UNIVERSIDAD DE GRANADA

Sistemas Concurrentes y Distribuidos

Portafolios de prácticas

Contiene las prácticas del curso

Javier Sáez de la Coba
2º A2

Curso 2017-2018

PDF generado el 11 de diciembre de 2017

Índice

1. Bloque 1

1.1. Seminario 1

1.1.1. Cálculo de una integral usando programación concurrente

Se ha optado por la solución contigua, ya que simplifica los cálculos y la implementación del for que corre en cada hebra.

El valor resultado es:

```
g++ -pthread ejemplo09-plantilla.cpp -o ejemplo9 --std=c++11
javier@javier-zenbook:~/D/2/S/Seminario1
  ./ejemplo9
Número de muestras (m)   : 1073741824
Número de hebras (n)     : 4
Valor de PI              : 3.14159265358979312
Resultado secuencial     : 3.14159265358998185
Resultado concurrente    : 3.14159265172718216
Tiempo secuencial        : 19261 milisegundos.
Tiempo concurrente       : 5434.5 milisegundos.
Porcentaje t.conc/t.sec. : 28.21%
```

Se puede apreciar la correlación entre el número de hebras y la mejora del tiempo de ejecución.

El código que ha generado este resultado es el siguiente:

```
double funcion_hebra( long id )
{
  long inferior = (m*(id-1))/n +1;
  long superior = (m*(id))/n;
  double semisuma = 0.0;
  for (long i = inferior; i < superior; i++) {
    semisuma += f( (i+double(0.5)) /m );
  }
  return semisuma;
}
```

```
double calcular_integral_concurrente( )
{
  future<double> hebras[n];
  double suma = 0.0;
```

```
  for (int i = 0; i < n; i++) {
    hebras[i] = async( launch::async, funcion_hebra, i+1);
  }

  for (int i = 0; i < n; i++) {
    suma += hebras[i].get();
  }

  return suma/m;

}
```

1.2. Práctica 1

1.2.1. Problema del productor consumidor

En este apartado vamos a resolver los ejercicios relacionados con el problema del productor consumidor (versión LIFO). En esta versión de la solución, se utiliza un único buffer que es leido como una pila. De ese modo, el último elemento producido e insertado en el buffer es el primero que es leido.

Se ha intentado implementar la solución FIFO pero no ha sido posible debido a errores en el uso del buffer circular.

Para la solución implementada, necesitamos varias variables compartidas: dos semáforos (uno de lectura y otro de escritura), el propio vector de elementos que actúa de buffer de lectura/escritura y una variable de tipo entero que indica cual es la primera posición libre en el vector usado.

```
int buffer[tam_vec-1]; //Vector de elementos producidos
int primera_libre = 0; //Indice de posición libre del vector
Semaphore libres = tam_vec; //Semáforo que bloquea la escritura
Semaphore ocupadas = 0; //Semáforo que bloquea la lectura
```

Para determinar la posición en la que leer o escribir se hace la siguiente operación:

- La hebra productora escribe en la posición guardada por `primera_libre`
- La hebra productora lee el buffer en la posición determinada por `primera_libre` -1

Así mismo, los dos semáforos: `libres` y `ocupadas` tienen los valores iniciales `tam_vec` y 0 respectivamente. De este modo, la hebra productora consulta el semáforo `libres` que tiene como valor el número de casillas libres del vector. En caso de que estuviera lleno, el valor del

semáforos sería 0 y bloquearía la hebra, que se desbloquearía cuando la hebra consumidora lea un valor del vector y haga un `sem_signal` sobre el semáforo.

De manera similar funciona el semáforo `ocupadas`, cuyo valor corresponde con los datos producidos y listos para consumir del vector. De manera que la función de la hebra consumidora hace un `sem_wait` sobre el semáforo de casillas ocupadas a la espera que la hebra productora introduzca algún valor.

