UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Práctica 1. Sincronización de hebras con semáforos.

Carlos Ureña / Jose M. Mantas / Pedro Villar

2021-22

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Práctica 1. Sincronización de hebras con semáforos.

Índice.

Sección 1.
Objetivos. Espera bloqueda..

Objetivos.

En esta práctica se realizarán dos implementaciones de dos problemas sencillos de sincronización usando librerías abiertas para programación multihebra y semáforos. Los objetivos son:

- Conocer como se pueden generar números aleatorios y dejar una hebra bloqueada durante un intervalo de tiempo finito.
- Conocer el *problema del productor-consumidor* y sus aplicaciones.
  - Diseñar una solución al problema basada en semáforos.
  - Implementar esa solución con la biblioteca para semáforos.
- Conocer un problema sencillo de sincronización de hebras (el *problema de los fumadores*)
  - Diseñar una solución basada en semáforos, teniendo en cuenta los problemas que pueden aparecer.
  - Implementar esa solución con la biblioteca para semáforos.

Generación de números aleatorios

En C++11 se ofrecen disitintas clases para generar números aleatorios.

- Por simplicidad, usaremos la plantilla de función de nombre **`aleatorio`**, ya implementada en **`scd.h`**
- Sirve para generar un número entero aleatorio, cuyo valor estará entre un mínimo y un máximo (ambos incluidos), deben ser dos constantes conocidas al compilar, p.ej:

```
const int desde = 34, hasta = 45 ;  // const es necesario (o constexpr)
....
num1 = aleatorio< 0, 2 >();         // núm. aleatorio entre 0 y 2
num2 = aleatorio< desde, hasta >(); // aleatorio entre 34 y 45
num3 = aleatorio< desde, 65 >();    // aleatorio entre 34 y 65
```

Espera bloqueada de una hebra

En los ejemplos de esta prácticas queremos introducir esperas bloqueadas en las hebras

- El objetivo es simular la realización de trabajo útil por parte de las hebras durante un intervalo de tiempo.
- Las esperas serán de una duración aleatoria, para producir una variedad mayor de posibles interfoliaciones.

Para hacer las esperas se puede usar el método **`sleep_for`** de la clase **`this_thread`**. El argumento es un valor de tipo **`duration`**. En este ejemplo usamos una duración en milisegundos (milésimas de segundo):

```
// calcular una duración aleatoria de entre 20 y 200 milisegundos
chrono::milliseconds duracion_bloqueo_ms( aleatorio<20,200>() );
// esperar durante ese tiempo
this_thread::sleep_for( duracion_bloqueo_ms );
```

Sección 2.
El problema del productor-consumidor.

2.1. Descripción del problema.
2.2. Diseño de la sincronización con semáforos
2.3. Plantillas para la implementación
2.4. Actividades y documentación

Subsección 2.1.
Descripción del problema..

Problema y aplicaciones

El problema del productor consumidor surge cuando se quiere diseñar un programa en el cual un proceso o hebra produce items de datos en memoria que otro proceso o hebra consume.

- Un ejemplo sería una aplicación de reproducción de vídeo:
  - El **productor** se encarga de leer de disco o la red y descodificar cada cuadro de vídeo.
  - El **consumidor** lee los cuadros descodificados y los envía a la memoria de vídeo para que se muestren en pantalla

  hay muchos ejemplos de situaciones parecidas.
- En general, el productor calcula o produce una secuencia de items de datos (uno a uno), y el consumidor lee o consume dichos items (tambien uno a uno).
- El tiempo que se tarda en producir un item de datos puede ser variable y en general distinto al que se tarda en consumirlo (también variable).

Solución de dos hebras con un vector de items

Para diseñar un programa que solucione este problema:

- Suele ser conveniente implementar el productor y el consumidor como dos hebras independientes, ya que esto permite tener ocupadas las CPUs disponibles el máximo de tiempo.
- Se puede usar una variable compartida que contiene un ítem de datos, pero las esperas asociadas a la lectura y la escritura pueden empeorar la eficiencia.
- Esto puede mejorarse usando un vector que pueda contener muchos items de datos producidos y pendientes de leer.
- Para ello, usamos un vector o array de tamaño fijo conocido k.

Esquema de las hebras sin sincronización

La hebra productora y la consumidora ejecutan ambas un bucle. La productora produce e inserta un valor en el buffer intermedio. La consumidora extrae un valor un los consume. El esquema (en pseudo-código) puede ser así:

```
{ variables compartidas y valores iniciales }
var tam_vec   : integer := k  ;                    { tamaño del vector   }
    num_items : integer := .... ;                  { número de items     }
    vec       : array[0..tam_vec-1] of integer;    { vector intermedio   }
```

```
process HebraProductora ;
var a : integer ;
begin
 for i := 0 to num_items-1 do begin
   a := ProducirValor() ;
   { Sentencia E:                    }
   { (insertar valor 'a' en 'vec') }
 end
end
```

```
process HebraConsumidora
var b : integer ;
begin
 for i := 0 to num_items-1 do begin
   { Sentencia L:                    }
   { (extraer valor 'b' de 'vec') }
   ConsumirValor(b) ;
 end
end
```

Condición de sincronización

En esta situación, la implementación debe asegurar que :

- Cada ítem producido es leído (ningún ítem se pierde)
- Ningún ítem se lee más de una vez.

lo cual implica:

- El productor tendrá que esperar antes de poder escribir en el vector cuando haya creado un ítem pero el vector esté completamente ocupado por ítems pendientes de leer
- El consumidor debe esperar cuando vaya a leer un ítem del vector pero dicho vector no contenga ningún ítem pendiente de leer.
- En algunas aplicaciones el orden de lectura o extracción de datos del buffer debe coincidir con el de escritura o inserción, en otras podría ser irrelevante.

Otras propiedades requeridas

Además de lo anterior:

- El programa **no debe impedir** (mediante los semáforos) que el escritor pueda estar produciendo un item al mismo tiempo que el consumidor esté consumiendo otro item (si se impidiese, no tendría sentido usar programación concurrente para esto).
- Lo anterior se refiere exclusivamente a los subprogramas o funciones encargadas de producir o consumir un dato, no se refiere a las operaciones de extraer un dato del buffer o insertar un dato en dicho buffer.
- En el programa, la producción de un dato y su consumo no emplean mucho tiempo de cálculo, ya que es un ejemplo simplificado. Por tanto, se deben introducir retrasos aleatorios variables para poder experimentar distintos patrones de interfoliación de las hebras (ver las plantillas).

Subsección 2.2.
Diseño de la sincronización con semáforos.

