ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Ingeniería Informática + ADE

Universidad de Granada (UGR)

Autor: Ismael Sallami Moreno

Asignatura: Sistemas Concurrentes y Distribuidos

Índice

1 Prácticas

1.1. Práctica 1

UNIVERSIDAD DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2024-25 (archivo generado el 2 de octubre de 2024)

Grado en Ingeniería Informática,
Grado en Informática y Matemáticas,
Grado en Informática y Administración de Empresas.
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 una hebra produce items de datos en memoria mientras que otra hebra los consume.

- Un ejemplo sería una aplicación de reproducción de vídeo:
  - La hebra **productora** se encarga de leer de disco o la red y descodificar cada cuadro de vídeo.
  - La hebra **consumidora** 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, la productora calcula o produce una secuencia de items de datos (uno a uno), y la consumidora 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 la productora y la consumidora 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 cada una un bucle.

- La productora, en cada iteración, produce un valor y después lo inserta en el vector.
- La consumidora, en cada iteración, extrae un valor del vector y después lo consume.

```
{ 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:

- La productora 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.
- La consumidora 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:

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

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 0.
- 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
   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:

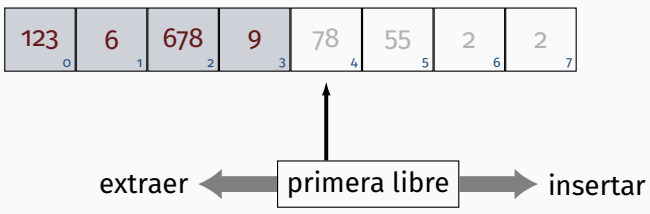
- LIFO (pila acotada), se usa una variable entera (≥ 0):
  - `primera_libre` = índice en el vector de la primera celda libre (inicialmente `0`). Esta variable se incrementa al escribir, y se decrementa al leer.
- FIFO (cola circular), se usan dos variables enteras no negativas:
  - `primera_ocupada`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

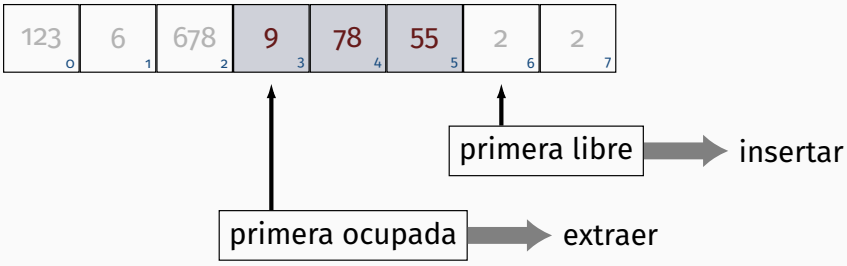
Soluciones para la gestión de la ocupación (2/2)

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.

Sección 3.
El problema de los (múltiples) productores y consumidores.

Múltiples productores y consumidores

En este ejercicio queremos extender la solución al problema del productor-consumidor, manteniendo todos los requerimientos del mismo, pero ahora:

- Queremos permitir un número de hebras productoras que no tiene que ser necesariamente la unidad (puede ser mayor).
- Igualmente podemos tener un número de hebras consumidoras superior a uno (no necesariamente igual al número de productoras)
- Varios productores pueden estar produciendo a la vez, y varios consumidores pueden estar consumiendo a la vez.
- La gestión del vector es similar a antes (haz una opción FIFO y una LIFO).

Actividad: múltiples productores y consumidores

Copia el archivo `prodcons.cpp` en `prodcons-multi.cpp`, y en este nuevo archivo adapta la implementación para permitir múltiples productores y consumidores.

Ten en cuenta estos requerimientos:

- El número de hebras productoras es una constante n_p, (> 0). El número de hebras consumidoras será otra constante n_c (> 0). Ambos valores deben ser divisores del número de items a producir m, y no tienen que ser necesariamente iguales. Se definen en el programa como dos constantes (con nombres descriptivos).
- Cada productor produce $p = m/n_p$ items. Cada consumidor consume $c = m/n_c$ items.
- Cada entero entre 0 y $m-1$ es producido una única vez (igual que antes).

Diseño de la solución: hebras y producción

Para poder cumplir los requisitos anteriores:

- Las funciones **`producir_dato`** y **`consumir_dato`** tienen ahora como argumento el número de hebra productora (o consumidora) que lo invoca (un valor i entre 0 y $n_p - 1$ (o entre 0 y $n_c - 1$), ambos incluidos.
- La hebra productora número i produce de forma consecutiva los p números enteros que hay entre el número $i \cdot p$ y el número $i \cdot p + (p - 1)$, ambos incluidos.
- Para esto, debemos tener un array compartido con n_p entradas que indique, en cada momento, para cada hebra productora, cuantos items ha producido ya. Este array se consulta y actualiza en **`producir_dato`**. Debe estar inicializado a 0. La hebra productora i es la única que usa la entrada número i (por tanto no hay requerimientos de EM en los accesos a este array).

Diseño de la solución: concurrencia

La solución debe solucionar los problemas de exclusión mutua **exclusivamente usando semáforos**, pero también debe permitir el mayor grado de paralelismo potencial posible. Ten en cuenta:

- En la solución LIFO:
  - Dos o más hebras pueden intentar a la vez leer y modificar la variable **`primera_libre`** y la correspondiente celda del vector.
- En la solución FIFO:
  - Dos o más hebras productoras pueden intentar leer y modificar a la vez la variable **`primera_libre`** y la correspondiente celda.
  - Dos o más hebras consumidoras pueden intentar leer y modificar a la vez la variable **`primera_ocupada`** y la correspondiente celda.

Sección 4.
El problema de los fumadores..

4.1. Descripción del problema.
4.2. Plantillas de código
4.3. Actividades y documentación.

Subsección **4.1.** 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 el fumador i ha retirado el ingrediente 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 **4.2.** 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 **4.3.** 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.

Fin de la presentación.

1.1.1. Resolución

Para ver los ficheros de la práctica 1 resueltos pincha aquí.

1.2. Práctica 2

UNIVERSIDAD DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2024-25 (archivo generado el 2 de octubre de 2024)

Grado en Ingeniería Informática,
Grado en Informática y Matemáticas,
Grado en Informática y Administración de Empresas.
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 varios casos prácticos.

Veremos cómo resolver con monitores SU un par de problemas ya resueltos con semáforos:

(1) Productores y consumidores múltiples.

(2) El problema de *Los fumadores*.

Veremos un problema nuevo, una variante de exclusión mutua con requerimientos adicionales:

(3) El problema de los *Lectores-escritores*.

Sistemas Concurrentes y Distribuidos, curso 2024-25.
Práctica 2. Casos prácticos de monitores en C++11.

Sección **1.**
Productores y consumidores múltiples.

Actividad: múltiples productores y consumidores

Copia el archivo `prodcons1_su.cpp` (desde el seminario 2) en `prodcons_mu.cpp` (en esta práctica 2), y en este nuevo archivo adapta la implementación para permitir múltiples productores y consumidores (llama a la clase **`ProdConsMu`**).

Ten en cuenta estos requerimientos (similares al problema equivalente en semáforos):

- El número de hebras productoras es una constante n_p, (> 0). El número de hebras consumidoras será otra constante n_c (> 0). Ambos valores deben ser divisores del número de items a producir m, y no tienen que ser necesariamente iguales. Se definen en el programa como dos constantes arbitrarias.
- Cada productor produce $p == m/n_p$ items. Cada consumidor consume $c == m/n_c$ items.
- Cada entero entre 0 y $m-1$ es producido una única vez (igual que antes).

Implementación

Para poder cumplir los requisitos anteriores:

- La función **`producir_dato`** tiene ahora como argumento el número de hebra productora que lo invoca (un valor i entre 0 y $n_p - 1$, ambos incluidos).
- La hebra productora número i produce de forma consecutiva los p números enteros que hay entre el número $i \cdot p$ y el número $i \cdot p + p - 1$, ambos incluidos.
- Para esto, debemos tener un array compartido con n_p entradas que indique, en cada momento, para cada hebra productora, cuántos items ha producido ya. Este array se consulta y actualiza en **`producir_dato`**. Debe estar inicializado a 0. La hebra productora i es la única que usa la entrada número i (por tanto no hay requerimientos de EM en los accesos a este array).

Sección 2.
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
```

Actividad.

