PORTAFOLIOS PRÁCTICA 3

1. Múltiples productores y consumidores

El primer cambio es en el main. Ahora no tenemos id_productor ni id_consumidor, sino que tenemos un rango. Por ello, si el id_propio es menor que el id del proceso que gestiona el buffer, entonces se lanza la función productora. Si el id es id_buffer (definido como cte. global), entonce se lanza la función del buffer. Si el id_propio es mayor que id_buffer, entonces se trata de una hebra que debe lanzar la función consumidora. Para poder enumerar las hebras consumidoras empezando en 0, habrá que restar el número de hebras productoras más la hebra que gestiona el buffer (así pasamos el parámetro correcto a funcion_consumidor.

El segundo cambio es en funcion_buffer. Ahora ya no tenemos un único productor y un único consumidor, por lo que ya no podemos trabajar con los id para hacer los receive y send. Por ello, definimos dos nuevas etiquetas (enteros) globales que indican si el mensaje lo ha enviado un productor o un consumidor. Dependiendo del estado del buffer, guardamos en tag las etiquetas que deben tener los mensajes que se pueden recibir. Si está lleno, tag es igual a etiq_cons; si está vacío, tag es igual a etiq_prod; y si no está vacío ni lleno, la etiqueta del mensaje a recibir puede ser cualquiera. Por último, después de recibir el mensaje, recuperamos la etiqueta del mensaje mediante la variable estado y en función de su valor, actualizamos el buffer escribiendo o borrando un valor. Si el mensaje recibido era de un consumidor, se le envía el valor a consumir usando estado.MPI_SOURCE.

Por último, en funcion_productor y funcion_consumidor he cambiado las etiquetas de los mensajes que envían a etiq_prod y etiq_cons, respectivamente.

Código

```
#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_items          = 60,
  tam_vector         = 10,
  num_prod                  = 4 ,
```

```
   num_cons                    = 5 ,
   num_procesos_esperado = num_cons + num_prod + 1,
   id_buffer             = num_prod ,
   etiq_cons             = 0 ,
   etiq_prod             = 1 ;

int contador[num_prod] = {0};

//
**********************************************************************
**
// 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 );
}
// ---------------------------------------------------------------------
// ptoducir produce los numeros en secuencia (1,2,3,....)
// y lleva espera aleatorio
int producir(int orden)
{

   int resultado =  orden*(num_items/num_prod) + contador[orden];
   sleep_for( milliseconds( aleatorio<10,100>()) );
   contador[orden]++ ;
   cout << "Productor " << orden << " ha producido valor " <<
   resultado << endl << flush;
   return resultado;
}
// ---------------------------------------------------------------------

void funcion_productor(int orden)
{
   for ( unsigned int i = 0 ; i < num_items/num_prod ; i++ )
   {
      // producir valor
      int valor_prod = producir(orden);
      // enviar valor
```

```
      cout << "Productor " << orden << " va a enviar valor " << valor_prod << endl << flush;
                  MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, etiq_prod, MPI_COMM_WORLD );
    }
}
// ----------------------------------------------------------------------

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

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

   for( unsigned int i = 0 ; i < num_items/num_cons; i++ )
   {
                  MPI_Ssend( &peticion,      1, MPI_INT, id_buffer, etiq_cons, MPI_COMM_WORLD);
                        MPI_Recv ( &valor_rec, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD,&estado );
       cout << "Consumidor " << orden << " ha recibido valor " << valor_rec << endl << flush ;
      consumir( valor_rec , 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
              tag ;                    // 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
      tag = etiq_prod ;                          // $~~~$ solo prod.
    else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
      tag = etiq_cons ;                          // $~~~$ solo cons.
    else                                  // si no vacío ni lleno
      tag = MPI_ANY_TAG ;     // $~~~$ cualquiera

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

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

    // 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 << "Buffer 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 << "Buffer va a enviar valor " << valor << endl ;
                      MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE, 0,
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_buffer )
      funcion_productor(id_propio);
    else if ( id_propio == id_buffer )
      funcion_buffer();
    else{
      funcion_consumidor(id_propio-(num_prod+1));
    }
  }
  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;
}
```