A continuación se presenta el código de las hebras productoras y consumidoras.

```
void  funcion_hebra_productora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato = producir_dato() ;
      sem_wait(libres);
      buffer[primera_libre] = dato;
      primera_libre++;
      sem_signal(ocupadas);
   }
}
```

```
void funcion_hebra_consumidora(  )
{
   for( unsigned i = 0 ; i < num_items ; i++ )
   {
      int dato ;
      sem_wait(ocupadas);
      dato = buffer[primera_libre - 1];
      primera_libre--;
      sem_signal(libres);
      consumir_dato( dato ) ;
    }
}
```

1.2.2. Problema de los fumadores

Para la solución del problema de los fumadores (variante del productor-consumidor) se ha recurrido a los siguientes semáforos:

- `mostrador_vacio`. Como únicamente puede haber un ingrediente en el mostrador y el estanquero no puede poner otro hasta que un fumador utiliza el producido se usa un semáforo cuyo valor inicial es 1 y sobre el que la hebra del estanquero hace un `sem_wait`. Las hebras de los fumadores son las que hacen un `sem_signal` sobre el semáforo.

- `ingredientes`: Es un array de semáforos, uno por cada ingrediente posible (y por tanto, según las condiciones del problema, por cada fumador. Especifica que ingredientes hay disponibles. Así cada fumador que necesite un ingrediente específico, hace un `sem_wait` sobre el semáforo de dicho ingrediente y se pone a la cola de espera. El estanquero hace un `sem_signal` sobre el semáforo del ingrediente que acaba de producir (y poner en el mostrador) en cada iteración.

Las variables compartidas implementadas han sido:

```
Semaphore ingredientes[3] = {0,0,0};
Semaphore mostrador_vacio = 1; //El mostrador empieza vacío
```

Las distintas funciones implementadas han sido las siguientes:

Hebra del estanquero:

```
void funcion_hebra_estanquero(  )
{
  while (true) {
    int ingrediente = Producir();
    sem_wait(mostrador_vacio);
    cout << "Puesto ingrediente " << ingrediente << endl;
    sem_signal(ingredientes[ingrediente]);
  }
}
```

Hebra del fumador:

```
void  funcion_hebra_fumador( int num_fumador )
{
   while( true )
   {
     sem_wait(ingredientes[num_fumador]);
     cout << "          Retirado ingrediente " << num_fumador <<
        endl;
     sem_signal(mostrador_vacio);
     fumar(num_fumador);
   }
}
```

Función de producir un ingrediente:

```
int Producir(){
  // calcular milisegundos aleatorios de duración de producir)
  chrono::milliseconds dur_producir( aleatorio<20,200>() );
  this_thread::sleep_for( dur_producir ); //Esperar
```

```
  int resultado = aleatorio<0,2>();
  cout << "Estanquero produce ingrediente " << resultado << endl;
  return resultado;
}
```

2. Bloque 2

2.1. Seminario 2

2.1.1. Actividades relativas al monitor Barrera1 (SC)

Se pasa a describir tres hechos:

La hebra que entra la última al método cita (la hebra señaladora) es siempre la primera en salir de dicho método.

Este comportamiento se debe a la semántica del monitor, que es Señalar y Continuar, esto es que la hebra señaladora termina su ejecución. Por eso, la última hebra, que es la que despierta al resto de hebras sale primero del monitor, porque nunca espera nada.

El orden en el que las hebras señaladas logran entrar de nuevo al monitor no siempre coincide con el orden de salida de wait (se observa porque los números de orden de entrada no aparecen ordenados a la salida).

Los procesos despertados con `notify_one()` se ponen en una cola dentro del monitor para recuperar el cerrojo. Esta cola tiene cierta aleatoriedad.

El constructor de la clase no necesita ejecutarse en exclusión mutua.

Esto es porque el constructor del monitor no se ejecuta desde ninguna hebra, sino desde el `main`. Por ello no es necesaria la exclusión mutua.

2.1.2. Propiedades de la barrera parcial con semántica SU

Describe razondamente en tu portafolio a que se debe que ahora, con la semántica SU, se cumplan las dos propiedades descritas.

El orden de salida de la cita coincide siempre con el orden de entrada

Esto se debe a que cuando se hace `signal` se pasa el cerrojo directamente a la hebra que más tiempo llevaba esperando en la cola.

Hasta que todas las hebras de un grupo han salido de la cita, ninguna otra hebra que no sea del grupo logra entrar.

