad del buffer selecciona qué etiquetas aceptará, si sólo las del consumidor, productor, o cualquiera.
- A la hora de procesar el mensaje recibido en el paso anterior, en caso de haber recibido el mensaje de un productor simplemente tiene que almacenar el valor en su buffer, pero si es un consumidor debe enviarle un mensaje con Ssend, para lo cual cambiaría el antiguo id del consumidor por el MPI_SOURCE del estado. Además, el switch ya no analiza el id del emisor, sino la etiqueta del estado.

Código Fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: prodcons_multiples.cpp
// Implementación del problema del productor-consumidor con
// un proceso intermedio que gestiona un buffer finito y recibe peticiones
// en orden arbitrario
// (versión con múltiples productores y consumidores)
//
// -----------------------------------------------------------------------------

#include <iostream>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <mpi.h>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
  num_prods             = 4 ,
  num_cons              = 5 ,
  id_max_productor      = num_prods -1 ,
  id_buffer             = num_prods ,
  id_max_consumidor     = num_prods + 1 + num_cons -1 ,
  num_procesos_esperado = 1+num_prods+num_cons ,
  num_items             = num_prods * num_cons *2,
  num_items_a_producir = num_items / num_prods,
  num_items_a_consumir = num_items /num_cons,
    // Debe ser múltiplo del número de prod y de consumidores
  tam_vector            = 20,
  etiq_prod         = 1,
  etiq_cons     = 2,
  etiq_buffer       = 3;

//**********************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//----------------------------------------------------------------------
```

```
template< int min, int max > int aleatorio()
{
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}
// ---------------------------------------------------------------------
// producir produce los numeros en secuencia (1,2,3,....)
// y lleva espera aleatorio
int producir(int id)
{
  static int contador = id* num_items_a_producir;
  sleep_for( milliseconds( aleatorio<10,100>()) );
  contador++ ;
  cout << "Productor " << id << " ha producido valor " << contador << endl << flush;
  return contador ;
}
// ---------------------------------------------------------------------

void funcion_productor(int num_orden)
{
  for ( unsigned int i= 0 ; i < num_items_a_producir ; i++ )
    {
      // producir valor
      int valor_prod = producir(num_orden);
      // enviar valor
      cout << "Productor " << num_orden << "  va a enviar valor " << valor_prod << endl << flus
      MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, etiq_prod, MPI_COMM_WORLD );
    }
}
// ---------------------------------------------------------------------

void consumir( int valor_cons, int id )
{
  // espera bloqueada
  sleep_for( milliseconds( aleatorio<110,200>()) );
  cout << "\tConsumidor " << id << " ha consumido valor " << valor_cons << endl << flush ;
}
// ---------------------------------------------------------------------

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

  for( unsigned int i=0 ; i < num_items_a_consumir; i++ )
```

```
    {
      MPI_Ssend( &peticion,  1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD);
      MPI_Recv ( &valor_rec, 1, MPI_INT, id_buffer, etiq_buffer, MPI_COMM_WORLD,&estado );
      cout << "\tConsumidor " << num_orden << "  ha recibido valor " << valor_rec << endl << fl
      consumir( valor_rec, num_orden );
    }
}
// -----------------------------------------------------------------------

void funcion_buffer()
{
  int        buffer[tam_vector],      // buffer con celdas ocupadas y vacías
    valor,                   // valor recibido o enviado
    primera_libre       = 0, // índice de primera celda libre
    primera_ocupada     = 0, // índice de primera celda ocupada
    num_celdas_ocupadas = 0, // número de celdas ocupadas
    etiq_emisor_aceptable ;    // etiqueta de emisor aceptable
  MPI_Status estado ;                 // metadatos del mensaje recibido

  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
    etiq_emisor_aceptable = etiq_prod ;       // $~~~$ solo prod.
      else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
    etiq_emisor_aceptable = etiq_cons ;      // $~~~$ solo cons.
      else                                          // si no vacío ni lleno
    etiq_emisor_aceptable = MPI_ANY_TAG ;     // $~~~$ cualquiera