Condiciones de sincronización

Durante la ejecución, en cualquier estado,

- El total de valores insertados en el buffer desde el inicio es $\#E$.
- El total de valores extraidos desde el inicio es $\#L$.
- El número de valores insertados en el buffer, y todavía pendientes de ser extraidos es $\#E - \#L$ (lo llamamos *ocupación* del buffer)

Por tanto, surgen estas dos condiciones de sincronización:

- En ningún momento pueden haberse extraído más valores de los que se han insertado, es decir:

$$\#L \leq \#E$$

- En ningún momento la ocupación del buffer puede ser superior a su tamaño fijo conocido (k), es decir:

$$\#E - \#L \leq k$$

Diseño de los semáforos

Por tanto, las dos condiciones anteriores pueden escribirse de esta forma:

$$0 \leq \#E - \#L \qquad \text{and} \qquad 0 \leq k + \#L - \#E$$

Y, al igual que en otros ejemplos, podemos sincronizar los procesos con dos semáforos:

- Un semáforo llamado **`ocupadas`**, cuyo valor será

  $$\#E - \#L$$

  (es el número de entradas ocupadas del buffer, inicialmente 0)
- Un semáforo llamado **`libres`**, cuyo valor será

  $$k + \#L - \#E$$

  (es el número de entradas libres del buffer, inicialmente k).

Esquema de las hebras con sincronización

Ahora podemos incluir las declaraciones de los semáforos y las correspondientes operaciones:

```
{ variables compartidas y valores iniciales }
var tam_vec   : integer := .... ;                { tamaño del vector   }
    num_items : integer := .... ;                { número de items     }
    vec       : array[0..tam_vec-1] of integer; { vector intermedio   }
    libres    : semaphore := tam_vec; { núm. entradas libres (k+#L-#E)}
    ocupadas  : semaphore := 0 ;      { núm. entradas ocup. (#E-#L) }
```

```
process HebraProductora ;
var a : integer ;
begin
 for i := 0 to num_items-1 do begin
   a := ProducirValor() ;
   sem_wait( libres );
   { Sentencia E:                    }
   { (insertar valor 'a' en 'vec') }
   sem_signal( ocupadas );
 end
end
```

```
process HebraConsumidora
var b : integer ;
begin
 for i := 0 to num_items-1 do begin
   sem_wait( ocupadas );
   { Sentencia L:                    }
   { (extraer valor 'b' de 'vec') }
   sem_signal( libres );
   ConsumirValor(b) ;
 end
end
```

Subsección 2.3.
Plantillas para la implementación.

Características

En esta práctica se diseñará e implementará un ejemplo sencillo en C/C++

- Cada ítem de datos será un valor entero de tipo int,
- El orden en el que se leen los items es irrelevante (en principio),
- El productor produce los valores enteros en secuencia, empezando en 1,
- El consumidor escribe cada valor leído en pantalla,
- Se usará un array compartido de valores tipo **`int`**, de tamaño fijo pero arbitrario.

Funciones para producir y consumir:

La hebra productora llama a **producir_dato** para producir el siguiente dato, incluye un retraso aleatorio y usa la variable global **siguiente_dato** (inicializada a **0**):

```
unsigned producir_dato()
{
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ));
   const unsigned dato_producido = siguiente_dato ;
   siguiente_dato++ ; // incrementarlo para la próxima llamada
   cont_prod[dato_producido] ++ ; // incrementar contador de verificación
   cout << "producido: " << dato_producido << endl  ;
   return dato_producido ;
}
```

La hebra consumidora llama a esta otra para consumir un dato:

```
void consumir_dato( int dato )
{
   this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() );
   cout << "Consumidor: dato consumido:                " << dato << endl ;
}
```

Funciones de las hebras productora y consumidora

Las funciones que ejecutan las hebras tienen esta forma:

```
void funcion_hebra_productora(  )
{  for( unsigned i = 0 ; i < num_items ; i++ )
   {  unsigned dato = producir_dato() ;
      // falta aquí: insertar dato en el vector intermedio:
      // ..........
   }
}
void funcion_hebra_consumidora(  )
{  for( unsigned i = 0 ; i < num_items ; i++ )
   {  unsigned dato ;
      // falta aquí: extraer dato desde el vector intermedio
      // .......
      consumir_dato( dato ) ;
   }
}
```

Es necesario definir la constante num_items con algún valor concreto (entre 50 y 100 es adecuado)

Gestión de la ocupación del vector intermedio

El vector intermedio (*buffer*) será un array C++ de tamaño fijo, tiene una capacidad (número de celdas usables) fija prestablecidada en una constante del programa que llamamos, por ejemplo, `tam_vec` (contiene el valor k)

- La constente `tam_vec` deberá ser estrictamente menor que `num_items` (entre 10 y 20 sería adecuado).
- En cualquier instante de la ejecución, el número de celdas ocupadas en el vector (por items de datos producidos pero pendientes de leer) es un número entre 0 (el buffer estaría vacío) y `tam_vec` (el buffer estaría lleno).
- Además del vector, es necesario usar alguna o algunas variables adicionales que reflejen el estado de ocupación de dicho vector.
- Es necesario estudiar si el acceso a dicha variable o variables requiere o no requiere sincronización alguna entre el productor y el consumidor.

Soluciones para la gestión de la ocupación (1/2)

Hay básicamente dos alternativas posibles para gestionar la ocupación, se detallan aquí:

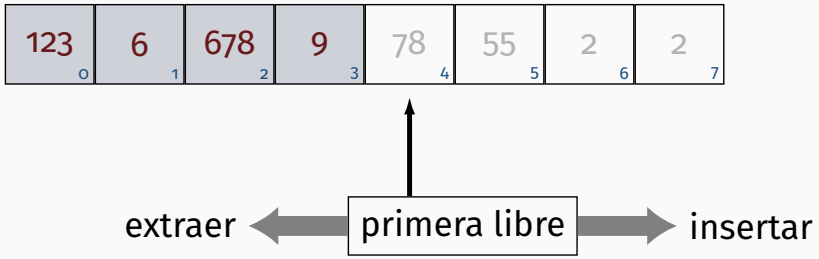
- LIFO (pila acotada), se usa una variable entera (≥ 0):
  - **`primera_libre`** = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir, y se decrementa al leer.
- FIFO (cola circular), se usan dos variables enteras no negativas:
  - **`primera_ocupada`** = índice en el vector de la primera celda ocupada (inicialmente `0`). Esta variable se incrementa al leer (módulo `tam_vec`).
  - **`primera_libre`** = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir (módulo `tam_vec`).

(los índices del vector van desde `0` hasta `tam_vec-1`, ambos incluidos)

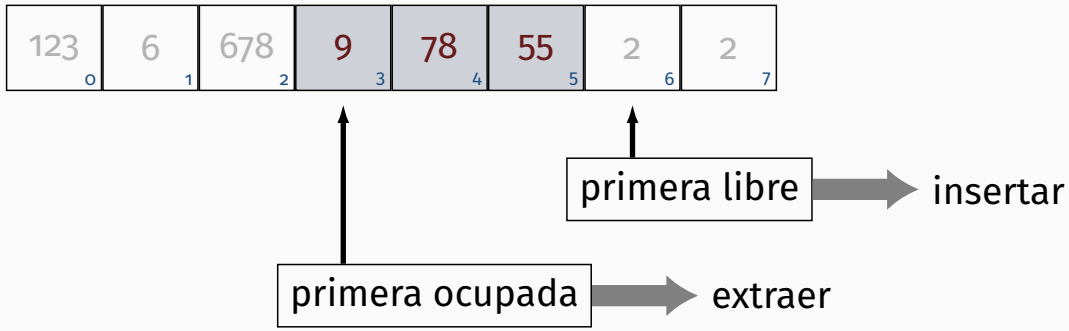
Soluciones para la gestión de la ocupación (2/2)