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

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).

Sección 3.
El problema de los lectores y 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 leyendo 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 correspondientes **`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.

1.2.1. Resolución

Para ver los ficheros de la práctica 2 resueltos pincha aquí.

1.3. Práctica 3

UNIVERSIDAD DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2024-25 (archivo generado el 20 de noviembre de 2024)

Grado en Ingeniería Informática,
Grado en Informática y Matemáticas,
Grado en Informática y Administración de Empresas.
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.

1.1. Aproximación inicial
1.2. Solución con selección no determinista

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

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_`**

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

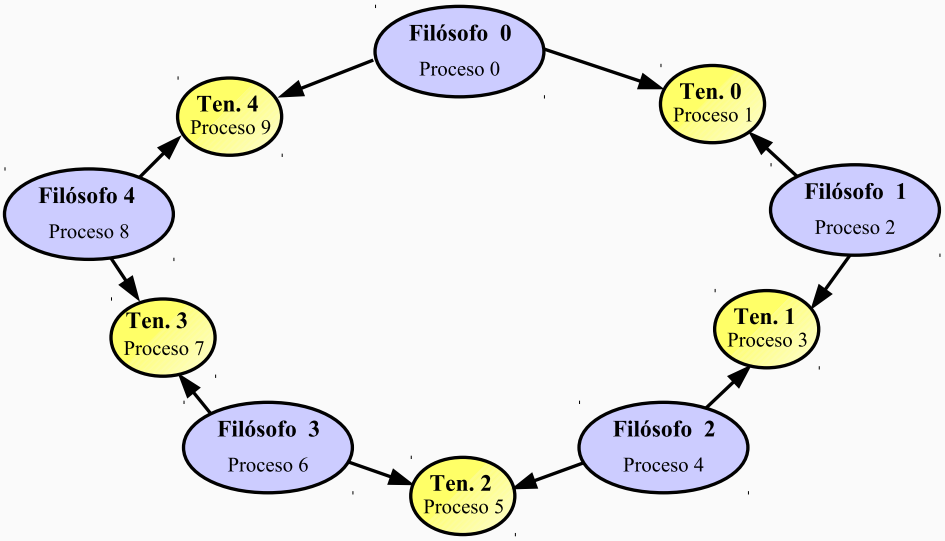
Por tanto, para implementar el problema, **debemos de ejecutar** 5 **procesos de tipo tenedor**, numerados del 0 al 4.

- 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.

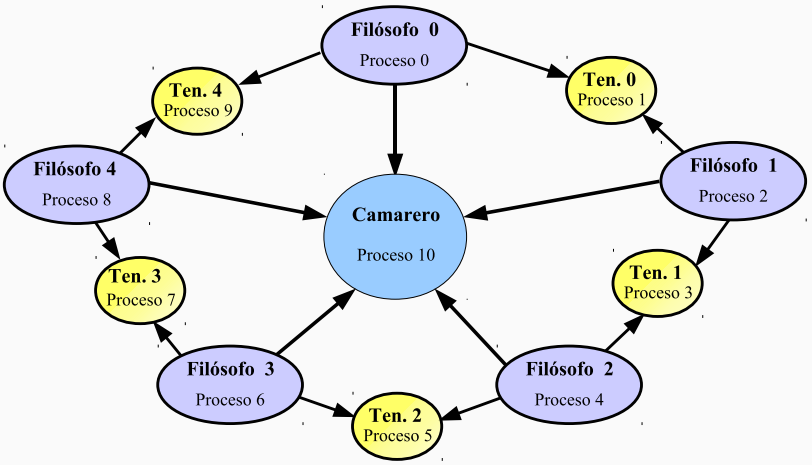
Sistemas Concurrentes y Distribuidos, curso 2024-25.
Práctica 3. Implementación de algoritmos distribuidos con MPI.
Sección 2. Cena de los Filósofos

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.

Fin de la presentación.

1.3.1. Resolución

Para ver los ficheros de la práctica 3 resueltos pincha aquí.

2 Seminarios

2.1. Seminario 1

UNIVERSIDAD DE GRANADA

Sistemas Concurrentes y Distribuidos: Seminario 1. Programación multihebra y semáforos.

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2024-25 (archivo generado el 18 de septiembre de 2024)

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

Seminario 1. Programación multihebra y semáforos.
Índice.

Introducción

Este seminario tiene cuatro partes, inicialmente se repasa el concepto de hebra, a continuación se da una breve introducción a la interfaz de las librerías de hebras y sincronización disponibles en C++ (versión 2011).

A continuación, se estudia el mecanismo de los semáforos como herramienta para solucionar problemas de sincronización y, por último, se hace una introducción a una librería para utilizar semáforos en C++.

- El objetivo es conocer algunas llamadas básicas de dicho interfaz para el desarrollo de ejemplos sencillos de sincronización con hebras usando semáforos (práctica 1)
- Las partes relacionadas con la estructura de las hebras están basadas en el texto disponible en esta web:
  ☞ https://computing.llnl.gov/tutorials/pthreads/

Sección 1.
Concepto e Implementaciones de Hebras.

Procesos: estructura

En un instante pueden existir muchos procesos ejecutándose concurrentemente, cada proceso corresponde a un programa en ejecución y ocupa una zona de memoria con (al menos) estas partes:

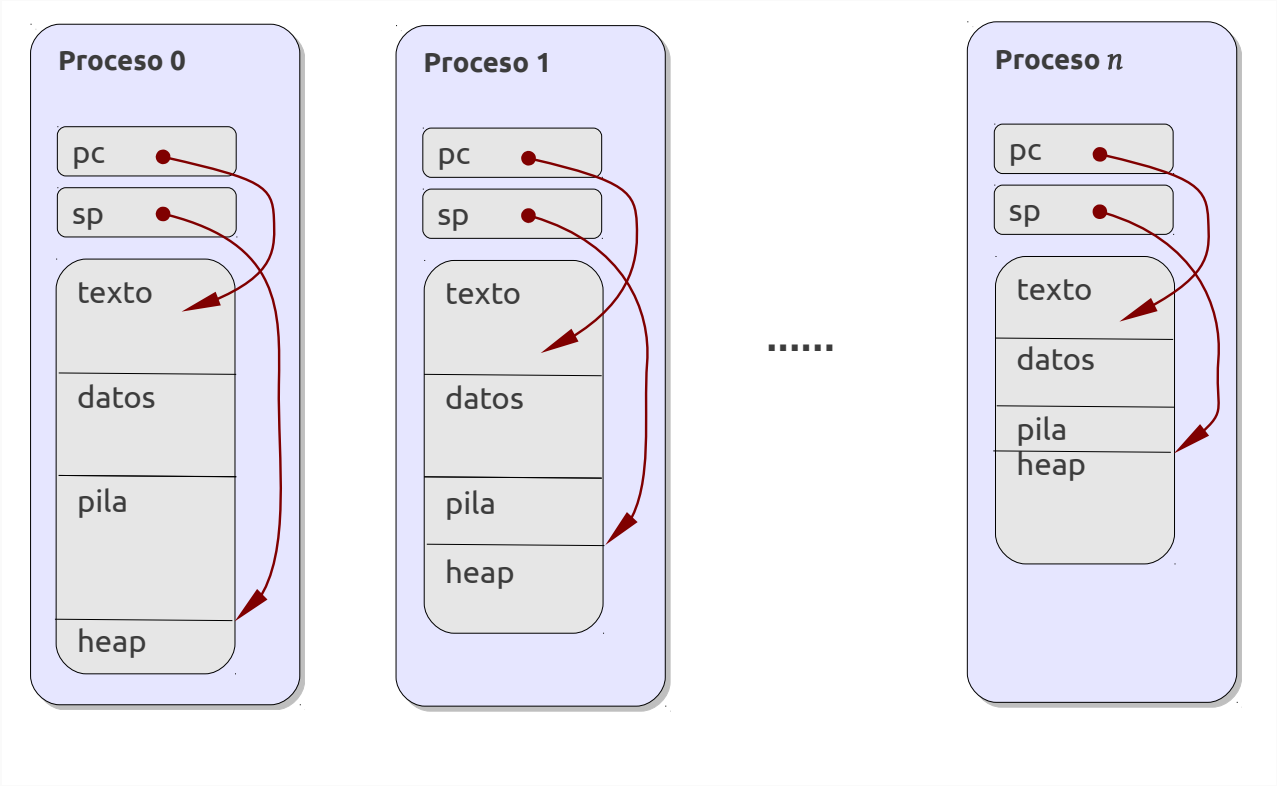
- **texto**: zona (tamaño fijo) con las instrucciones del programa
- **datos**: espacio (de tamaño fijo) para variables globales.
- **pila**: espacio (de tamaño cambiante) para variables locales.
- **mem. dinámica** (***heap***): espacio ocupado por variables dinámicas.