Esto se debe a que el código del monitor se ejecuta en exclusión mutua, pero como están saliendo las hebras debido a la naturaleza del `signal` no se puede ejecutar ninguna otra hebra dentro del monitor, por lo que no puede entrar al método `cita` y meterse en cola.

2.2. Práctica 2

2.2.1. Problema de los fumadores con monitor SU

Variables condición y colas de espera del monitor `Estanco`.

$suministro : integer$ ▷ Guarda el producto que está en el mostrador.
$fumador[num\_fum] : condition\ array$ ▷ Distintas colas para los distintos tipos de fumadores
$estanquero : condition$ ▷ Cola para la espera del estanquero

Pseudocódigo para los tres procedimientos del monitor.

procedure PONERINGREDIENTE($integer : ingrediente$) ▷ Ejecutado por la hebra Estanquero
  $suministro \leftarrow ingrediente$
  $fumador[ingrediente].signal()$
end procedure

procedure ESPERARRECOGIDA ▷ Ejecutado por la hebra estanquero
  if $suministro \neq -1$ **then** ▷ Si no se ha recogido el ingrediente anterior, esperar
    $estanquero.wait$
  end if
end procedure

procedure OBTENERINGREDIENTE($integer : num\_fumador$) ▷ Ejecutado por la hebra fumador
  if $suministro \neq num\_fumador$ **then** ▷ Si el ingrediente no es el necesario, esperar
    $fumador[num\_fumador].wait$
  end if
  $suministro \leftarrow -1$ ▷ Reiniciar el mostrador
  $estanquero.signal$
end procedure

Código fuente en C++ del monitor

```
class Estanco : public HoareMonitor
{
private:
  int suministro;
  CondVar fumador[num_fum] ;
```

```
  CondVar estanquero ;

public:                          // constructor y métodos públicos
  Estanco(  ) ;                // constructor
  void  ponerIngrediente( int ingrediente );
  void  obtenerIngrediente( int n_fumador );
  void  esperarRecogida();
} ;


Estanco::Estanco(  )
{
  suministro = -1;

  estanquero = newCondVar();

  for (int i = 0; i<num_fum; ++i)
    fumador[i] = newCondVar();
}

void Estanco::ponerIngrediente( int ingrediente )
{
  suministro = ingrediente;
  fumador[ingrediente].signal();
}

void Estanco::esperarRecogida()
{
  if( suministro != -1 )
    estanquero.wait();
}

void Estanco::obtenerIngrediente( int n_fumador )
{
  if( suministro != n_fumador )
    fumador[n_fumador].wait();
  suministro = -1;
  estanquero.signal();
}
```

Ejemplo de la salida del programa:

```
  ./fumadores
-------------------------------------------------------
Problema de los fumadores.
```

```
------------------------------------------------------
Estanquero: pone ingrediente 0
Fumador 0 retira el producto.
Estanquero: pone ingrediente 1
Fumador 1 retira el producto.
Fumador 0  : empieza a fumar (36 milisegundos)
Estanquero: pone ingrediente 0
Fumador 1  : empieza a fumar (78 milisegundos)
Fumador 0  : termina de fumar, comienza espera de ingrediente.
Estanquero: pone ingrediente 2
Fumador 0 retira el producto.
Fumador 0  : empieza a fumar (124 milisegundos)
Fumador 2 retira el producto.
Fumador 2  : empieza a fumar (147 milisegundos)
Estanquero: pone ingrediente 2
Fumador 1  : termina de fumar, comienza espera de ingrediente.
Fumador 0  : termina de fumar, comienza espera de ingrediente.
Fumador 2  : termina de fumar, comienza espera de ingrediente.
Fumador 2 retira el producto.
Fumador 2  : empieza a fumar (174 milisegundos)
Estanquero: pone ingrediente 0
Fumador 0  : retira el producto.
.
.
.
```

2.2.2. Simulación de una barbería con monitor SU

Variables condicion usadas en el monitor:

$gentePelandose : boolean$ ▷ Variable para ver si el barbero está pelando a alguien.