En un momento dado, suponiendo ($k =$ `tam_vec==8`), el estado del vector puede ser como se ve aquí. Las celdas ocupadas (en gris) contienen valores enteros pendientes de leer:

LIFO (pila acotada: último en entrar es el primero en salir)



FIFO (cola circular: primero en entrar es el primero en salir)



Seguimiento de las operaciones sobre el buffer

Para poder depurar el programa y hacer seguimiento:

- Es conveniente incluir sentencias para imprimir en pantalla el valor insertado o extraido del buffer, justo después de cada vez que se hace (estos mensajes son **adicionales** a los mensjes. que se imprimen al producir o al consumir). Por tanto, se dan dos pasos:
  1. Insertar o extraer el dato del buffer
  2. Escribir un mensaje en pantalla indicando lo que se ha hecho
- Ten en cuenta que el estado de la simulación puede cambiar (y en pantalla pueden aparecer otros mensajes) después del paso 1, pero antes del paso 2.
- Esto puede llevar a cierta confusión, ya que los mensajes no reflejan el estado cuando aparecen, sino un estado antiguo distinto del actual.

Subsección 2.4.
Actividades y documentación.

Lista de actividades

Debes realizar las siguientes actividades en el orden indicado:

1. Diseña una solución que permita conocer qué entradas del vector están ocupadas y qué entradas están libres (usa alguna de las dos opciones dadas).
2. Diseña una solución, mediante semáforos, que permita realizar las esperas necesarias para cumplir los requisitos descritos.
3. Implementa la solución descrita en un programa C/C++ con hebras C++11 y usando la biblioteca de semáforos, completando las plantillas incluidas en este guión. Ten en cuenta que el programa debe escribir la palabra `fin` cuando hayan terminado las dos hebras.
4. Comprueba que tu programa es correcto: verifica que cada número natural producido es consumido exactamente una vez.

Documentación a incluir dentro del portafolios

Se incorporará al portafolios un documento indicando la siguiente información:

1. Describe la variable o variables necesarias, y cómo se determina en qué posición se puede escribir y en qué posición se puede leer.
2. Describe los semáforos necesarios, la utilidad de los mismos, el valor inicial y en qué puntos del programa se debe usar **sem_wait** y **sem_signal** sobre ellos.
3. Incluye el código fuente completo de la solución adoptada.

Sección 3.
El problema de los fumadores..

3.1. Descripción del problema.
3.2. Plantillas de código
3.3. Actividades y documentación.

Subsección 3.1.
Descripción del problema..

Descripción del problema (1)

En este apartado se intenta resolver un problema algo más complejo usando hebras y semáforos. Considerar un estanco en el que hay tres fumadores y un estanquero. Se deben tener en cuenta estos requisitos:

1.1. Cada fumador representa una hebra que realiza una actividad (fumar), invocando a una función **fumar** , en un bucle infinito.

1.2. Cada fumador debe esperar antes de fumar a que se den ciertas condiciones (tener suministros para fumar), que dependen de la actividad del proceso que representa al estanquero.

1.3. El estanquero produce suministros para que los fumadores puedan fumar, también en un bucle infinito.

1.4. Para asegurar concurrencia real, es importante tener en cuenta que la solución diseñada **debe permitir que varios fumadores fumen simultáneamente**.

Descripción del problema (2)

Además de los anterios requisitos, se deben tener en cuenta estos:

2.1. Antes de fumar es necesario liar un cigarro, para ello el fumador necesita tres ingredientes: tabaco, papel y cerillas.

2.2. Uno de los fumadores tiene papel y tabaco, otro tiene papel y cerillas, y otro tabaco y cerillas.

2.3. El estanquero selecciona **aleatoriamente** un ingrediente de los tres que se necesitan para hacer un cigarro, lo pone en el mostrador, desbloquea al fumador que necesita dicho ingrediente y después se bloquea, esperando la retirada del ingrediente.

2.4. El fumador desbloqueado toma el ingrediente del mostrador, desbloquea al estanquero para que pueda seguir sirviendo ingredientes y **después** fuma durante un tiempo aleatorio.

2.5. El estanquero, cuando se desbloquea, vuelve a poner un ingrediente aleatorio en el mostrador, y se repite el ciclo.

Sentencias y funciones

Para poder expresar las condiciones de sincronización fácilmente, haremos estos supuestos:

- Numeramos los fumadores como fumadores 0,1 y 2. Se numeran igualmente los ingredientes. El fumador número i necesita obtener el ingrediente número i para fumar.
- Llamamos P_i a una sentencia que ejecuta el estanquero cuando pone el ingrediente número i en el mostrador. Consiste en la impresión de un mensaje informativo (tipo: "*estanquero produce ingrediente i*").
- LLamamos R_i a una sentencia que ejecuta el fumador número i, por la cual retira el ingrediente i del mostrador, previamente a fumar. Consiste en imprimir un mensaje informativo (tipo "*el fumador i retira su ingrediente*")

Esquema de las hebras sin sincronización

El esquema de las hebras es como sigue:

```
process HebraEstanquero ;
var i : integer ;
begin
  while true do begin
    { simular la producción: }
    i := Producir() ;
    { Sentencia P_i : }
    print("puesto ingr.: ",i);
 end
end
```

```
process HebraFumador[ i : 0..2 ]
 var b : integer ;
begin
  while true do begin
    { Sentencia R_i : }
    write("retirado ingr.:",i);
    { simular el fumar: }
    Fumar( i );
  end
end
```

- El estanquero produce un número aleatorio mediante una llamada a la función **Producir**, que conlleva un cierto retraso aleatorio y devuelve un entero (0,1 o 2).
- Los fumadores fuman llamando a la función **Fumar**, que conlleva un retraso aleatorio.

Condicion de sincronización y semáforos (1)

Hay un total de 6 instrucciones que se ejecutan repetidamente (P_i y R_i, tres en cada caso). Pero no podemos permitir cualquier interfoliación de las mismas. En concreto:

- Para cada i, el número de veces que ha el fumador número i ha retirado el ingrediente número i no puede ser mayor que el número de veces que se ha producido el ingrediente i, es decir $\#R_i \leq \#P_i$, o lo que es lo mismo:

$$0 \leq \#P_i - \#R_i$$

- Esto se puede resolver con tres semáforos s_i (un array de semáforos, con $i = 0, 1, 2$), cada uno de ellos vale $\#P_i - \#R_i$.
- En el semáforo s_i se debe de hacer:
  - `sem_wait` antes de R_i (aparece con signo negativo)
  - `sem_signal` después de P_i (aparece con signo positivo)

Condición de sincronización y semáforos (2)

La condición anterior no contempla todas las restricciones del problema, hay que añadir esta:

- El número p de ingredientes producidos (en el mostrador) y pendientes de ser usados por algún fumador es

$$p = \sum_{i=0}^{2} \#P_i - \sum_{i=0}^{2} \#R_i$$

- Solo cabe un ingrediente como mucho en el mostrador, luego p solo puede ser 0 o 1. Es decir, se debe cumplir (a) $0 \leq p$ y (b) $p \leq 1$.
- La condición (a) esta garantizada por la condición de la transparencia anterior, así que solo hay que asegurar (b), que se puede escribir como:

$$0 \leq 1 + \sum_{i=0}^{2} \#R_i - \sum_{i=0}^{2} \#P_i$$

Condición de sincronización y semáforos (3)

Por tanto, podemos usar un semáforo (lo llamamos **`mostr_vacio`**), cuyo valor es

$$1 + \sum_{i=0}^{2} \#R_i - \sum_{i=0}^{2} \#P_i$$

Es decir, vale 1 si el mostrador está libre, y 0 si hay un ingrediente en él. Sobre ese semáforo, debemos de hacer:

- **`sem_wait`** antes de cualquier sentencia P_i (ya que aparecen con signo negativo)
- **`sem_signal`** después de cualquier sentencia R_i (ya que aparecen con signo positivo)

Las sentencias P_i y R_i no suponen asignación ninguna (no hacen nada).