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

      MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE, etiq_emisor_aceptable, MPI_COMM_WORLD, &est

      // 3. procesar el mensaje recibido

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

    case etiq_cons: // si ha sido el consumidor: extraer y enviarle
      valor = buffer[primera_ocupada] ;
      primera_ocupada = (primera_ocupada+1) % tam_vector ;
```

```
      num_celdas_ocupadas-- ;
      cout << "\t\tBuffer va a enviar valor " << valor << endl ;
      MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE, etiq_buffer, MPI_COMM_WORLD);
      break;
    }
    }
}

// ---------------------------------------------------------------------

int main( int argc, char *argv[] )
{
  int id_propio, num_procesos_actual;

  // inicializar MPI, leer identif. de proceso y número de procesos
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );

  if ( num_procesos_esperado == num_procesos_actual )
    {
      // ejecutar la operación apropiada a 'id_propio'
      if ( id_propio <= id_max_productor ) // Si está entre 0 y el id max
    funcion_productor(id_propio);
      else if ( id_propio == id_buffer ) // Si coincide con el id del buffer
    funcion_buffer();
      else               // En caso contrario se usa
                  // el consumidor
    funcion_consumidor(id_propio);
    }
  else
    {
      if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
    { cout << "el número de procesos esperados es:    " << num_procesos_esperado << endl
           << "el número de procesos en ejecución es: " << num_procesos_actual << endl
           << "(programa abortado)" << endl ;
    }
    }

  // al terminar el proceso, finalizar MPI
  MPI_Finalize( );
  return 0;
}
```

Resultado ejecución

Una salida del programa sigue el siguiente patrón:

```
Productor 2 ha producido valor 21
Productor 2  va a enviar valor 21
        Buffer ha recibido valor 21
        Buffer va a enviar valor 21
    Consumidor 9  ha recibido valor 21
Productor 2 ha producido valor 22
Productor 2  va a enviar valor 22
        Buffer ha recibido valor 22
        Buffer va a enviar valor 22
    Consumidor 5  ha recibido valor 22
Productor 3 ha producido valor 31
Productor 3  va a enviar valor 31
        Buffer ha recibido valor 31
        Buffer va a enviar valor 31
    Consumidor 7  ha recibido valor 31
Productor 2 ha producido valor 23
Productor 2  va a enviar valor 23
        Buffer ha recibido valor 23
        Buffer va a enviar valor 23
    Consumidor 6  ha recibido valor 23
Productor 0 ha producido valor 1
Productor 0  va a enviar valor 1
        Buffer ha recibido valor 1
        Buffer va a enviar valor 1
    Consumidor 8  ha recibido valor 1
Productor 3 ha producido valor 32
Productor 3  va a enviar valor 32
        Buffer ha recibido valor 32
Productor 1 ha producido valor 11
Productor 1  va a enviar valor 11
        Buffer ha recibido valor 11
Productor 3 ha producido valor 33
Productor 3  va a enviar valor 33
        Buffer ha recibido valor 33
Productor 0 ha producido valor 2
Productor 0  va a enviar valor 2
        Buffer ha recibido valor 2
Productor 2 ha producido valor 24
Productor 2  va a enviar valor 24
        Buffer ha recibido valor 24
Productor 2 ha producido valor 25
Productor 2  va a enviar valor 25
        Buffer ha recibido valor 25
Productor 3 ha producido valor 34
```