Cada proceso tiene asociados (entre otros) estos datos:

- **contador de programa** (**pc**): dirección en memoria (en la zona de texto) de la siguiente instrucción a ejecutar.
- **puntero de pila** (**sp**): dirección en memoria (en la zona de pila) de la última posición ocupada por la pila.

Diagrama de la estructura de los procesos

Podemos visualizarla (simplificadamente) como sigue:

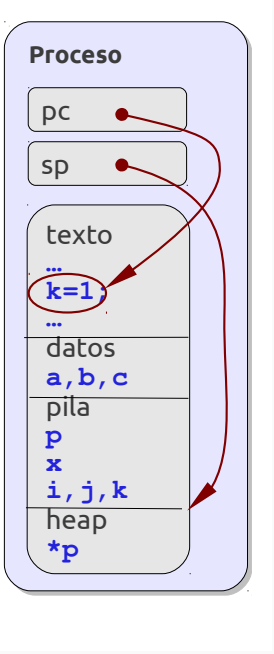


Ejemplo de un proceso

En el siguiente programa escrito en C/C++, el estado del proceso (durante la ejecución de **k=1;**) es el que se ve a la derecha:

```
int a,b,c ; // variables globales

void subprograma1()
{
   int i,j,k ; // vars. locales (1)
   k = 1 ;
}
void subprograma2()
{
   float x ; // vars. locales (2)
   subprograma1() ;
}
int main()
{
   char * p = new char ; // ”p”local
   *p = 'a'; // *p en el heap
   subprograma2() ;
}
```



Procesos y hebras

La gestión de varios procesos no independientes (cooperantes) es muy útil pero consume una cantidad apreciable de recursos del SO:

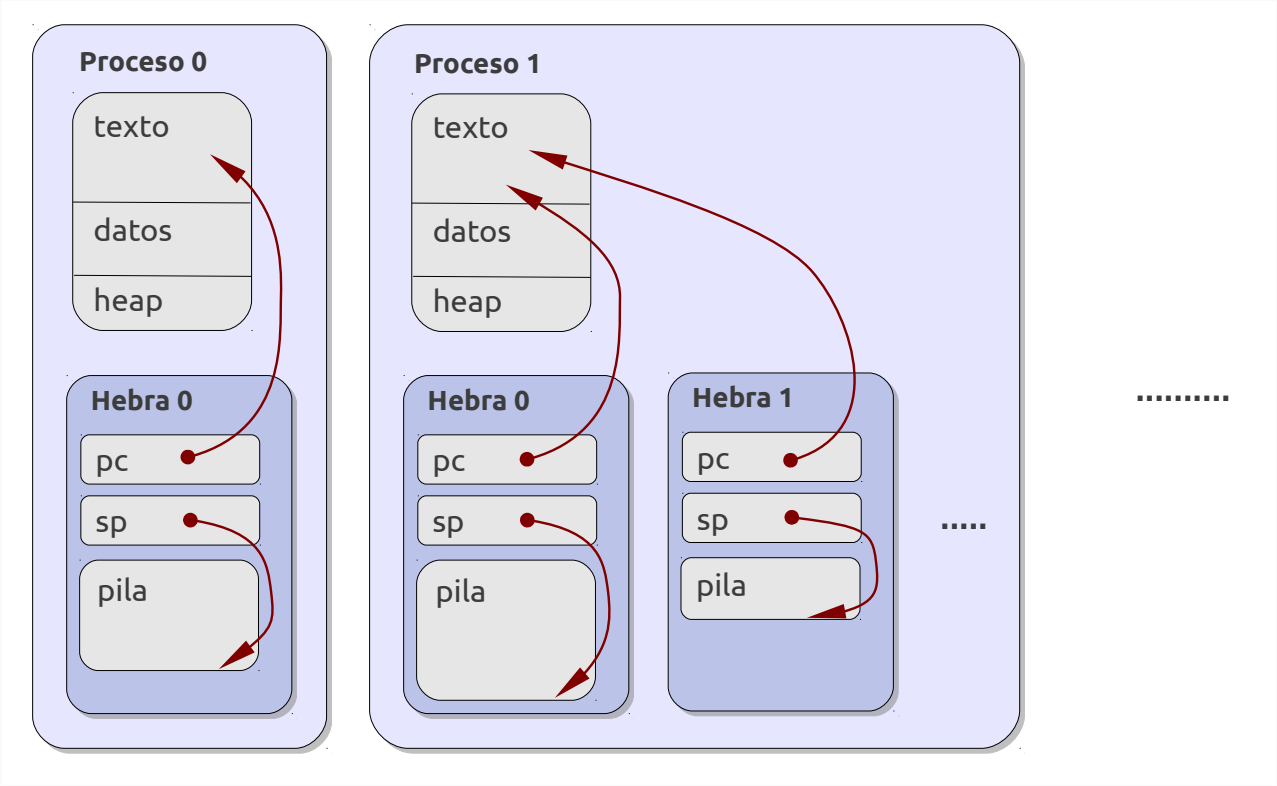
- Tiempo de procesamiento para repartir la CPU entre ellos
- Memoria con datos del SO relativos a cada proceso
- Tiempo y memoria para comunicaciones entre esos procesos

para mayor eficiencia en esta situación se diseñó el concepto de **hebra**:

- Un proceso puede contener una o varias hebras.
- Una hebra es un flujo de control en el texto (común) del proceso al que pertencen.
- Cada hebra tiene su propia pila (vars. locales), vacía al inicio.
- Las hebras de un proceso comparten la zona de datos (vars. globales), y el *heap*.