Esquema de las hebras

Por tanto, el esquema de las hebras, incluyendo la sincronización, será este:

```
{ variables compartidas y valores iniciales }
var mostr_vacio : semaphore := 1 ; { 1 si mostrador vacío, 0 si ocupado }
    ingr_disp   : array[0..2] of semaphore := { 0,0,0 } ;
        { 1 si el ingrediente i esta disponible en el mostrador, 0 si no}
```

```
process HebraEstanquero ;
  var i : integer ;
begin
  while true do begin
    i := Producir();
    sem_wait( mostr_vacio );
    print("puesto ingr.: ",i); {P_i}
    sem_signal( ingr_disp[i] );
 end
end
```

```
process HebraFumador[ i : 0..2 ]
begin
  while true do begin
    sem_wait( ingr_disp[i] ) ;
    print("retirado ingr.:",i); {R_i}
    sem_signal( mostr_vacio );
    Fumar( i );
  end
end
```

Subsección 3.2.
Plantillas de código.

Simulación de la acción de fumar

Para simular la acción de fumar se puede usar la función **fumar**, que tiene como parámetro el número de fumador, y produce un retraso aleatorio.

```
// función que simula la acción de fumar, como un retardo aleatorio de la hebra.
// recibe como parámetro el numero de fumador
void fumar( int num_fum )
{
  cout << "Fumador número " << num_fum << ": comienza a fumar." << endl;
  this_thread::sleep_for( chrono::milliseconds( aleatorio<50,200>() ));
  cout << "Fumador número " << num_fum << ": termina de fumar." << endl;
}

// funciones que ejecutan las hebras
void funcion_hebra_estanquero(  ) { .... }
void funcion_hebra_fumador( int num_fum ) { .... }

int main()
{
  // poner en marcha las hebras y esperar que terminen ......
}
```

Subsección 3.3.
Actividades y documentación..

Diseño de la solución

Diseña e implementa una solución al problema en C/C++ usando cuatro hebras y los semáforos necesarios. La solución debe cumplir los requisitos incluidos en la descripción, y además debe:

- Evitar interbloqueos entre las distintas hebras.
- Producir mensajes en la salida estándar que permitan hacer un seguimiento de la actividad de las hebras:
  - El estanquero debe indicar cuándo produce un suministro y qué suministro produce.
  - Cada fumador debe indicar cuándo espera, qué producto espera, y cuándo comienza y finaliza de fumar.

Documentación a incluir dentro del portafolios

Se incorporará al portafolios un documento incluyendo los siguientes puntos:

1. Nombres de los semáforos empleados para sincronización y, para cada uno de ellos:
   - Utilidad.
   - Valor inicial.
   - Hebras que hacen **`wait`** y **`signal`** sobre dicho semáforo.
2. Código fuente completo de la solución adoptada.

Fin de la presentación.

UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Práctica 2. Casos prácticos de monitores en C++11.

Carlos Ureña / Jose M. Mantas / Pedro Villar

2021-22

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Práctica 2. Casos prácticos de monitores en C++11.

Índice.

Objetivos

Los objetivos de esta práctica son ilustrar el proceso de diseño e implementación de los monitores SU no triviales, usando para ello dos casos prácticos:

- El problema de los fumadores, que ya hemos visto resuelto con semáforos, pero ahora con monitores
- El problema de los lectores-escritores

Al igual que en los problemas anteriores, veremos el proceso de diseño (creación del pseudo-código), seguido de la implementación (traducción a monitores SU u monitores Hoare)

Sección 1.
El problema de los fumadores.

El problema de los fumadores

En este ejercicio consideraremos de nuevo el mismo problema de los fumadores y el estanquero cuya solución con semáforos ya vimos en la práctica 1. Queremos diseñar e implementar **un monitor SU** (**monitor Hoare**) que cumpla los requerimientos:

- Se mantienen las tres hebras de fumadores y la hebra de estanquero.
- Se mantienen exactamente igual todas las condiciones de sincronización entre esas hebras.
- El diseño de la solución incluye un monitor (de nombre `Estanco`) y las variables condición necesarias.
- Hay que tener en cuenta que ahora no disponemos de los valores de los semáforos para conseguir la sincronización.

A continuación haremos un diseño del monitor, y se deja como actividad la implementación de dicho diseño.

Hebras de fumadores

Los **fumadores**, en cada iteración de su bucle infinito:

- LLaman al procedimiento del monitor `obtenerIngrediente(i)`, donde `i` es el número de fumador (o el número del ingrediente que esperan). En este procedimiento el fumador espera bloqueado a que su ingrediente esté disponible, y luego lo retira del mostrador.
- Fuman, esto es una llamada a la función **Fumar**, que es una espera aleatoria.

```
process Fumador[ i : 0..2 ] ;
begin
  while true do begin
    Estanco.obtenerIngrediente( i );
    Fumar( i ) ;
  end
end
```

Hebra estanquero

El **estanquero**, en cada iteración de su bucle infinito:

- Produce un ingrediente aleatorio (llama a una función **`ProducirIngrediente()`**, que hace una espera de duración aleatoria y devuelve un número de ingrediente aleatorio).
- Llama al procedimiento del monitor **`ponerIngrediente(i)`**, (se pone el ingrediente `i` en el mostrador) y después a **`esperarRecogidaIngrediente()`** (espera bloqueado hasta que el mostrador está libre).

```
process Estanquero ;
  var ingre : integer ;
begin
  while true do begin
    ingre := ProducirIngrediente();
    Estanco.ponerIngrediente( ingre );
    Estanco.esperarRecogidaIngrediente();
  end
end
```

Variables permanentes y condiciones

Las variables permanentes necesarias se deducen de las esperas que deben hacer las hebras:

- Cada fumador debe esperar a que el mostrador tenga un ingrediente y que ese ingrediente coincida con su número de ingrediente o fumador.
- El estanquero debe esperar a que el mostrador esté vacío (no tenga ningún ingrediente)

Como actividad, debes de escribir (en tu portafolio):

- **Variable o variables permanentes**: para cada una describe el tipo, nombre, valores posibles y significado de la variable.
- **Cola o colas condición**: para cada una, escribe el nombre y la condición de espera asociada (una expresión lógica de las variables permenentes).
- **Pseudo-código** de los tres procedimientos del monitor.