```
Productor 3  va a enviar valor 34
        Buffer ha recibido valor 34
        Buffer va a enviar valor 32
    Consumidor 9 ha consumido valor 21
    Consumidor 9  ha recibido valor 32
```

Esto quiere decir que cada proceso productor genera 10 valores que se envían al buffer. Este afirma haberlo recibido y envía dicho valor para que sea recibido y posteriormente consumido por un consumidor.

Ejercicio 2. La cena filosofar. Solución con interbloqueo.

Especificaciones

En primer lugar modificaremos la función de los filósofos, de manera que en cada iteración el filósofo requiera el tenedor de la izquierda y luego el de la derecha. Esto puede dar lugar a interbloqueo, pero lo solventaremos en la próxima solución. Posteriormente el filósofo come (espera aleatoria), suelta los tenedores y comienza a pensar (retraso aleatorio antes de comenzar la próxima iteración). Para ello enviaremos mensajes a los procesos tenedores síncronos con MPI_Ssend, pero necesitamos entonces un buffer auxiliar (aux, de tipo entero) cuyo valor no será relevante para el ejercicio pero será necesario para el paso de mensajes. Se enviará un mensaje a cada tenedor para solicitarlo, empezando siempre por el izquierdo, se realizará una espera, ahora se sueltan con un nuevo mensaje síncrono a los tenedores y se realiza una última espera aleatoria.

Ahora continuaremos con la implementación de la función de los tenedores. Una vez más debemos hacer uso de un buffer auxiliar para el paso de mensajes. Y al comienzo de cada iteración el recurso se encuentra disponible, por lo que el tenedor esperará de manera síncrona a que el mensaje de petición del tenedor sea enviado. Puesto que puede recibir de cualquier filósofo su fuente será `MPI_ANY_SOURCE`. A continuación el tenedor muestra un mensaje informando de qué filósofo ha cogido el tenedor. Y por último espera el mensaje del filósofo que cogió el tenedor para que lo suelte e informa e ello con un mensaje.

Código fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos-interb.cpp
// Implementación del problema de los filósofos (sin camarero).
//
// -----------------------------------------------------------------------------


#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
   num_filosofos = 5 ,
   num_procesos  = 2*num_filosofos ;


//**************************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//--------------------------------------------------------------------------

template< int min, int max > int aleatorio()
{
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}

// -----------------------------------------------------------------------

void funcion_filosofos( int id )
{
  int id_ten_izq = (id+1)              % num_procesos, //id. tenedor izq.
      id_ten_der = (id+num_procesos-1) % num_procesos; //id. tenedor der.
```

```
  int aux = 0; // No es usado, pero es necesario algún buffer para transmitir los mensajes
  while ( true )
    {
      cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

      cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);

      cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
      sleep_for( milliseconds( aleatorio<10,100>() ) );

      cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

      cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);

      cout << "Filosofo " << id << " comienza a pensar" << endl;
      sleep_for( milliseconds( aleatorio<10,100>() ) );

 }
}
// ---------------------------------------------------------------------

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

  int aux;
  while ( true )
  {
    MPI_Recv(&aux,1,MPI_INT,MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &estado);
    id_filosofo = estado.MPI_SOURCE;
    cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

    MPI_Recv(&aux, 1, MPI_INT, id_filosofo, 0, MPI_COMM_WORLD, &estado);
    // ..... recibir liberación de filósofo 'id_filosofo' (completar)
    cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
  }
}
// ---------------------------------------------------------------------

int main( int argc, char** argv )
{
   int id_propio, num_procesos_actual ;
```

```
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
   MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


   if ( num_procesos == num_procesos_actual )
   {
      // ejecutar la función correspondiente a 'id_propio'
      if ( id_propio % 2 == 0 )          // si es par
         funcion_filosofos( id_propio ); //   es un filósofo
      else                               // si es impar
         funcion_tenedores( id_propio ); //   es un tenedor
   }
   else
   {
      if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
      { cout << "el número de procesos esperados es:    " << num_procesos << endl
             << "el número de procesos en ejecución es: " << num_procesos_actual << endl
             << "(programa abortado)" << endl ;
      }
   }

   MPI_Finalize( );
   return 0;
}

// ----------------------------------------------------------------------
```