Diagrama de la estructura de procesos y hebras

Podriamos visualizarlos (simplificadamente) como sigue:



Inicio y finalización de hebras

Al inicio de un programa, existe una única hebra (que ejecuta la función **`main`** en C/C++). Durante la ejecución del programa:

- Una hebra A en ejecución puede crear otra hebra B en el mismo proceso de A
- Para ello, A designa un subprograma `f` (una función C/C++) del texto del proceso (y opcionalmente sus parámetros), y después continúa su ejecución. La hebra B:
  - ejecuta la función `f` concurrentemente con el resto de hebras.
  - termina normalmente cuando finaliza de ejecutar dicha función (bien ejecutando **`return`** o bien cuando el flujo de control llega al final de `f`)
- Una hebra puede esperar a que cualquier otra hebra en ejecución finalice (y opcionalmente puede obtener un valor resultado)

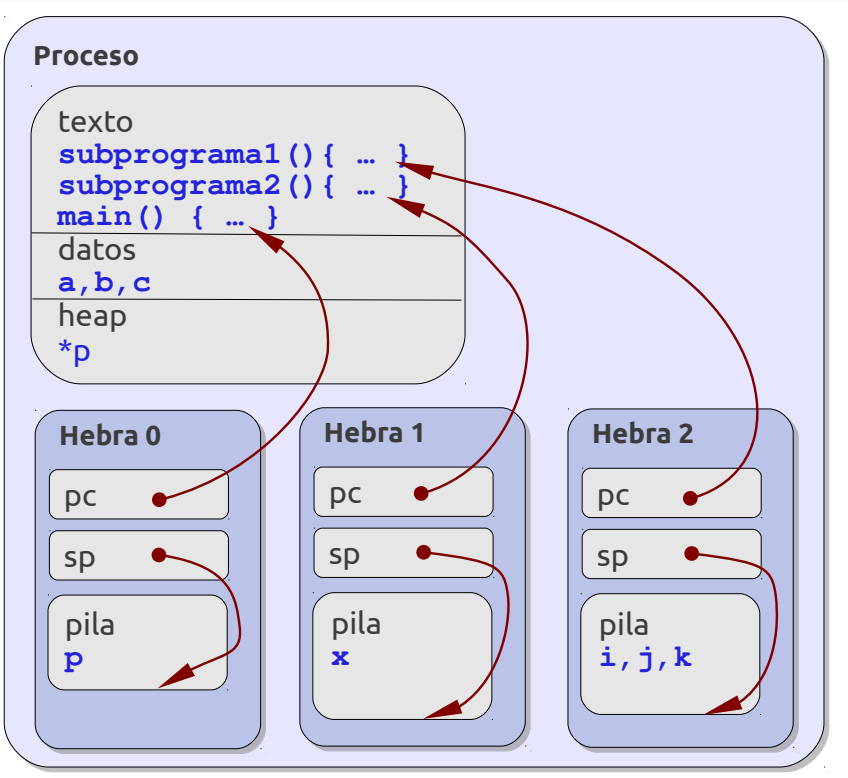
Ejemplo de estado de un proceso con tres hebras

En **main** se crean dos hebras, después se llega al estado que vemos:

```
int a,b,c ;

void subprograma1()
{
   int i,j,k ;
   // ...
}
void subprograma2()
{
   float x ;
   // ...
}

int main()
{
 char * p = new char ;
 // crear hebra (subprog.1)
 // crear hebra (subprog.2)
 // ...
}
```



Sección 2.
Hebras en C++11.

2.1. Introducción a las hebras en C++11
2.2. Compilación y ejecución desde la línea de órdenes
2.3. Creación y finalización de hebras
2.4. Sincronización mediante unión
2.5. Paso de parámetros y obtención de un resultado
2.6. Vectores de hebras y futuros
2.7. Medición de tiempos
2.8. Ejemplo de hebras: cálculo numérico de integrales

El estándard C++11

El acrónimo C++11 designa la versión del lenguaje de programación C++ publicada por ISO (la *International Standards Organization*) en Septiembre de 2011.

- Denominado oficialmente como estándard **ISO/IEC 14882:2011**:
  - Página del estándard en la web de ISO:
    ☞ https://www.iso.org/standard/50372.html
  - Borrador revisado en PDF:
    ☞ https://github.com/cplusplus/draft/blob/master/papers/n3337.pdf
- Hay revisiones posteriores de C++ (2014, 2017, 2020), pero no modifican las características que veremos aquí.
- Los fuentes C++ que usan este estándard son portables a Linux, macOS y Windows.
- Los compiladores de código abierto de GNU (g++) y del proyecto LLVM (clang++), así como *Visual C++* implementan este estándard.

Sistemas Concurrentes y Distribuidos, curso 2024-25.
Seminario 1. Programación multihebra y semáforos.
Sección **2. Hebras en C++11**

Subsección **2.1.** Introducción a las hebras en C++11.

Introducción

En esta sección veremos la funcionalidad básica para creacion y sincronización de hebras en el estándar C++11. El estándar define tipos de datos, clases y funciones para, entre otras muchas cosas:

- Crear una nueva hebra concurrente en un proceso, y esperar a que termine.
- Declaración de variables de *tipos atómicos*.
- Sincronización de hebras con *exclusión mutua*, *variables condición* y (en C++20) semáforos.
- Bloqueo de una hebra durante un intervalo de tiempo, o hasta un instante de tiempo.
- Generación de números aleatorios.
- Medición tiempos reales y de proceso, con alta precisión.

En este seminario veremos todas estas características (excepto las variables condición y los semáforos). Asimismo, se incluye información para compilar en Linux, macOS y Windows.

Subsección **2.2.** Compilación y ejecución desde la línea de órdenes.

Prerequisitos

Los programas fuente C++11 que vamos a usar o crear se pueden compilar y ejecutar en Linux, macOS y Windows sin modificación alguna. Se pueden instalar y usar estos compiladores:

- **Linux**: compilador C++ de GNU (`g++`), incorporado al paquete `build-essential` para `apt`. Alternativamente, se puede usar el compilador del proyecto LLVM (`clang++`), con el paquete `clang`.
- **macOS**: compiladores y entorno de desarrollo *XCode* (☞ developer.apple.com/xcode). Adicionalmente, se necesita el software *Command line Tools* (CLT), el cual se instala con: `xcode-select --install`.
- **Windows**: compiladores y entorno de desarrollo *Microsoft Visual Studio* (☞ visualstudio.microsoft.com), únicamente se necesita la componente para *desarrollo de aplicaciones de escritorio C/C++*.

Edición de archivos. Terminal a usar.

Para editar y compilar los fuentes hay que tener en cuenta estos aspectos:

- En macOS y Linux podemos usa cualquier tipo de aplicación de terminal y cualquier tipo de *shell* (*bash* u otras).
- En Windows debemos de usar un terminal de tipo *Developer Powershell* (o alternativamente *Developer Command Prompt*). Son terminales al uso, pero con las variables de entorno necesarias para ejecutar fácilmente el compilador o enlazador. Se instalan al instalar *Visual Studio*.
- Para editar los fuentes se puede usar cualquier editor de texto o entorno de desarrollo. En particular, el software de fuentes abiertas *VS Code* de Microsoft funciona bien en los tres sistemas operativos.

Codificación de archivos fuente.

Todos los archivos fuente C++ que se entregan están codificados **exclusivamente usando UTF-8**:

- Todos los fuentes que se entregan para evaluar deben estar codificados asimismo con UTF-8.
- Los editores de texto suelen estar configurados para reconocer automaticamente esta codificación.
- En Windows es necesario configurar el terminal *Powershell* de forma que, al ejecutar los programas compilados, los acentos (y otros caracteres especiales) se lean e impriman correctamente. Se puede hacer ejecutando esta orden una vez (es una única línea):

```
$OutputEncoding = [console]::InputEncoding = [console]::OutputEncoding =
                                    New-Object System.Text.UTF8Encoding
```

Compilar con la línea de órdenes en Linux y macOS

En Linux o macOS podemos compilar y enlazar con `g++` en la línea de órdenes un fuente, compuesto posiblemente de varios archivos `.cpp`, llamados f_1`.cpp`, f_2`.cpp` ... f_n`.cpp`. Usariamos esta órden:

```
g++ -std=c++11 -pthread -o ejecutable  f1.cpp f2.cpp ... fn.cpp
```

Esto crearía el archivo `ejecutable` en la carpeta de trabajo, se ejecuta con `./ejecutable` (adicionalmente)

Tambien se pueden compilar por separado los archivos (creando un archivo `.o` por cada `.cpp`) y enlazar todos los `.o` después:

```
g++ -std=c++11 -pthread -c f1.cpp
g++ -std=c++11 -pthread -c f2.cpp
      ......
g++ -std=c++11 -pthread -c fn.cpp
g++ -std=c++11 -pthread -o ejecutable f1.o f2.o ... fn.o
```

Se puede sustituir `g++` por `clang++` en macOS, y en Linux si se ha instalado *clang*.