$cliente : condition$ ▷ Cola de espera de los clientes. El wait lo ejecutan los clientes cuando el barbero está ocupado. El signal lo ejecuta el barbero cuando termina de pelar a un cliente para llamar al siguiente.

$barbero : condition$ ▷ Cola para la espera del barbero. El wait lo ejecuta el barbero cuando no hay más clientes esperando. El signal lo ejecutan los clientes cuando entran en la barbería para despertar al barbero.

Código en C++11 del programa de simulación de barbería.

```cpp
#include <iostream>
#include <cassert>
#include <thread>
#include <mutex>
#include <random> // dispositivos, generadores y distribuciones
```

```
   aleatorias
#include <chrono> // duraciones (duration), unidades de tiempo
#include "HoareMonitor.hpp"
#include <vector>


using namespace std ;
using namespace HM ;

mutex mtx; //candado de escritura en pantalla

const int num_clientes = 10;


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 );
}

//**************************************************
// Monitor que representa la barbería

class Barberia : public HoareMonitor
{
private:
  bool gentePelandose;
  CondVar barbero;
  CondVar cliente;

public:
  Barberia(  );
  void avisarCliente();
  void avisarBarbero();
  void esperarBarbero();
  void esperarCliente();
} ;


Barberia::Barberia(  )
{
  gentePelandose = false;
  barbero = newCondVar();
```

```
  cliente = newCondVar();
}

void Barberia::avisarCliente() //Ejecuta Barbero
{
  cliente.signal();
  gentePelandose = false;
}

void Barberia::avisarBarbero() //Ejecuta cliente
{
  barbero.signal();
}

void Barberia::esperarBarbero //Ejecuta cliente
{
  if ( gentePelandose )
    cliente.wait();
}

void Barberia::esperarCliente() //Ejecuta barbero
{
  if ( cliente.empty() )
    barbero.wait();

  gentePelandose = true;
}
//
   ----------------------------------------------------------------------



//
   **********************************************************************

//función que simula la acción de pelar.

void cortarPelo(  )
{
  //Calcula cuando va a tardar en pelar
  chrono::milliseconds duracion_pelado( aleatorio<20,200>() );

  //Informa de las acciones del barbero

  mtx.lock();
  cout << "Barbero afeita cliente." << endl;
```

```
  mtx.unlock();

  this_thread::sleep_for( duracion_pelado );
}
//
   ----------------------------------------------------------------------



//Función para poner a los clientes en bucle infinito
void crecerPelo ( int n_client )
{
  //Calcula cuando va a tardar en pelar
  chrono::milliseconds duracion_crecimiento( aleatorio<20,200>()
     );

  this_thread::sleep_for( duracion_crecimiento );

  mtx.lock();
  cout << "Cliente: " << n_client << " \"Ya me creció el pelo\""
     << endl;
  mtx.unlock();
}

//
   ************************************************************************

//Funciones para las hebras


void funcion_hebra_barbero( MRef<Barberia> monitor )
{
  while( true )
    {
      monitor->esperarCliente();
      cortarPelo();
      monitor->avisarCliente();
    }
}


void funcion_hebra_cliente( MRef<Barberia> monitor, int i )
{
  while(true){
    monitor->avisarBarbero();
```

```
    monitor->esperarBarbero();
    cout << "Cliente : " << i << " se ha cortado el pelo." <<
       endl;
    crecerPelo(i);
  }
}



int main()
{
  cout << "
    ------------------------------------------------------------" <<
     endl
      << "Problema de la barbería" << endl
      << "
         ------------------------------------------------------------
         " << endl
      << flush ;

  auto monitor = Create<Barberia>( );

  thread hebra_barbero( funcion_hebra_barbero, monitor );
  thread hebra_cliente[num_clientes];

  for ( int i = 0; i < num_clientes; ++i)
    hebra_cliente[i] = thread ( funcion_hebra_cliente, monitor, i
       );

  hebra_barbero.join();

  for ( int i = 0; i < num_clientes; ++i)
    hebra_cliente[i].join();

  cout << "
    ------------------------------------------------------------" <<
     endl
      << "FIN"
      << "
         ------------------------------------------------------------
         " << endl;
}
```