Salida de la ejecución

En este programa todos los filósofos cogen el tenedor de la izquierda en primer lugar y luego el de la derecha. Esto puede dar lugar finalmente a una (improbable) situación de interbloqueo donde todos tengan su tenedor de la izquierda pero nadie pueda coger el siguiente.

```
<prompt>$ mpirun -np 10 ./filosofos-interb
Filósofo Filósofo 4 solicita ten. izq.5
Filósofo 4 solicita ten. der.3
Filósofo 4 comienza a comer
Filósofo 6 solicita ten. izq.7
Ten. 3 ha sido cogido por filo. 4Ten. 5 ha sido cogido por filo. 4

0 solicita ten. izq.1
Filósofo Ten. 1 ha sido cogido por filo. 0
0 solicita ten. der.9
Filósofo 0Filósofo  comienza a comer
Ten. 2 solicita ten. izq.3
```

```
9 ha sido cogido por filo. 0
Filósofo 6 solicita ten. der.5
Ten. 7 ha sido cogido por filo. 6
Filósofo 8 solicita ten. izq.9
Filósofo 4 suelta ten. izq. 5
Filósofo Ten. 3 ha sido liberado por filo. 4
4 suelta ten. der. 3
Filosofo 4 comienza a pensar
Ten. 5 ha sido liberado por filo. 4
Ten. 5 ha sido cogido por filo. 6
Filósofo 6 comienza a comer
Ten. 3 ha sido cogido por filo. 2
Filósofo 2 solicita ten. der.1
Filósofo 6 suelta ten. izq. 7
Ten. 5 ha sido liberado por filo. 6Filósofo 6 suelta ten. der. 5
Filosofo 6 comienza a pensar
Ten. 7 ha sido liberado por filo. 6

Filósofo 4 solicita ten. izq.5
Filósofo 4 solicita ten. der.3
Ten. 5 ha sido cogido por filo. 4
Filósofo 6 solicita ten. izq.7
Ten. 7 ha sido cogido por filo. 6
```

Ejercicio 3. La cena filosofar. Sin interbloqueo.

Cambios

Los cambios con respecto a la versión anterior son muy pequeños y muy sutiles, de hecho, en todo el código sólo estamos cambiando un detalle. A la hora de procesar la función de los productores qué tenedor se va a coger se mira la propia id. Así si su id coincide con la del primer productor, este quiere obtener primero el tenedor de la derecha y luego el de la izquierda. Al contrario que todos sus otros compañeros filósofos.

Código fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos.cpp
// Implementación del problema de los filósofos (sin camarero).
// Solución SIN INTERBLOQUEO
//
// -----------------------------------------------------------------------------


#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
   num_filosofos = 5 ,
   num_procesos  = 2*num_filosofos ;


//**************************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//--------------------------------------------------------------------------

template< int min, int max > int aleatorio()
{
```

```
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}

// ---------------------------------------------------------------------

void funcion_filosofos( int id )
{
  int id_ten_izq = (id+1)              % num_procesos, //id. tenedor izq.
      id_ten_der = (id+num_procesos-1) % num_procesos; //id. tenedor der.

  int aux = 0; // No es usado, pero es necesario algún buffer para transmitir los mensajes
  while ( true )
    {
      if(id == 0){
    cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
    MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);

    cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
    MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

      }else{
    cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
    MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

    cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
    MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);
      }

      cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
      sleep_for( milliseconds( aleatorio<10,100>() ) );

      cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

      cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);

      cout << "Filosofo " << id << " comienza a pensar" << endl;
      sleep_for( milliseconds( aleatorio<10,100>() ) );

 }
}
// ---------------------------------------------------------------------

void funcion_tenedores( int id )
{
```

```
  int valor, id_filosofo ;  // valor recibido, identificador del filósofo
  MPI_Status estado ;       // metadatos de las dos recepciones

  int aux;
  while ( true )
  {
    MPI_Recv(&aux,1,MPI_INT,MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &estado);
    id_filosofo = estado.MPI_SOURCE;
    cout <<"Ten. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