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/Tercero/scd-p3-fuentes$ mpirun -oversubscribe -np 10 ./prodconsMPI
Productor 2 ha producido valor 30
Productor 2 va a enviar valor 30
Buffer ha recibido valor 30
Buffer va a enviar valor 30
Consumidor 0 ha recibido valor 30
Productor 1 ha producido valor 15
Productor 1 va a enviar valor 15
Buffer ha recibido valor 15
Buffer va a enviar valor 15
Consumidor 1 ha recibido valor 15
Productor 2 ha producido valor 31
Productor 2 va a enviar valor 31
Buffer ha recibido valor 31
Buffer va a enviar valor 31
Consumidor 2 ha recibido valor 31
Productor 0 ha producido valor 0
Productor 0 va a enviar valor 0
Buffer ha recibido valor 0
Buffer va a enviar valor 0
Consumidor 3 ha recibido valor 0
Productor 2 ha producido valor 32
Productor 2 va a enviar valor 32
Buffer ha recibido valor 32
Buffer va a enviar valor 32
Consumidor 4 ha recibido valor 32
```

2. Filósofos sin camarero

El interbloqueo que se puede producir es que justo todos cojan el tenedor a su izquierda. De esta forma, al intentar coger el tenedor derecho, se quedarían bloqueados esperando a que se libere, lo cual nunca ocurrirá porque todos están esperándose entre sí y ninguno avanzará.

La forma en que se me ha ocurrido solucionarlo es haciendo que uno de ellos coja los tenedores en orden inverso, es decir, coge primero el derecho y luego el izquierdo.

En cuanto al código, solo he completado las partes indicadas. En funcion_filosofos he hecho los respectivos MPI_Ssend para obtener los tenedores (por lo que en cada caso he puesto el id del proceso tenedor al que se le envía el mensaje) y luego otros dos MPI_Ssend para liberarlos.

En el código de funcion_tenedores, se hacen dos MPI_Recv. El primero es para recibir la solicitud por alguno de los dos filósofos que lo pueden usar y el segundo para recibir el mensaje de liberación por parte del filósofo que lo hubiese cogido. La única complejidad que hay es que tras el primer mensaje, la forma de obtener el id del filósofo que lo ha cogido es usando estado.MPI_SOURCE, lo cual usaremos luego para el segundo MPI_Recv.

Código

```
#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 ,              // número de filósofos
   num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
    num_procesos  = num_filo_ten ;    // número de procesos total (por ahora solo hay
filo y ten)


//
**********************************************************************
**
// 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_filo_ten, //id. tenedor izq.
      id_ten_der = (id+num_filo_ten-1) % num_filo_ten; //id. tenedor der.

  int valor = 1;

  while ( true )
  {

    if(id == 0){

          cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
        // ... solicitar tenedor derecho (completar)
        MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD );
```

```
      cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
      // ... solicitar tenedor izquierdo (completar)
      MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD );

   }
   else{

      cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
      // ... solicitar tenedor izquierdo (completar)
      MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD );

      cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
      // ... solicitar tenedor derecho (completar)
      MPI_Ssend( &valor, 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;
   // ... soltar el tenedor izquierdo (completar)
   MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD );

   cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
   // ... soltar el tenedor derecho (completar)
   MPI_Ssend( &valor, 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

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

                MPI_Recv ( &valor, 1, MPI_INT, MPI_ANY_SOURCE, 0,
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, 0, MPI_COMM_WORLD,
&estado );

    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;
}
```

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/Tercero/scd-p3-fuentes$ mpicxx -std=c++11 -o filosofos-interb filosofos-interb.cpp
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/Tercero/scd-p3-fuentes$ mpirun -oversubscribe -np 10 ./filosofos-interb
Filósofo 2 solicita ten. izq.3
Filósofo 2 solicita ten. der.1
Filósofo 8 solicita ten. izq.9
Ten. 3 ha sido cogido por filo. 2
Filósofo 4 solicita ten. izq.5
Filósofo 8 solicita ten. der.7
Ten. 5 ha sido cogido por filo. 4
Filósofo 8 comienza a comer
Ten. 7 ha sido cogido por filo. 8
Filósofo 6 solicita ten. izq.7
Filósofo 0 solicita ten. der.9
Ten. 1 ha sido cogido por filo. 2
Filósofo 4 solicita ten. der.3
Filósofo 2 comienza a comer
Ten. 9 ha sido cogido por filo. 8
Filósofo 8 suelta ten. izq. 9
Ten. 9 ha sido liberado por filo. 8
Ten. 9 ha sido cogido por filo. 0
Filósofo 8 suelta ten. der. 7
Filósofo 0 solicita ten. izq.1
Filosofo 8 comienza a pensar
Ten. 7 ha sido liberado por filo. 8
Ten. 7 ha sido cogido por filo. 6
```

3. Filósofos con camarero

Ahora el camarero es el que controla que no se den interbloqueos haciendo que no haya más de 4 filósofos sentados simultáneamente. Uno de los cambios se da en funcion_filosofos ya que ahora tienen que solicitar sentarse, además de levantarse después de soltar los tenedores. Para ello, se hacen dos nuevos MPI_Ssend; el primero para sentarse y el último para levantarse. Para poder identificar desde el proceso camarero si nos están pidiendo sentarse o levantarse, definimos dos etiquetas globales.