Ejemplo de salida del programa:

```
  ./barberia
------------------------------------------------------------
Problema de la barbería
------------------------------------------------------------
Barbero afeita cliente.
Cliente : Barbero afeita cliente.
2 se ha cortado el pelo.
Cliente : 0 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 0 "Ya me creció el pelo"
Cliente: 2 "Ya me creció el pelo"
Cliente : 1 se ha cortado el pelo.
Barbero afeita cliente.
Cliente : 5 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 1 "Ya me creció el pelo"
Cliente : 3 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 5 "Ya me creció el pelo"
Cliente : 6 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 6 "Ya me creció el pelo"
Cliente : 4 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 3 "Ya me creció el pelo"
Cliente : 8 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 4 "Ya me creció el pelo"
Cliente: 8 "Ya me creció el pelo"
Cliente : 7 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 7 "Ya me creció el pelo"
Cliente : 9 se ha cortado el pelo.
Barbero afeita cliente.
Cliente: 9 "Ya me creció el pelo"
Cliente : 0 se ha cortado el pelo.
.
.
.
```

3. Bloque 3

3.1. Práctica 3

3.1.1. Problema del productor-consumidor con buffer intermedio

Para poder solucionar este problema, se han realizado los siguientes cambios:

- Adaptar la selección de rol según el ID del proceso.
- Calcular los numeros de orden dentro de cada rol, siendo los productores id_propio y los consumidores $id\_propio - num\_productores - 1$
- Modificar la función producir dato para que tenga en cuenta el rango en el que produce.
- Modificar la función buffer para que identifique los mensajes por etiquetas en vez de por ID's de proceso.

Código fuente de la práctica:

```
#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
   id_buffer              = 4 ,
   num_productores        = 4 ,
   num_consumidores       = 5 ,
   num_procesos_esperado  = 10 ,
   num_items              = 20,
   tam_vector             = 10;

const int
  etiq_prod      = 1,
  etiq_cons      = 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 );
}
//
   ----------------------------------------------------------------------

// ptoducir produce los numeros en secuencia (1,2,3,....)
// y lleva espera aleatorio
int producir(int num_productor)
{
   static int k = num_items/num_productores;
   static int contador =  num_productor*k;
   sleep_for( milliseconds( aleatorio<10,100>()) );
   contador++ ;
   cout << "Productor ha producido valor " << contador << endl <<
       flush;
   return contador ;
}
//
   ----------------------------------------------------------------------


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

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


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

   for( unsigned int i=0 ; i < num_items; 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 ha recibido valor " << valor_rec <<
         endl << flush ;
      consumir( valor_rec );
   }
}
//
   ----------------------------------------------------------------------


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_emisor_aceptable ;    // identificador 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_emisor_aceptable = etiq_prod ;       // $~~~$ solo
         prod.
   else if ( num_celdas_ocupadas == tam_vector ) // si buffer
      lleno
      tag_emisor_aceptable = etiq_cons ;      // $~~~$ solo
         cons.
   else                                          // si no
      vacío ni lleno
      tag_emisor_aceptable = MPI_ANY_TAG ;     // $~~~$
         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 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_productores-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;
}
```