    MPI_Recv(&aux, 1, MPI_INT, id_filosofo, 0, MPI_COMM_WORLD, &estado);
    // ..... recibir liberación de filósofo 'id_filosofo' (completar)
    cout <<"Ten. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
  }
}
// ----------------------------------------------------------------------

int main( int argc, char** argv )
{
   int id_propio, num_procesos_actual ;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
   MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


   if ( num_procesos == num_procesos_actual )
   {
      // ejecutar la función correspondiente a 'id_propio'
      if ( id_propio % 2 == 0 )          // si es par
         funcion_filosofos( id_propio ); //   es un filósofo
      else                               // si es impar
         funcion_tenedores( id_propio ); //   es un tenedor
   }
   else
   {
      if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
      { cout << "el número de procesos esperados es:    " << num_procesos << endl
             << "el número de procesos en ejecución es: " << num_procesos_actual << endl
             << "(programa abortado)" << endl ;
      }
   }

   MPI_Finalize( );
   return 0;
}

// ----------------------------------------------------------------------
```

Salida del programa

La principal diferencia respecto a la salida del ejercicio anterior está en que ahora el filósofo 0 coge en primer lugar el tenedor derecho en lugar del izquierdo, lo que imposibilita el interbloqueo.

```
<prompt>$ mpirun -np 10 ./filosofos
Filósofo 0 solicita ten. der.9
Filósofo Filósofo 6 solicita ten. izq.7
Filósofo 6 solicita ten. der.5
Filósofo 0 solicita ten. izq.1
Filósofo 0 comienza a comer
Filósofo 2 solicita ten. izq.3Ten. 7 ha sido cogido por filo. 6
Ten. 9 ha sido cogido por filo. 0
Ten. 1 ha sido cogido por filo. 0

Filósofo 2 solicita ten. der.1
Ten. 3 ha sido cogido por filo. 2
4 solicita ten. izq.5
Filósofo Ten. 5 ha sido cogido por filo. 4
4 solicita ten. der.3
Filósofo 8 solicita ten. izq.9
Filósofo 0 suelta ten. izq. 1
Filósofo 0 suelta ten. der. 9
Filosofo 0 comienza a pensar
Ten. 1 ha sido liberado por filo. 0
Ten. 1 ha sido cogido por filo. 2
Filósofo 2 comienza a comer
Ten. 9 ha sido liberado por filo. 0
Ten. 9 ha sido cogido por filo. 8
Filósofo 8 solicita ten. der.7
Filósofo 0 solicita ten. der.9
Filósofo 2 suelta ten. izq. 3
Filósofo 2 suelta ten. der. 1
```

Ejercicio 4. La cena de los filósofos con camarero.

Modificaciones

Para este programa partimos de la *solución con interbloqueo a la cena de los filósofos*

El principal elemento distintivo de este ejercicio es la presencia del camarero. Un proceso que se encargará de gestionar una espera selectiva para los procesos filósofos, que ahora requerirán estar sentados para poder coger los tenedores y enviarán un mensaje al camarero para realizar dicha petición.

El primer resultado que implica esto es que el número de procesos total se ha modificado, hay que incrementar en una unidad el número de procesos existentes.

El segundo resultado que surge directamente es la aparición de una nueva función camarera, que será la que se encargue de la gestión de los asientos. Esta se encontrará en un bucle infinito en el que analice si el número de filósofos sentados coincide con el número de filósofos totales menos uno, pues cuando sólo quede un sitio querremos evitar el interbloqueo impidiendo que se siente, y para esto haremos uso de etiquetas, de modo que cuando esté en este caso, en lugar de aceptar cualquier etiqueta (cualquier petición de sentarse o levantarse), sólo pueda aceptar peticiones de levantamiento.

Estas etiquetas de las que hablamos serán `etiq_pedir_asiento` (1) y `etiq_liberar_asiento` (2).

Las etiquetas además aparecerán en la función de los filósofos. Estos ahora al haber un nuevo proceso deberán de alterar la manera en la que se asignan las IDs de los tenedores que necesitan. Además, en cada iteración del bucle se realizará una petición de asiento y la liberación del mismo mediante el envío de mensajes al camarero.

Afortunadamente, la función de los tenedores se mantiene igual.

Y, al haber añadido una nueva función, tenemos que lanzarla desde el main. El id correspondiente sería el cero.

Código Fuente