Actividad: implementación del programa

Escribe y prueba un programa que haga la simulación del problema de los fumadores, y que incluya el monitor SU descrito en pseudo-código. En tu portafolio:

- Incluye el código fuente completo de la solución adoptada.
- Incluye un listado de la salida del programa durante una ejecución.

Sección 2.
El problema de los Lectores/Escritores.

El problema de los Lectores/Escritores.

Dos tipos de procesos acceden concurrentemente a datos compartidos:

- **Escritores**: procesos que modifican la estructura de datos (escriben en ella). El código de escritura no puede ejecutarse concurrentemente con ninguna otra escritura ni lectura (ya que modifica el estado de la estructura de datos)
- **Lectores**: procesos que leen la estructura de datos, pero no modifican su estado en absoluto. El código de lectura puede (y debe) ejecutarse concurrentemente por varios lectores de forma arbitraria, pero no puede hacerse a la vez que la escritura.

La solución de este problema usando semáforos es compleja, veremos que con monitores es sencillo.

Uso del monitor

Los procesos lectores y escritores usan el monitor de esta forma:

```
process Lector[ i:1..n ] ;
begin
   while true do begin
     Lec_Esc.ini_lectura() ;
     { código de lectura }
     Lec_Esc.fin_lectura() ;
     { resto de código }
   end
end
```

```
process Escritor[ i:1..m ] ;
begin
   while true do begin
      Lec_Esc.ini_escritura() ;
      { código de escritura }
      Lec_Esc.fin_escritura() ;
      { resto de código }
   end
end
```

- En esta implementación se ha dado prioridad a los lectores (en el momento que un escritor termina, si hay escritores y lectores esperando, pasan los lectores).
- Hay otras opciones: prioridad a escritores, prioridad al que más tiempo lleva esperando.
- El código de lectura, el de escritura y el resto del código son retrasos aleatorios.

Diseño de la solución: variables permanentes

Para poder introducir las esperas necesarias, tenemos que tener variables permanentes del monitor que nos describan el estado del recurso compartido. En este ejemplo, necesitamos dos variables:

- **`escrib`**
  variable lógica, vale **`true`** si un escritor está escribiendo, **`false`** si no hay escritores escribiendo (inicialmente **`false`**)
- **`n_lec`**
  variable entera (no negativa), es el número de lectores que están escribiendo en un momento dado (inicialmente `0`).

Los valores de estas variables reflejan correctamente el estado del monitor solo cuando no se está ejecutando código del mismo por alguna hebra (en ese caso pueden estar siendo actualizadas).

Diseño de la solución: variables condición

Las esperas se hacen en dos variables condición

- Variable condición **`lectura`**: se usa por los lectores (en **`ini_lectura`**) para esperar cuando ya hay un escritor escribiendo (es decir, cuando **`escrib==true`**).
- Variable condición **`escritura`**: se usa por los escritores (en **`ini_escritura`**) para esperar cuando ya hay otro escritor escribiendo (**`escrib==true`**) o bien hay lectores leyendo (**`n_lec>0`**).

Estas esperas aseguran que siempre se cumple:

```
(not escrib) or (n_lec == 0)
```

Cuando se termina de leer o de escribir, habrá que hacer los corresponientes **`signal`** en estas variables condición.

Vars. permanentes y procedimientos para lectores

Por todo lo dicho, el monitor se puede diseñar así:

```
monitor Lec_Esc ;

var n_lec     : integer;   { numero de lectores leyendo }
    escrib    : boolean;   { true si hay algun escritor escribiendo }
    lectura   : condition; { no hay escrit. escribiendo, lectura posible }
    escritura : condition; { no hay lect. ni escrit., escritura posible }

export ini_lectura, fin_lectura,      { invocados por lectores }
       ini_escritura, fin_escritura ; { invocados por escritores }
```

```
procedure ini_lectura()
begin
 if escrib then { si hay escritor: }
   lectura.wait(); { esperar }
 { registrar un lector más }
 n_lec := n_lec + 1 ;
 { desbloqueo en cadena de }
 { posibles lectores bloqueados }
 lectura.signal()
end
```

```
procedure fin_lectura()
begin
  { registrar un lector menos }
  n_lec:= n_lec - 1 ;
  { si es el ultimo lector:   }
  {   desbloquear un escritor }
  if n_lec == 0 then
     escritura.signal()
end
```

Procedimientos para escritores

Los procedimientos para escritores son estos dos:

```
procedure ini_escritura()
begin
 { si hay otros, esperar }
 if n_lec > 0 or escrib then
   escritura.wait()
 { registrar que hay un escritor }
 escrib := true;
end;
```

```
procedure fin_escritura()
begin
 { registrar que ya no hay escritor}
 escrib := false;
 { si hay lectores, despertar uno}
 { si no hay, despertar un escritor}
 if not lectura.empty() then
    lectura.signal();
 else
    escritura.signal() ;
end;
```

```
begin { inicializacion }
   n_lec  := 0 ;
   escrib := false ;
end
```

Fin de la presentación.

UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Práctica 3. Implementación de algoritmos distribuidos con MPI.

Carlos Ureña / Jose M. Mantas / Pedro Villar

2021-22

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Práctica 3. Implementación de algoritmos distribuidos con MPI.
Índice.

Objetivos

Los objetivos de esta práctica son:

- Iniciar a los alumnos en la programación de algoritmos distribuidos.
- Conocer varios problemas sencillos de sincronización y su solución distribuida mediante el uso de la interfaz de paso de mensajes MPI:
  - Diseñar una solución distribuida al problema del **productor - consumidor** con buffer acotado, para varios productores y varios consumidores. El planteamiento del problema es similar al ya visto para múltiples hebras en memoria compartida.
  - Diseñar diversas soluciones al problema de la **cena de los filósofos**.

Sección 1.
Productor-Consumidor con buffer acotado.

Subsección 1.1.
Aproximación inicial.

Aproximación inicial en MPI

En la solución distribuida habrá (inicialmente) **tres procesos**:

- **Productor**: produce una secuencia de datos (números enteros, comenzando en 0), y los envía al proceso buffer.
- **Buffer**: Recibe (de forma alterna) enteros del proceso productor y peticiones del consumidor. Responde al consumidor enviándole los enteros recibidos, en el mismo orden.
- **Consumidor**: realiza peticiones al proceso buffer, como respuesta recibe los enteros y los consume.

El esquema de comunicación entre estos procesos se muestra a continuación:

Aproximación inicial. Estructura del programa.