En la función funcion_camarero se controla qué tipo de mensajes se pueden recibir dependiendo de cuántos filósofos están sentados. Esto se controla con una variable contador. Si esta es menor que 4 (num_filosofos-1), la etiqueta de los mensajes que se pueden recibir es cualquiera, mientras que si no lo es entonces solo se permite recibir mensajes para levantarse (que llevarán la etiqueta etiq_levantarse). Después de recibir el mensaje, recuperamos la etiqueta del mensaje con estado.MPI_TAG para poder actualizar la variable contador como corresponde.

Por último, en el main hemos añadido el caso de que id_propio sea igual a id_camarero, el cual hemos definido como variable global. En ese caso, se llama a funcion_camarero ya explicada.

Código

```
#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 ,              // número de filósofos
   num_filo_ten  = 2*num_filosofos, // número de filósofos y tenedores
    num_procesos  = num_filo_ten + 1 ,   // número de procesos total (por ahora solo
hay filo y ten)
   id_camarero = 10,
   etiq_levantarse = 0,
   etiq_sentarse = 1;



//
**********************************************************************
**
// 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_filo_ten, //id. tenedor izq.
      id_ten_der = (id+num_filo_ten-1) % num_filo_ten; //id. tenedor der.

  int valor = 1;

  while ( true )
  {

    cout << "Filósofo " <<id << " solicita sentarse " << endl;
                MPI_Ssend(  &valor,  1,  MPI_INT,  id_camarero,  etiq_sentarse,
MPI_COMM_WORLD );
```

```
   cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
   // ... solicitar tenedor izquierdo (completar)
   MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD );

   cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
   // ... solicitar tenedor derecho (completar)
   MPI_Ssend( &valor, 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;
   // ... soltar el tenedor izquierdo (completar)
   MPI_Ssend( &valor, 1, MPI_INT, id_ten_izq, 0, MPI_COMM_WORLD );

   cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
   // ... soltar el tenedor derecho (completar)
   MPI_Ssend( &valor, 1, MPI_INT, id_ten_der, 0, MPI_COMM_WORLD );

   cout << "Filósofo " <<id << " solicita levantarse " << endl;
            MPI_Ssend( &valor, 1, MPI_INT, id_camarero, etiq_levantarse,
MPI_COMM_WORLD );

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

void funcion_camarero(){

  int contador = 0,
  valor,
  etiq_permitida;

  MPI_Status estado;

  while( true )
  {
        if(contador < num_filosofos-1){
              etiq_permitida = MPI_ANY_TAG;
        }
        else
              etiq_permitida = etiq_levantarse;
```

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

        switch( estado.MPI_TAG ){

                case etiq_levantarse:
                        contador--;
                        break;
                case etiq_sentarse:
                        contador++;
                        break;
        }
  }

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

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, 0, 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, 0, MPI_COMM_WORLD, &estado );

    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 == id_camarero )
      funcion_camarero();               // es el camarero
    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;
}
```

```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/Tercero/scd-p3-fuentes$ mpicxx -std=c++11 -o filosofos-cam filosofos-cam.cpp
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/Tercero/scd-p3-fuentes$ mpirun -oversubscribe -np 11 ./filosofos-cam
Filósofo 2 solicita sentarse
Filósofo 4 solicita sentarse
Filósofo 4 solicita ten. izq.5
Filósofo 6 solicita sentarse
Filósofo 8 solicita sentarse
Filósofo 2 solicita ten. izq.3
Filósofo 6 solicita ten. izq.7
Filósofo 0 solicita sentarse
Filósofo 4 solicita ten. der.3
Ten. 5 ha sido cogido por filo. 4
Ten. 7 ha sido cogido por filo. 6
Filósofo 2 solicita ten. der.1
Filósofo 2 comienza a comer
Filósofo 6 solicita ten. der.5
Ten. 1 ha sido cogido por filo. 2
Ten. 3 ha sido cogido por filo. 2
Filósofo 0 solicita ten. izq.1
Filósofo 2 suelta ten. izq. 3
Filósofo 2 suelta ten. der. 1
Filósofo 2 solicita levantarse
Ten. 3 ha sido liberado por filo. 2
Ten. 3 ha sido cogido por filo. 4
```

José Alberto Hoces Castro 3ºDGIIM