```
// -----------------------------------------------------------------------------
//
// Sistemas concurrentes y Distribuidos.
// Práctica 3. Implementación de algoritmos distribuidos con MPI
//
// Archivo: filosofos-cam.cpp
// Implementación del problema de los filósofos (con camarero).
//
// -----------------------------------------------------------------------------


#include <mpi.h>
#include <thread> // this_thread::sleep_for
#include <random> // dispositivos, generadores y distribuciones aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include <iostream>

using namespace std;
using namespace std::this_thread ;
using namespace std::chrono ;

const int
num_filosofos = 5 ,
  num_procesos  = 2*num_filosofos + 1,
  etiq_pedir_asiento = 1,
  etiq_liberar_asiento = 2;


//**************************************************************************
// plantilla de función para generar un entero aleatorio uniformemente
// distribuido entre dos valores enteros, ambos incluidos
// (ambos tienen que ser dos constantes, conocidas en tiempo de compilación)
//--------------------------------------------------------------------------

template< int min, int max > int aleatorio()
{
  static default_random_engine generador( (random_device())() );
  static uniform_int_distribution<int> distribucion_uniforme( min, max ) ;
  return distribucion_uniforme( generador );
}

// -----------------------------------------------------------------------------

void funcion_filosofos( int id )
{
```

```
  int id_ten_izq = (id+1) % num_procesos, //id. tenedor izq.
    id_ten_der = (id-1); //id. tenedor der.

  if(id_ten_izq == 0) id_ten_izq = 1; // En caso de que sea el último filósofo, ya no correspo

  int id_camarero = 0;

  int aux = 0; // No es usado, pero es necesario algún buffer para transmitir los mensajes
  while ( true )
    {
      // Se añade a las peticiones anteriores, la petición de sentarse
      cout << "Filósofo " << id << " solicita sentarse " << endl;
      MPI_Ssend(&aux, 1, MPI_INT, id_camarero, etiq_pedir_asiento, MPI_COMM_WORLD);

      // Procedimiento original
      cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

      cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);

      cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
      sleep_for( milliseconds( aleatorio<10,100>() ) );

      cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD);

      cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
      MPI_Ssend( &aux, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD);

      // Tras soltar ambos tenedores el filósofo envía la petición de levantarse
      cout <<"Filósofo " << id << " solicita levantarse" <<endl;
      MPI_Ssend(&aux, 1, MPI_INT, id_camarero, etiq_liberar_asiento, MPI_COMM_WORLD);

      cout << "Filósofo " << id << " comienza a pensar" << endl;
      sleep_for( milliseconds( aleatorio<10,100>() ) );

    }
}
// -----------------------------------------------------------------------

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

  int aux;
  while ( true )
```

```
    {
      MPI_Recv(&aux,1,MPI_INT,MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &estado);
      id_filosofo = estado.MPI_SOURCE;
      cout <<"\t\tTen. " <<id <<" ha sido cogido por filo. " <<id_filosofo <<endl;

      MPI_Recv(&aux, 1, MPI_INT, id_filosofo, 0, MPI_COMM_WORLD, &estado);

      cout <<"\t\tTen. "<< id<< " ha sido liberado por filo. " <<id_filosofo <<endl ;
    }
}


void funcion_camarero(){
  int aux, etiq_aceptable; // Buffer auxiliar y etiqueta receptible
  int filosofos_sentados = 0;

  MPI_Status estado ;       // metadatos de las dos recepciones

  while ( true ){
    if (filosofos_sentados < num_filosofos-1) {
      etiq_aceptable = MPI_ANY_TAG;
    } else {
      etiq_aceptable = etiq_liberar_asiento;
    }