En `prodcons.cpp` podemos ver una solución inicial al problema. La estructura del programa es como sigue:

```
#include ......   // includes varios
#include <mpi.h>  // includes de MPI
using ..... ;     // using varios

// contantes: asignación de identificadores a roles
const int id_productor           = 0, // identificador del proceso productor
          id_buffer              = 1, // identificador del proceso buffer
          id_consumidor          = 2, // identificador del proceso consumidor
          num_procesos_esperado = 3, // número total de procesos esperado
          num_iteraciones        = 20;// núm. de datos producidos/consum.
// funciones auxiliares
int  producir()           { ... } // produce un valor (usada por productor)
void consumir( int valor ){ ... } // consume un valor (usada por consumidor)
// funciones ejecutadas por los procesos en cada rol:
void funcion_productor()  { ... } // función ejecutada por proceso productor
void funcion_consumidor() { ... } // función ejecutada por proceso consumidor
void funcion_buffer()     { ... } // función ejecutada por proceso buffer
// función main (punto de entrada común a todos los procesos)
int main( int argc, char *argv[] ) { ... }
```

Aproximación inicial. Función `main`.

`main` se encarga de que cada proceso ejecute su función:

```
int main( int argc, char *argv[] )
{
  int id_propio, num_procesos_actual; // ident. propio, núm. 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 )
  {
    if ( id_propio == id_productor )  // si mi ident. es el del productor
      funcion_productor();            //     ejecutar función del productor
    else if ( id_propio == id_buffer )// si mi ident. es el del buffer
      funcion_buffer();               //     ejecutar función buffer
    else                              // en otro caso, mi ident es consumidor
      funcion_consumidor();           //     ejecutar función consumidor
  }
  else if ( id_propio == 0 )  // si hay error, el proceso 0 informa
    cerr << "error: número de procesos distinto del esperado." << endl ;
  MPI_Finalize( );
  return 0;
}
```

Aproximación inicial. Productor y consumidor

Los procesos productor y consumidor usan envío **síncrono seguro**

```
void funcion_productor()
{
 for ( unsigned i = 0 ; i < num_items ; i++ )
 {
   int valor_prod = producir(); // producir (espera bloqueado tiempo aleat.)
   cout << "Productor va a enviar valor " << valor_prod << endl;
   MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD );
 }
}
void funcion_consumidor()
{
 int peticion, valor_rec = 1 ; MPI_Status estado ;

 for( unsigned i = 0 ; i < num_items; i++ )
 {
   MPI_Ssend( &peticion,  1,MPI_INT, id_buffer, 0,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;
   consumir( valor_rec ); // consumir (espera bloqueado tiempo aleat.)
 }
}
```

Aproximación inicial. Proceso buffer

El proceso buffer recibe un dato, luego recibe una petición, y finalmente responde a la petición enviando el dato recibido:

```
void funcion_buffer()
{
 int        valor, peticion ;
 MPI_Status estado ;

 for ( unsigned int i = 0 ; i < num_items ; i++ )
 {
   // recibir valor del productor
   MPI_Recv( &valor,    1,MPI_INT, id_productor, 0,MPI_COMM_WORLD,&estado);
   cout << "Buffer ha recibido valor " << valor << endl ;

   // recibir petición de consumidor, enviarle el dato
   MPI_Recv( &peticion, 1,MPI_INT, id_consumidor,0,MPI_COMM_WORLD,&estado);
   cout << "Buffer va a enviar " << valor << endl;
   MPI_Ssend( &valor,   1,MPI_INT, id_consumidor,0, MPI_COMM_WORLD);
 }
}
```

Valoración de la solución inicial

Sin embargo, esta solución **fuerza una excesiva sincronización entre productor y consumidor**. A largo plazo, el tiempo promedio empleado en **`producir`** será similar al empleado en **`consumir`**, sin embargo:

- En cada llamada hay diferencias arbitrarias entre ambos tiempos.
- Frecuentemente la hebra productora o la hebra consumidora quedará esperando un tiempo hasta que el buffer puede procesar su envío o solicitud.
- Si las hebras consumidora y productora se ejecutan en dos procesadores distintos (en exclusiva para ellas), esos procesadores quedarán sin usar (desocupados) una fracción del tiempo total y el programa puede tarda más en acabar.

Necesitamos algún mecanismo de reducción de las esperas.

Subsección 1.2.
Solución con selección no determinista.

Solución con selección no determinista

Para lograr nuestro objetivo, permitimos que el proceso buffer acomode diferencias temporales en la duración de producir y consumir:

- El proceso buffer puede guardar un vector de valores pendientes de consumir, en lugar de un único valor.
- De esta forma: el productor puede producir varios valores seguidos (sin esperar al consumidor), y el consumidor puede consumir varios seguidos (sin esperar al productor)
- Las esperas se reducen, las CPUs están menos tiempo desocupadas.
- El tiempo total hasta acabar el programa se reduce.

Para lograr esto, necesitamos que el proceso buffer pueda recibir una petición cuando hay valores pendientes de enviar, y a la vez pueda recibir un valor cuando hay celdas donde se pueda guardar.

Espera selectiva en MPI

El comportamiento del buffer que queremos implementar se llama **espera selectiva**

- En cada iteración, el buffer podrá aceptar un mensaje exclusivamente del productor (si el vector está vacío), exclusivamente del consumidor (si el vector está lleno), o de ambos (ni vacío ni lleno).
- MPI permite implementar este comportamiento usando para ello la posibilidad de especificar, en cada operación de recepción, un emisor concreto o cualquier emisor (igualmente con las etiquetas).
- Por tanto, el buffer aceptará, en función del estado del vector, un mensaje solo del productor, solo del consumidor, o de cualquier emisor (es decir, de ambos)

Estructura del proceso buffer

El proceso buffer tiene esta estructura (archivo `prodcons2.cpp`)

```
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
              id_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
      ....
      // 2. recibir un mensaje del emisor o emisores aceptables
      .....
      // 3. procesar el mensaje recibido
      .....
   }
}
```

Recepción de un mensaje

En el cuerpo del bucle, en primer lugar se calcula (en **`id_emisor_aceptable`**) de que proceso o procesos podemos aceptar un mensaje. Después, lo recibimos:

```
// 1. determinar si puede enviar solo prod., solo cons, o de ambos

if ( num_celdas_ocupadas == 0 )                // si buffer vacío
   id_emisor_aceptable = id_productor ;        //    solo prod.
else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
   id_emisor_aceptable = id_consumidor ;       //    solo cons.
else                                           // si no vacío ni lleno
   id_emisor_aceptable = MPI_ANY_SOURCE ;      //    cualquiera

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

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

Procesamiento del mensaje

Una vez recibido el mensaje, el tercer paso es actualizar el buffer en función de que proceso haya sido el que lo ha enviado:

```
// 3. procesar el mensaje recibido
switch( estado.MPI_SOURCE )   // leer emisor del mensaje en metadatos
{
  case id_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 id_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, id_consumidor, 0, MPI_COMM_WORLD);

    break;
}
```

Ejercicio propuesto: múltiples productores y consumidores

Extenderemos el programa anterior (para 1 productor y 1 consumidor) a múltiples productores y consumidores:

- Habrá $n_p = 4$ procesos productores y $n_c = 5$ procesos consumidores.
- Sigue habiendo un único proceso buffer.
- El número m total de items a producir o consumir (constante `num_items`) debe ser múltiplo de n_p y múltiplo de n_c.
- Los procesos con identificador entre 0 y $n_p - 1$ son productores.
- El proceso con identificador n_p es el buffer.
- Los procesos con identificdor entre $n_p + 1$ y $n_p + n_c$ son consumidores.
- Debes declarar dos constantes enteras con el núm. de prods. (n_p) y el núm. de consum. (n_c), asegurate que el programa es correcto aunque se usen otros valores distintos de 4 y 5 para n_p y n_c.

Números de orden de los procesos y producción de valores

Puesto que ahora tenemos múltiples productores y consumidores:

- La función de los productores y la de los consumidores reciben como parámetro el número de orden del productor o del consumidor, respectivamente (esos números son los números de orden en cada rol, comenzando en 0, no son los identificadores de proceso).
- Los números de orden deben calcularse en **`main`**.
- La función de producir dato recibe como parámetro el número de orden del productor que la invoca. Esto permite que los productores usen cada uno su contador (variable **`contador`** de **`producir_dato`**) para producir un rango distinto de valores: el productor con número de orden i producirá los valores entre ik y $ik+k-1$ (ambos incluidos), donde k es el número de valores producidos por cada productor (es decir $k = m/n_p$).

Diseño de la solución con etiquetas

Para solucionar el problema con múltiples prods./cons.:

- Según el estado del buffer, debemos de aceptar un mensaje de cualquier productor, de cualquier consumidor, o de cualquier proceso.
- No es posible usar exactamente la misma estrategia que antes: con MPI no es posible restringir el emisor aceptable a cualquiera de un subconjunto de procesos dentro de un comunicador (o aceptamos de un proceso concreto o aceptamos de todos los del comunicador)
- El problema se puede solucionar usando múltiples comunicadores, pero no hemos visto como definirlos.
- También se puede solucionar **usando dos etiquetas distintas para diferenciar los mensajes de los productores y los consumidores**

Actividad y documentación para el portafolios

Partiendo de `prodcons2.cpp`, crea un nuevo archivo (llamado `prodcons2-mu.cpp`) con tu solución al problema descrito, ahora para múltiples productores y consumidores.

- Diseña una solución basada en el uso de etiquetas, que por lo demás es similar a la ya vista
- Define constantes enteras para las etiquetas: el programa será mucho más legible. Estas constantes deben tener nombres que comienzen con **`etiq_`**

Documentación para el portafolio:

1. Describe qué cambios has realizado sobre el programa de partida y el propósito de dichos cambios.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado parcial de la salida del programa.

Sección 2.
Cena de los Filósofos.

2.1. Aproximación inicial.

2.2. Uso del proceso camarero con espera selectiva

Subsección 2.1.
Aproximación inicial..

Cena de los filósofos en MPI.

Consideramos un programa MPI para el problema de **la cena de los filósofos**. En este problema intervienen 5 **filósofos** y 5 **tenedores**:

- **Los filósofos son 5 procesos** (numerados del 0 al 4) que ejecutan un bucle infinito, en cada iteración comen primero y piensan después (ambas son actividades de duración arbitraria).
- Los filósofos, para comer, se disponen en una mesa circular donde hay un tenedor entre cada dos filósofos. Cuando un filósofo está comiendo, **usa en exclusión mutua sus dos tenedores adyacentes**.

En programación distribuida cada recurso compartido (de uso exclusivo por un único proceso en un instante) debe de implementarse con un proceso gestor adicional (específico para ese recurso), proceso que alterna entre dos estados (libre o en uso).

Procesos tenedor. Sincronización

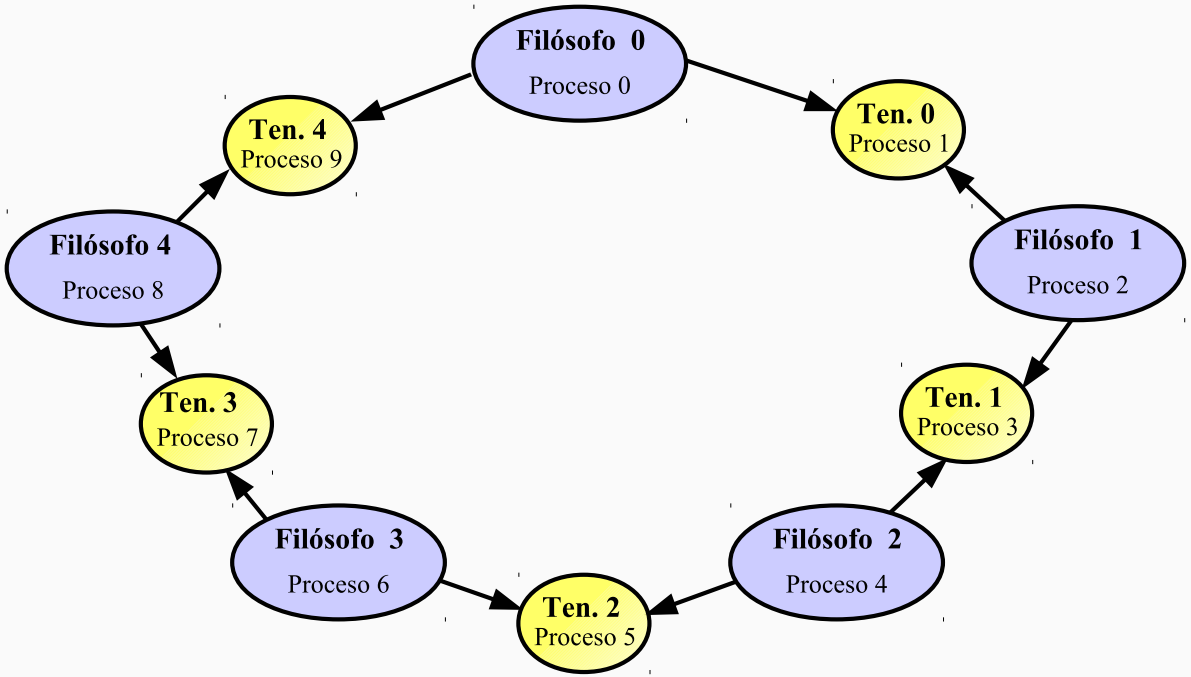
Por tanto, para implementar el problema, **debemos de ejecutar 5 procesos de tipo tenedor**, numerados del 0 al 4.

- Cuando un proceso filósofo va a usar un tenedor, debe de enviar (al proceso tenedor correspondiente) un mensaje síncrono antes de usarlo y otro mensaje después de haberlo usado.
- Cada proceso tenedor ejecuta un bucle infinito, al inicio de cada iteración está libre, y da estos dos pasos:
  1. Espera hasta recibir un mensaje de cualquier filósofo, al recibirlo el tenedor pasa a estar ocupado por ese filósofo.
  2. Espera hasta recibir un mensaje del filósofo que lo adquirió en el paso anterior. Al recibirlo, pasa a estar libre.
- Puesto que el envío (por el filósofo) del mensaje previo al uso es síncrono, supone para dicho filósofo una espera bloqueada hasta adquirir el tenedor en exclusión mutua.

Identificadores de los procesos

Para facilitar la comunicación entre filósofos y tenedores:

- Los procesos filósofos tienen identificadores MPI pares, es decir, el filósofo número i tendrá identificador $2i$.
- Los procesos tenedor tienen identificadores MPI impares, es decir, el tenedor número i tendrá identificador $2i + 1$.



Cena de los filósofos. Programa principal

La función **main** (en `filosofos-plantilla.cpp`) es esta:

```
const int num_filosofos = 5 ,                // número de filósofos
          num_filo_ten  = 2*num_filosofos,   // núm. de filo. + ten.
          num_procesos  = num_filo_ten;      // núm. total de procs.
.....
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 % 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 )
    cerr << "Error: se esperaban 10 procesos. Programa abortado." <<endl;
  MPI_Finalize( );
  return 0;
}
```

Procesos filósofos

En cada iteración del bucle un filósofo realiza repetidamente estas acciones:

1. Tomar los tenedores (primero el tenedor izquierdo y después el derecho).
2. Comer (bloqueo de duración aleatoria).
3. Soltar tenedores (el orden da igual).
4. Pensar (bloqueo de duración aleatoria).

- Las acciones pensar y comer pueden implementarse mediante un mensaje por pantalla seguido de un retardo durante un tiempo aleatorio.
- Las acciones de tomar tenedores y soltar tenedores deben implementarse enviando mensajes **síncronos seguros** de petición y de liberación a los procesos tenedor situados a ambos lados de cada filósofo.

Esquema de los procesos filósofos

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

  while ( true )
  {
    cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
    // ... solicitar tenedor izquierdo (completar)
    cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
    // ... solicitar tenedor derecho (completar)

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

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

Procesos tenedor

Cada proceso tenedor ejecutará en un bucle estas dos acciones:

1. Esperar hasta recibir un mensaje de cualquier filósofo (lo llamamos *mensaje de petición*)
2. Esperar hasta recibir un mensaje del mismo filósofo emisor del anterior (lo llamamos *mensaje de liberación*)

```
void funcion_tenedores( int id ) // id es el identificador del proceso tenedor
{
  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)
    // ..... guardar en id_filosofo el id. del emisor (completar)
    cout <<"Ten. " <<id <<" cogido por filo. " <<id_filosofo <<endl;

    // ..... recibir liberación de filósofo id_filosofo (completar)
    cout <<"Ten. "<< id<< " liberado por filo. " <<id_filosofo <<endl ;
  }
}
```

Actividades : soluciones con interbloqueo y sin interbloqueo

Se propone realizar las siguientes actividades

1. Implementar una solución distribuida al problema de los filósofos de acuerdo con el esquema descrito en las plantillas. Usar la operación síncrona de envío **`MPI_Ssend`**. Copia el archivo de la plantilla (`filosofos-plantilla.cpp`) en el archivo `filosofos-interb.cpp` y completa este último archivo.
2. El esquema propuesto (cada filósofo coge primero el tenedor de su izquierda y después el de la derecha) puede conducir a interbloqueo:
   - Identifica la secuencia de peticiones de filósofos que conduce a interbloqueo.
   - Diseña una modificación que solucione dicho problema.
   - Copia `filosofos-interb.cpp` en `filosofos.cpp` e implementa tu solución en este último archivo.

Documentación para el portafolios

Incluye en tu portafolios cada uno de los siguientes puntos:

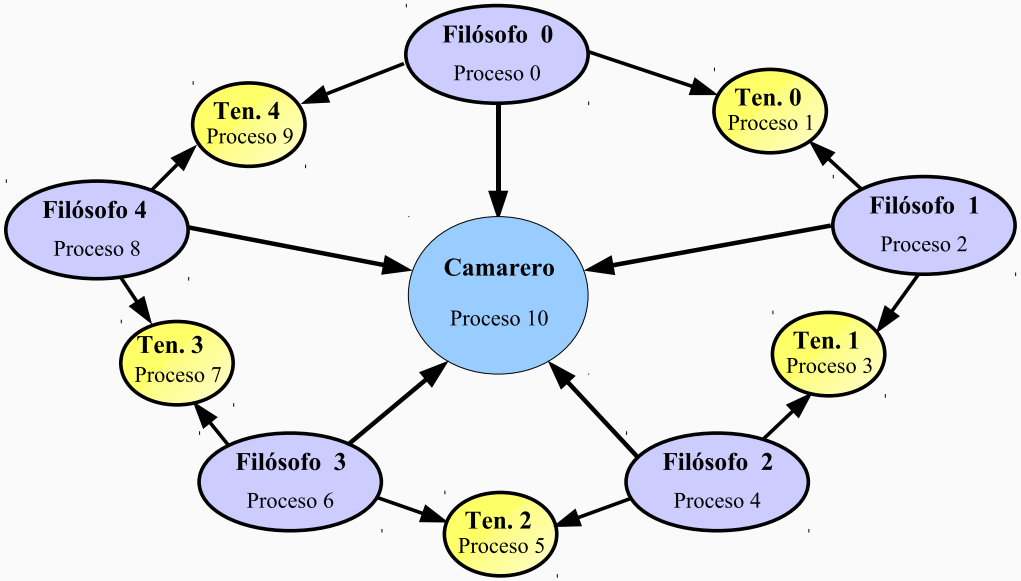
1. Describe los aspectos más destacados de tu solución al problema de los filósofos, la situación que conduce al interbloqueo y tu solución al problema del interbloqueo.
2. Incluye el código fuente completo de la solución adoptada para evitar la posibilidad de interbloqueo.
3. Incluye un listado parcial de la salida de este programa.

Subsección 2.2.
Uso del proceso camarero con espera selectiva.

Cena de los filósofos con camarero

Existe otra opción para solucionar el problema del interbloqueo:

- Se introducen dos pasos nuevos en los filósofos:
  - *sentarse en la mesa* (antes de coger los tenedores)
  - *levantarse de la mesa* (después de soltar los tenedores)
- Un proceso adicional llamado **camarero** (identificador 10) impedirá que haya 5 filósofos sentados a la vez.



Procesos filósofos y camarero

Ahora, cada filósofo ejecutará repetidamente esta secuencia:

1. Sentarse
2. Tomar tenedores
3. Comer
4. Soltar tenedores
5. Levantarse
6. Pensar

Cada filósofo pedirá permiso para sentarse o levantarse haciendo un envío síncrono al camarero. Debemos de implementar de nuevo una **espera selectiva** en el camarero, el cual

- llevará una cuenta (s) del número de filósofos sentados.
- solo cuando $s < 4$ aceptará las peticiones de sentarse.
- siempre aceptará las peticiones para levantarse.

De nuevo, debes de usar etiquetas para esta implementación. Recuerda definir constantes enteras para etiquetas, cuyos nombres deben comenzar por `etiq_`.

Actividades: solución con camarero

Realiza las siguientes actividades:

1. Copia tu solución con interbloqueo (`filosofos-interb.cpp`) sobre un nuevo archivo llamado `filosofos-cam.cpp`
2. Implementa, en este último archivo, el método descrito, basado en un proceso camarero con espera selectiva.

Documentación para el portafolios

Responde en tu portafolios, de forma razonada, a cada uno de los siguientes puntos:

1. Describe tu solución al problema de los filósofos con camarero central.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado parcial de la salida del programa.

Fin de la presentación.