Ejemplo de la ejecución del programa para 20 elementos a producir:

```
 mpirun -np 10 prodcons2-mu
Productor ha producido valor 11
Productor va a enviar valor 11
Buffer ha recibido valor 11
Buffer va a enviar valor 11
Consumidor ha recibido valor 11
Productor ha producido valor 1
Productor va a enviar valor 1
Buffer ha recibido valor 1
Buffer va a enviar valor 1
Consumidor ha recibido valor 1
Productor ha producido valor 6
Productor va a enviar valor 6
Buffer ha recibido valor 6
Buffer va a enviar valor 6
Consumidor ha recibido valor 6
Productor ha producido valor 16
Productor va a enviar valor 16
Productor ha producido valor 12
Productor va a enviar valor 12
Buffer ha recibido valor 16
Buffer va a enviar valor 16
Buffer ha recibido valor 12
Buffer va a enviar valor 12
Consumidor ha recibido valor 16
```

```
Consumidor ha recibido valor 12
Productor ha producido valor 13
Productor va a enviar valor 13
Buffer ha recibido valor 13
Productor ha producido valor 2
Productor va a enviar valor 2
Buffer ha recibido valor 2
Buffer ha recibido valor 17
Productor ha producido valor 17
Productor va a enviar valor 17
Productor ha producido valor 7
Productor va a enviar valor 7
Buffer ha recibido valor 7
Buffer ha recibido valor 3
Productor ha producido valor 3
Productor va a enviar valor 3
Productor ha producido valor 14
Productor va a enviar valor 14
Buffer ha recibido valor 14
Buffer va a enviar valor 13
Consumidor ha consumido valor 11
Consumidor ha recibido valor 13
Buffer va a enviar valor 2
Consumidor ha consumido valor 12
Consumidor ha recibido valor 2
Consumidor ha consumido valor 6
Consumidor ha recibido valor 17
Buffer va a enviar valor 17
Productor ha producido valor 8
Productor va a enviar valor 8
Buffer ha recibido valor 8
Buffer ha recibido valor 18
Productor ha producido valor 18
Productor va a enviar valor 18
Buffer ha recibido valor 4
Productor ha producido valor 4
Productor va a enviar valor 4
Buffer va a enviar valor 7
Consumidor ha consumido valor 16
Consumidor ha recibido valor 7
Buffer va a enviar valor 3
Consumidor ha consumido valor 1
Consumidor ha recibido valor 3
Productor ha producido valor 15
Productor va a enviar valor 15
```

```
Buffer ha recibido valor 15
Buffer ha recibido valor 9
Productor ha producido valor 9
Productor va a enviar valor 9
Buffer ha recibido valor 10
Productor ha producido valor 10
Productor va a enviar valor 10
Productor ha producido valor 16
Productor va a enviar valor 16
Buffer ha recibido valor 16
Productor ha producido valor 19
Productor va a enviar valor 19
Buffer ha recibido valor 19
Buffer ha recibido valor 5
Productor ha producido valor 5
Productor va a enviar valor 5
Consumidor ha consumido valor 13
Consumidor ha recibido valor 14
Buffer va a enviar valor 14
Productor ha producido valor 20
Productor va a enviar valor 20
Buffer ha recibido valor 20
Productor ha producido valor 17
Productor va a enviar valor 17
Buffer va a enviar valor 8
Buffer ha recibido valor 17
Consumidor ha consumido valor 7
Consumidor ha recibido valor 8
Consumidor ha consumido valor 2
Consumidor ha recibido valor 18
Buffer va a enviar valor 18
Productor ha producido valor 21
Productor va a enviar valor 21
Buffer ha recibido valor 21
Productor ha producido valor 18
Productor va a enviar valor 18
Productor ha producido valor 6
Productor va a enviar valor 6
Productor ha producido valor 11
Productor va a enviar valor 11
Buffer va a enviar valor 4
Consumidor ha consumido valor 17
Buffer ha recibido valor 6
Consumidor ha recibido valor 4
Productor ha producido valor 22
```

```
Productor va a enviar valor 22
Consumidor ha consumido valor 3
Buffer va a enviar valor 15
Consumidor ha recibido valor 15
Buffer ha recibido valor 11
Consumidor ha consumido valor 14
Productor ha producido valor 12
Productor va a enviar valor 12
Productor ha producido valor 7
Productor va a enviar valor 7
Consumidor ha consumido valor 8
Consumidor ha consumido valor 18
Consumidor ha consumido valor 4
Consumidor ha consumido valor 15
```

A. Entorno de ejecución

```
javier@javier-zenbook
OS: Ubuntu 16.04 xenial
Kernel: x86_64 Linux 4.10.0-35-lowlatency
Uptime: 3d 4h 14m
Packages: 3866
Shell: fish
Resolution: 1920x1080
DE: Cinnamon 2.8.6
WM: Muffin
WM Theme:  (Numix)
Adwaita [GTK2/3]
Icon Theme: ubuntustudio
Font: Sans 9
CPU: Intel Core i7-7500U CPU @ 3.5GHz
GPU: Mesa DRI Intel(R) HD Graphics 620 (Kabylake GT2)
RAM: 2850MiB / 7862MiB                          `
```