Compilar con la línea de órdenes en Windows

En Windows la orden usa el compilador de Microsoft (`cl`), sería de esta forma:

```
cl /EHsc /Fe:ejecutable  f1.cpp f2.cpp ... fn.cpp
```

Esto creará el archivo `ejecutable.exe` en la carpeta de trabajo, se ejecuta con `./ejecutable`. También crea un archivo `.obj` por cada `.cpp` (no los usamos).

Tambien se pueden compilar por separado los archivos (creando un archivo `.obj` por cada `.cpp`) y enlazar todos los `.obj` después:

```
cl /EHsc /c f1.cpp
cl /EHsc /c f2.cpp
      ......
cl /EHsc /c fn.cpp
link /out:ejecutable.exe f1.obj f2.obj ... fn.obj
```

Subsección **2.3.** Creación y finalización de hebras.

Creación de hebras

El tipo de datos (o clase) **`std::thread`** permite definir objetos (variables) de *tipo hebra*. Un objeto (una variable) de este tipo puede contener información sobre una hebra en ejecución.

- En la declaración de la variable, se indica el nombre de la función que ejecutará la hebra
- En tiempo de ejecución, cuando se crea la variable, se comienza la ejecución concurrente de la función por parte de la nueva hebra.
- En la declaración se pueden especificar los parámetros de la nueva hebra.
- La variable sirve para poder referenciar a la hebra posteriormente.

Ejemplo de creación de hebras.

En este ejemplo (archivo **ejemplo01.cpp**) se crean dos hebras:

```
#include <iostream>
#include <thread>      // declaraciones del tipo std::thread
using namespace std ; // permite acortar la notación

void funcion_hebra_1(  ) // función que va a ejecutar la hebra primera
{ for( unsigned long i = 0 ; i < 5000 ; i++ )
     cout << "hebra 1, i == " << i << endl ;
}
void funcion_hebra_2(  )  // función que va a ejecutar la hebra segunda
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 2, i == " << i << endl ;
}
int main()
{
  thread hebra1( funcion_hebra_1 ), // crear hebra1 ejecutando funcion_hebra_1
         hebra2( funcion_hebra_2 ); // crear hebra2 ejecutando funcion_hebra_2

   // ... finalizacion ....
}
```

Este ejemplo **produce error** por finalización incorrecta.

Declaración e inicio separados

En el ejemplo anterior, las hebras se ponen en marcha cuando el flujo de control llega a la declaración de las variables tipo hebra:

```
thread hebra1( funcion_hebra_1 ), // crear hebra1 ejecutando funcion_hebra_1
       hebra2( funcion_hebra_2 ); // crear hebra2 ejecutando funcion_hebra_2
```

Sin embargo, también es posible declarar las variables y después poner en marcha las hebras. Para ello, en la declaración no incluimos la función a ejecutar:

```
thread hebra1, hebra2 ; // declaraciones (no se ejecuta nada)
....
hebra1 = thread( funcion_hebra_1 ); // hebra1 comienza funcion_hebra_1
hebra2 = thread( funcion_hebra_2 ); // hebra2 comienza funcion_hebra_2
```

Entre la declaración y el inicio (en los puntos suspensivos), las variables de tipo hebra no tienen asociada ninguna hebra en ejecución (esto permite variables tipo hebra globales).

Finalización de hebras

Una hebra cualquiera A que está ejecutando `f` finaliza cuando:

- La hebra A llega al final de `f`.
- La hebra A ejecuta un **`return`** en `f`.
- Se lanza una excepción que no se captura en `f` ni en ninguna función llamada desde `f`.
- Se destruye la variable tipo hebra asociada (es un situación de error, a evitar).

Todas las hebras en ejecución de un programa finalizan cuando:

- Cualquiera de ellas llama a la función **`exit()`** (o `abort`, o `terminate`), en este caso se termina el proceso completo.
- La hebra principal termina de ejecutar **`main`** (esta es una situación de error, que debemos de evitar, y que ocure en el ejemplo visto).

Sistemas Concurrentes y Distribuidos, curso 2024-25.
Seminario 1. Programación multihebra y semáforos.
Sección **2. Hebras en C++11**

Subsección **2.4.** Sincronización mediante unión.

Terminación incorrecta

En los ejemplos que hemos visto, la ejecución del programa no produce los resultados esperados (se obtiene un mensaje de error o no se imprimen todos los mensajes). El error se debe a la finalización incorrecta, en concreto, puede deberse a que:

- La hebra principal acaba **`main`** mientras las otras están ejecutándose
- Las variables tipo hebra (locales) **`hebra1`** y **`hebra2`** se destruyen cuando dichas hebras están ejecutándose.

En cualquier caso es necesario esperar a que las hebras **`hebra1`** y **`hebra2`** terminen antes de terminar el programa o la hebra principal

- Para ello, veremos la *operación de unión*, que es el primer mecanismo de sincronización de hebras que vamos a ver en este seminario.

La operación de unión.

C++11 provee diversos mecanismos para sincronizar hebras:

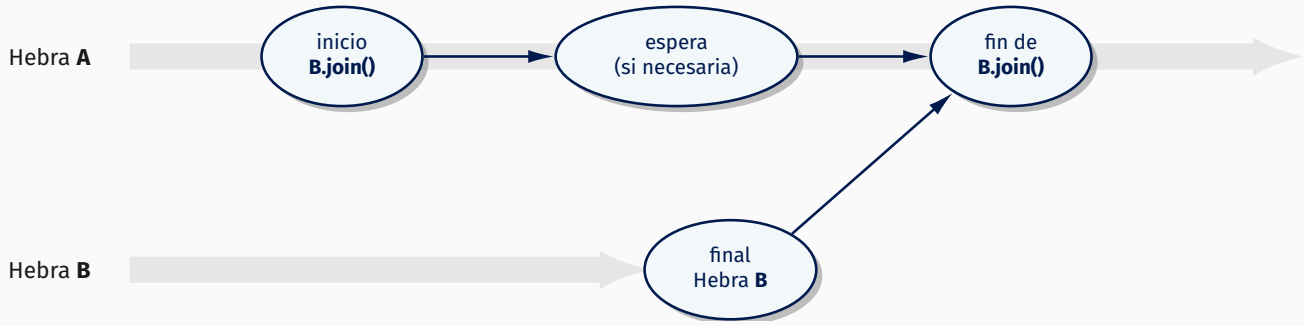
- Usando la operación de **unión** (***join***).
- Usando *mutex* o *variables condición*

La operación de unión permite que una hebra A espere a que otra hebra B termine:

- A es la hebra que invoca la unión, y B la hebra *objetivo*.
- Al finalizar la llamada, la hebra objetivo B ha terminado con seguridad.
- Si B ya ha terminado, no se hace nada.
- Si la espera es necesaria, se produce sin que la hebra que llama (A) consuma CPU durante dicha espera (A queda suspendida).

Sincronización asociada a una unión.

El grafo de dependencia de las tareas de las dos hebras ilustra la sincronización que se produce entre A y B



- El final de la operación de unión no puede ocurrir antes de que termine la hebra objetivo.
- Cualquier tarea ejecutada por A después de la unión, se ejecutará cuando B ya haya terminado.