    MPI_Recv(&aux, 1, MPI_INT, MPI_ANY_SOURCE, etiq_aceptable, MPI_COMM_WORLD, &estado);

    if (estado.MPI_TAG == etiq_pedir_asiento) {
      filosofos_sentados++;
      cout <<"\tCamarero recibe petición de sentamiento del filósofo " << estado.MPI_SOURCE <<
    } else if (estado.MPI_TAG == etiq_liberar_asiento) {
      filosofos_sentados--;
      cout <<"\tCamarero recibe petición levantamiento del filosofo " << estado.MPI_SOURCE << '
    }
  }
}
// ---------------------------------------------------------------------

int main( int argc, char** argv )
{
  int id_propio, num_procesos_actual ;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );


  if ( num_procesos == num_procesos_actual )
```

```
    {
      if(id_propio == 0)
    funcion_camarero();
      // ejecutar la función correspondiente a 'id_propio'
      else if ( id_propio % 2 == 0 )          // si es par
    funcion_filosofos( id_propio ); //   es un filósofo
      else                                    // si es impar
    funcion_tenedores( id_propio ); //   es un tenedor
    }
  else
    {
      if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
    { cout << "el número de procesos esperados es:    " << num_procesos << endl
           << "el número de procesos en ejecución es: " << num_procesos_actual << endl
           << "(programa abortado)" << endl ;
    }
    }

  MPI_Finalize( );
  return 0;
}

// ----------------------------------------------------------------------
```

Salida del programa

Un fragmento de la ejecución (que es infinita) devuelve el siguiente patrón:

```
mpirun -np 11 ./filosofos-cam
Filósofo 2 solicita sentarse
Filósofo Filósofo 8 solicita sentarse
Filósofo 10 solicita sentarse
4 solicita sentarse
Filósofo 2 solicita ten. izq.3
    Camarero recibe petición de sentamiento del filósofo 2. Total sentados: 1
    Camarero recibe petición de sentamiento del filósofo 4. Total sentados: 2
    Camarero recibe petición de sentamiento del filósofo 8. Total sentados: 3
    Camarero recibe petición de sentamiento del filósofo 10. Total sentados: 4
Filósofo 2 solicita ten. der.1
Filósofo 10 solicita ten. izq.1
        Ten. Filósofo 2 comienza a comer
        Ten. 3 ha sido cogido por filo. 21 ha sido cogido por filo. 2

Filósofo 6 solicita sentarse
Filósofo 4 solicita ten. izq.5
Filósofo 4 solicita ten. der.3
        Ten. 5 ha sido cogido por filo. 4
Filósofo 8 solicita ten. izq.9
        Ten. 9 ha sido cogido por filo. 8
Filósofo 8 solicita ten. der.7
Filósofo 8 comienza a comer
        Ten. 7 ha sido cogido por filo. 8
    Camarero recibe petición levantamiento del filosofo 2. Total sentados: 3
    Camarero recibe petición de sentamiento del filósofo 6. Total sentados: 4
        Ten. 1 ha sido liberado por filo. 2
        Ten. 1 ha sido cogido por filo. 10
Filósofo 2 suelta ten. izq. 3
Filósofo 2 suelta ten. der. 1
Filósofo 2 solicita levantarse
Filósofo 2 comienza a pensar
        Ten. 3 ha sido liberado por filo. 2
        Ten. 3 ha sido cogido por filo. 4
Filósofo 10 solicita ten. der.9
Filósofo 6 solicita ten. izq.7
Filósofo 4 comienza a comer
Filósofo 8 suelta ten. izq. 9
Filósofo 8 suelta ten. der. 7
Filósofo   Camarero recibe petición levantamiento del filosofo 8. Total sentados: 3
        Ten. 7 ha sido liberado por filo. 8
        Ten. 7 ha sido cogido por filo. 6
8 solicita levantarse
Filósofo 8 comienza a pensar
```

```
Filósofo 6 solicita ten. der.5
        Ten. 9 ha sido liberado por filo. 8
        Ten. Filósofo 10 comienza a comer
```

Analizando esto vemos cómo cada proceso que quiere sentarse envía una petición al camarero, que los va sentando, pero nunca supera los cuatro filósofos sentados.