El método *join* de la clase *thread*

Para que una hebra A espere hasta que termine una hebra objetivo B, la hebra A debe invocar el método **`join`** sobre la variable de tipo hebra asociada a la hebra B. En el ejemplo anterior, se haría así:

```
void funcion_hebra_1(  ) // función que va a ejecutar la hebra primera
{ for( unsigned long i = 0 ; i < 5000 ; i++ )
     cout << "hebra 1, i == " << i << endl ;
}
void funcion_hebra_2(  )  // función que va a ejecutar la hebra segunda
{  for( unsigned long i = 0 ; i < 5000 ; i++ )
      cout << "hebra 2, i == " << i << endl ;
}
int main()
{
  thread hebra1( funcion_hebra_1 ), // crear hebra1 ejecutando funcion_hebra_1
         hebra2( funcion_hebra_2 ); // crear hebra2 ejecutando funcion_hebra_2

  hebra1.join(); // la hebra principal espera a que hebra1 termine
  hebra2.join(); // la hebra principal espera a que hebra2 termine
}
```

Este ejemplo (archivo `ejemplo02.cpp`) sí funciona correctamente.

Uso correcto de la unión. Hebras activas.

La operación **`join`** solo puede invocarse sobre una **hebra activa**, es decir, que se encuentra en uno de estos dos casos:

- La hebra ha comenzado a ejecutarse y no ha terminado todavía.
- La hebra se ha ejecutado una vez y ha terminado, pero no se ha invocado todavía ningún otro **`join`** previo sobre ella.

En cualquier otro caso es incorrecto hacer unión (la hebra **no está activa**). Es decir, **no se puede invocar `join`**:

- Entre la declaración y el inicio (cuando se usa declaración e inicio por separado), ya que en ese intervalo no hay una hebra ejecutándose.
- Cuando ya se ha realizado un **`join`** sobre la hebra.

Cuando se intenta hacer **`join`** de una hebra no activa, se produce un error, esto se debe evitar.

Subsección 2.5.
Paso de parámetros y obtención de un resultado.

Parámetros y resultados

En lo que hemos visto hasta ahora, la función que ejecuta una hebra no tiene parámetros y no devuelve nada (el tipo devuelto es **`void`**).

- Se pueden usar funciones con parámetros. En este caso, al iniciar la hebra se deben de especificar los valores de los parámetros, igual que en una llamada normal.
- Si la función devuelve un valor de un tipo distinto de **`void`**, dicho valor es ignorado cuando se hace **`join`**
- Para poder obtener un valor resultado, hay varias opciones:
  - Uso de variables globales compartidas.
  - Uso de parámetros de salida (referencias o punteros).
  - Uso de la sentencia return, lanzando la hebra con **`async`**.

Paso de parámetros a hebras

Si la función tiene parámetros, al poner en marcha la hebra es necesario especificar valores para esos parámetros (después del nombre de la función). Por ejemplo, si tenemos estas declaraciones:

```
void funcion_hebra_1( int a, float x ) { .... }
void funcion_hebra_2( char * p, bool b ) { .... }
```

Debemos entonces iniciar las hebras dando los valores de los parámetros:

```
thread hebra1( funcion_hebra_1, 3+2, 45.678 ),    // a = 5, x=45.678
       hebra2( funcion_hebra_2, "hola!", true ); // p = "hola", b = true
```

o bien, usando declaración e inicio separados:

```
thread hebra1, hebra2 ;
...
hebra1 = thread( funcion_hebra_1, 3+2, 45.678 );    // a = 5, x = 45.678
hebra2 = thread( funcion_hebra_2, "hola!", true ); // p = "hola", b = true
```

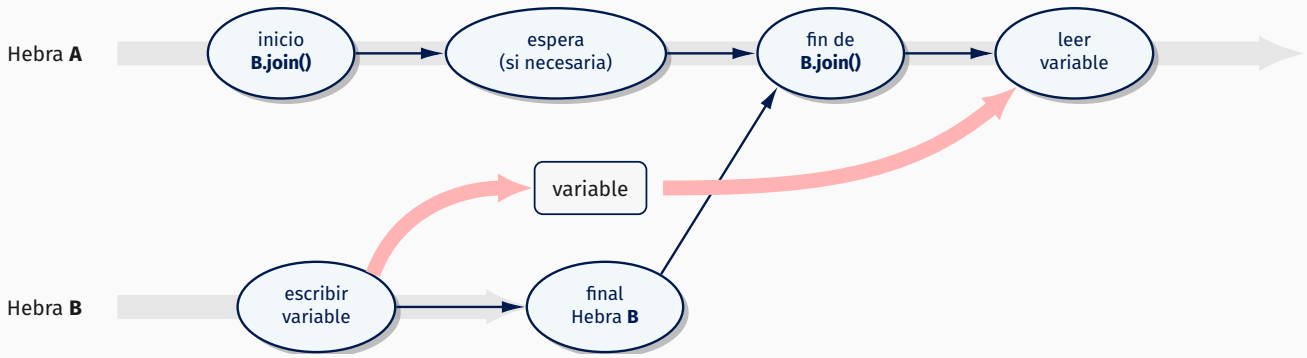
Métodos de obtención de valores resultado

Supongamos que una hebra A quiere leer un valor resultado, calculado por una hebra B que ejecuta una función `f`, hay estas formas de hacerlo:

- Mediante una **variable global** `v` (compartida): la función `f` (la hebra B) escribe el valor resultado en `v` y la hebra A lo lee tras hacer **`join`**. Esto constituye un *efecto lateral* (no recomendable).
- Mediante un **parámetro de salida** en `f` (puntero o referencia). La función `f` escribe en ese parámetro. La hebra A lee el resultado tras hacer **`join`**. También es un efecto lateral.
- Mediante **`return`**: la función `f` devuelve el valor resultado mediante **`return`**. La hebra A inicia B mediante la función **`async`**. Es la opción más simple y legible, y no obliga a diseñar `f` de forma que tenga efectos laterales-

Resultado en variable compartida. Sincronización.

Si se usa una variable compartida, la hebra que recoge el resultado (A) debe esperar a que la hebra que calcula el resultado haya finalizado, eso se consigue mediante una operación de unión. El grafo de dependencia de las tareas ejecutadas es el siguiente:



- El **`join`** ejecutado por A no termina antes de que B termine
- De esta forma aseguramos (por transitividad) que la lectura de la variable no ocurra nunca antes que su escritura.

Obtención de valores resultado

Supongamos que queremos que dos hebras calculen de forma concurrente el factorial de dos números, para ello disponemos de la función **factorial**, declarada como indica a continuación:

```
#include <iostream>
#include <future>      // declaracion de std::thread, std::async, std::future
using namespace std ; // permite acortar la notación (abc en lugar de std::abc)

// declaración de la función factorial (parámetro int, resultado long)
long factorial( int n ) { return n > 0 ? n*factorial(n-1) : 1 ; }
...
```

- Si usamos variables globales, necesitamos definir funciones de hebra que llaman a **factorial**, y dos variables globales distintas.
- Si usamos parámetros de salida, necesitamos una función de hebra que llama a **factorial**.
- Si se usa la función **async** se puede invocar directamente **factorial**.

Uso de variables globales

Usamos las variables compartidas **`resultado1`** y **`resultado2`** (archivo **`ejemplo03.cpp`**)

```
.......
// variables globales donde se escriben los resultados
long resultado1, resultado2 ;

// funciones que ejecutan las hebras
void funcion_hebra_1( int n ) { resultado1 = factorial( n ) ; }
void funcion_hebra_2( int n ) { resultado2 = factorial( n ) ; }

int main()
{
  // iniciar las hebras
  thread hebra1( funcion_hebra_1, 5  ), // calcula factorial(5) en resultado1
         hebra2( funcion_hebra_2, 10 ); // calcula factorial(10) en resultado2

  // esperar a que terminen las hebras,
  hebra1.join() ; hebra2.join() ;

  // imprimir los resultads:
  cout << "factorial(5)  == " << resultado1 << endl
       << "factorial(10) == " << resultado2 << endl ;
}
```

Uso de un parámetro de salida

Añadimos un parámetro de salida (por referencia) a la fun. de hebra (archivo `ejemplo04.cpp`)

```
.......
// función que ejecutan las hebras
void funcion_hebra( int n, long & resultado) { resultado= factorial(n); }

int main()
{
 long resultado1, resultado2 ; // variables (locales) con los resultados

 // iniciar las hebras (los parámetros por referencia se ponen con ref)
 thread hebra1( funcion_hebra, 5,  ref(resultado1) ), // calcula fact.(5)
        hebra2( funcion_hebra, 10, ref(resultado2) ); // calcula fact.(10)

 // esperar a que terminen las hebras,
 hebra1.join() ; hebra2.join() ;

 // imprimir los resultads:
 cout << "factorial(5)  == " << resultado1 << endl
      << "factorial(10) == " << resultado2 << endl ;
}
```

Uso de la función async

Lo más natural es que la función que ejecuta la hebra y que hace los cálculos devuelva el resultado usando la sentencia **`return`**.

- La función que ejecuta la hebra devuelve el resultado de la forma usual, mediante una sentencia **`return`**.
- La hebra se pone en marcha con una llamada a la función **`async`**, se especifica: el *modo*, el nombre de la función que ejecuta la hebra y sus parámetros, si los hay.
- El *modo* es una constante que indica que la función se debe ejecutar mediante una hebra concurrente específica para ello (hay otros modos de async que no estudiaremos)
- La llamada a **`async`** devuelve una variable (u objeto) de tipo *futuro* (**`future`**), ligada a la hebra que se pone en marcha.
- El tipo o clase **`future`** incorpora un método (**`get`**) para esperar a que termine la hebra y leer el resultado calculado.

Obtención de valores resultado mediante *futuros*

En este ejemplo (archivo **ejemplo05.cpp**) vemos como obtener los resultados directamente de la función **factorial**, sin tener que usar variables globales ni parámetros de salida. Se usa el método **get** de la clase **future**.

```
.......

int main()
{
  // iniciar las hebras y obtener los objetos future (conteniendo un long)
  // (la constante launch::async indica que se debe usar una hebra concurrente
  // para evaluar la función):
  future<long> futuro1 = async( launch::async, factorial, 5  ),
               futuro2 = async( launch::async, factorial, 10 );

  // esperar a que terminen las hebras, obtener resultado e imprimirlos
  cout << "factorial(5)  == " << futuro1.get() << endl
       << "factorial(10) == " << futuro2.get() << endl ;
}
```

Subsección **2.6.** Vectores de hebras y futuros.

Hebras idénticas

En muchos casos, un problema se puede resolver con un proceso en el que varias hebras distintas ejecutan la misma función con distintos datos de entrada. En estos casos

- Es necesario que cada hebra reciba parámetros distintos, lo cual permite que operen sobre datos distintos.
- Un caso muy común es que cada hebra reciba un número de orden o identificador de la hebra distinto, empezando en 0.
- Por simplicidad, se puede usar un vector de variables de tipo hebra o variables de tipo futuro.
- Lo anterior permite, además, que el número de hebras sea un parámetro configurable, sin cambiar el ćodigo.

Ejemplo de un vector de hebras

Supongamos que queremos usar n hebras idénticas para calcular concurrentemente el factorial de cada uno de los números entre 1 y n, ambos incluidos. Podemos usar un vector de **`thread`** para esto (archivo `ejemplo06.cpp`):

```
.....
const int num_hebras = 8 ; // número de hebras
// función que ejecutan las hebras: (cada una recibe i == índice de la hebra)
void funcion_hebra( int i )
{
  int fac = factorial( i+1 );
  cout <<"hebra número " <<i <<", factorial(" <<i+1 <<") = " <<fac <<endl;
}
int main()
{ // declarar el array de variables de tipo 'thread'
  thread hebras[num_hebras] ;
  // poner en marcha todas las hebras (cada una de ellas imprime el result.)
  for( int i = 0 ; i < num_hebras ; i++ )
    hebras[i] = thread( funcion_hebra, i ) ;
  // esperar a que terminen todas las hebras
  for( int i = 0 ; i < num_hebras ; i++ )
    hebras[i].join() ;
}
```

Ejemplo de un vector de futuros

En este caso, usamos un vector de futuros y la hebra principal imprime secuencialmente los resultados obtenidos (archivo `ejemplo07.cpp`)

```
.....

const int num_hebras = 8 ; // número de hebras

int main()
{
  // declarar el array de variables de tipo future
  future<long> futuros[num_hebras] ;

  // poner en marcha todas las hebras y obtener los futuros
  for( int i = 0 ; i < num_hebras ; i++ )
    futuros[i] = async( launch::async, factorial, i+1 ) ;

  // esperar a que acabe cada hebra e imprimir el resultado
  for( int i = 0 ; i < num_hebras ; i++ )
     cout << "factorial(" << i+1 << ") = " << futuros[i].get() << endl ;
}
```

Subsección 2.7.
Medición de tiempos.

Medición de tiempo real

En C++11 es posible medir la duración del intervalo de tiempo real empleado en cualquier parte de la ejecución de un programa.

- Estas medidas se basan en servicios del S.O., y son de alta precisión. C++11 proporciona una interfaz sencilla y portable para ello.
- Las mediciones se basan en dos tipos de datos (en `std::chrono`)
  - **Instantes en el tiempo**: tipo `time_point` (representado como tiempo desde un instante de inicio de un determinado *reloj*).
  - **Duraciones de intervalos de tiempo**: tipo `duration`. Una duración es la diferencia entre dos instantes de tiempo. Puede representarse con enteros o flotantes, y en cualquier unidad de tiempo (nanosegundos, microsegundos, milisegundos, segundos, minutos, horas, años, etc...).

Relojes

En C++11 se definen tres clases (tipos de datos) para tres relojes distintos. Son los siguientes:

- **Reloj del sistema**: (tipo `system_clock`). Tiempo indicado por la hora/fecha del sistema, y por tanto puede sufrir ajustes y cambios que lo hagan dar saltos hacia adelante o retroceder hacia a atrás.
- **Reloj monotónico**: (tipo `steady_clock`). Mide tiempo real desde un instante en el pasado, y no sufre saltos: nunca retrocede.
- **Reloj de alta precisión**: (tipo `high_resolution_clock`). Es el reloj de máxima precisión en el sistema, puede ser el mismo que uno de los dos anteriores o un tercero distinto.

Para medir tiempos, usaremos el reloj `steady_clock`

Medición de tiempos con el reloj monotónico

Usamos **now** para medir lo que tardan unas instrucciones (archivo `ejemplo08.cpp`):

```
#include <iostream>
#include <chrono>  // incluye now, time_point, duration
using namespace std ;
using namespace std::chrono;

int main()
{
  // leer instante de inicio de las instrucciones
  time_point<steady_clock> instante_inicio = steady_clock::now() ;
  // aquí se ejecutan las instrucciones cuya duración se quiere medir
  // .....
  // leer instante final de las instrucciones
  time_point<steady_clock> instante_final = steady_clock::now() ;
  // restar ambos instantes y obtener una duración (en microsegundos, flotantes)
  duration<float,micro> duracion_micros = instante_final - instante_inicio ;
  // imprimir los tiempos usando el método count
  cout << "La actividad ha tardado : "
       << duracion_micros.count() << " microsegundos." << endl ;
}
```

Subsección 2.8.
Ejemplo de hebras: cálculo numérico de integrales.

Cálculo numérico de integrales

La programación concurrente puede ser usada para resolver más rápidamente multitud de problemas, entre ellos los que conllevan muchas operaciones con números flotantes

- Un ejemplo típico es el cálculo del valor I de la integral de una función f de variable real (entre 0 y 1, por ejemplo) y valores reales positivos:

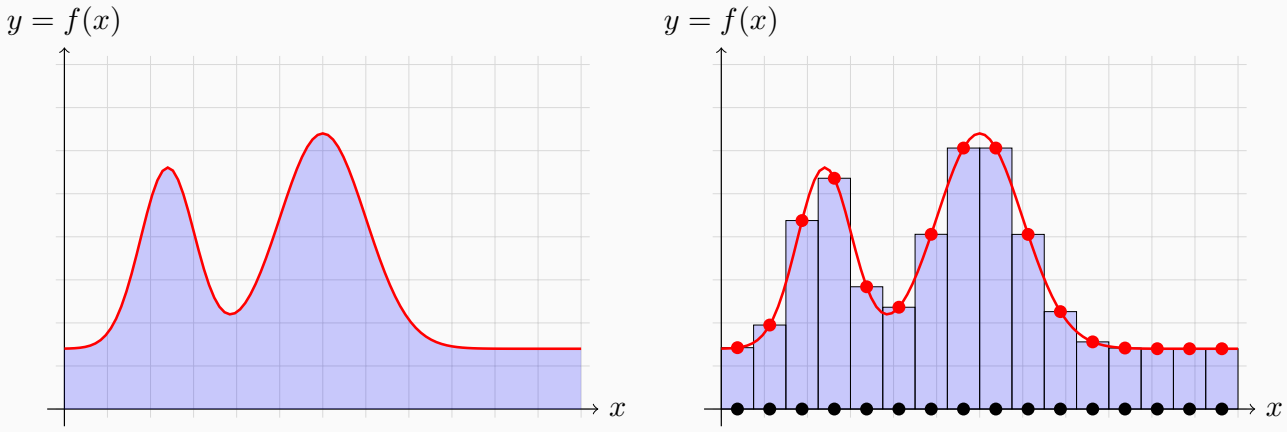
$$I = \int_0^1 f(x)\,dx$$

- El cálculo se puede hacer evaluando la función f en un conjunto de m puntos uniformemente espaciados en el intervalo $[0,1]$, y aproximando I como la media de todos esos valores:

$$I \approx \frac{1}{m}\sum_{j=0}^{m-1} f(x_j) \qquad \text{donde:} \quad x_j = \frac{j+1/2}{m}$$

Interpretación geométrica

Aproximamos el área azul (es I) (izquierda), usando la suma de las áreas de las m barras (derecha):



- Cada punto de muestra es el valor x_i (puntos negros)
- Cada barra tiene el mismo ancho $1/m$, y su altura es $f(x_i)$.

Cálculo secuencial del número π

Para verificar la corrección del método, se puede usar una integral I con valor conocido. A modo de ejemplo, usaremos una función f cuya integral entre 0 y 1 es el número π:

$$I = \pi = \int_0^1 \frac{4}{1+x^2}\,dx \qquad \text{aquí } f(x) = \frac{4}{1+x^2}$$

una implementación secuencial sencilla sería mediante esta función:

```
const long m = ..., n = ...; // el valor m es alto (del orden de millones)
// implementa función f
double f( double x )
{ return 4.0/(1+x*x) ; // f(x) = 4/(1+x²)
}
// calcula la integral de forma secuencial, devuelve resultado:
double calcular_integral_secuencial(  )
{  double suma = 0.0 ;                        // inicializar suma
   for( long j = 0 ; j < m ; j++ )            // para cada j entre 0 y m-1:
   {  const double xj = double(j+0.5)/m;      //   calcular x_j
      suma += f( xj );                        //   añadir f(x_j) a suma
   }
   return suma/m ;                            // devolver valor promedio de f
}
```

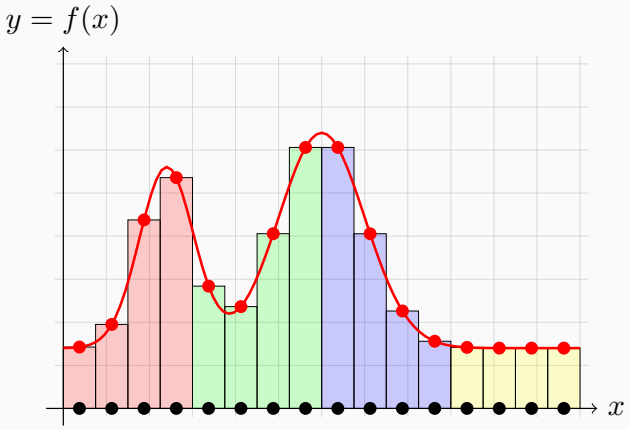
Versión concurrente de la integración

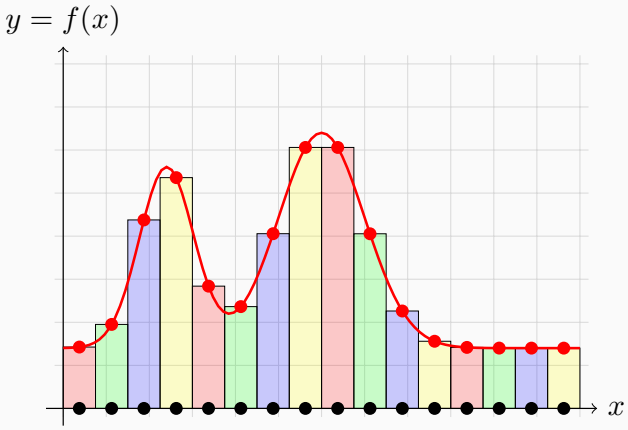
El cálculo citado anteriormente se puede hacer mediante un total de n hebras idénticas (asumimos que m es múltiplo de n)

- Cada una de las hebras evalua f en m/n puntos del dominio
- La cantidad de trabajo es similar para todas, y los cálculos son independientes.
- Cada hebra calcula la suma parcial de los valores de f
- La hebra principal recoge las sumas parciales y calcula la suma total.
- En un entorno con k procesadores o núcleos, el cálculo puede hacerse hasta k veces más rápido. Esta mejora ocurre solo para valores de m varios órdenes de magnitud más grandes que n.

Distribución de cálculos

Para distribuir los cálculos entre hebras, hay dos opciones simples, hacerlo de forma **contigua** (izquierda) o de forma **entrelazada** (derecha)





Cada valor $f(x_i)$ es calculado por:

- la hebra número i/n (en la opción contigua).
- la hebra número $i \mod n$ (en la opción entrelazada).

Esquema de la implementación concurrente

La función que ejecutará cada hebra recibe `ih`, el índice de la hebra, que va desde 0 hasta $n-1$ (ambos incluidos). Devuelve la sumatoria parcial correspondiente a las muestras calculadas:

```
double funcion_hebra( long ih )
{ .......
}
```

La función que calcula la integral de forma concurrente lanza n hebras (con **`async`**), y crea un vector de **`future`**. La hebra principal espera que vayan acabando, obtiene las sumas parciales y devuelve la suma total:

```
double calcular_integral_concurrente(  )
{ ......
}
```

Hebra principal

La función **main** (que será ejecutada por la hebra principal) tiene la forma que vemos aquí (archivo **ejemplo09-plantilla.cpp**)

```
....
int main(  )
{
  const double pi = 3.14159265358979312 ; // valor de π con bastantes decimales

  // hacer los cálculos y medir los tiempos:
  ...
  const double pi_sec  = calcular_integral_secuencial( );
  ...

  ...
  const double pi_conc = calcular_integral_concurrente( );
  ...

  // escribir en cout los resultados:
  ...
}
```

Actividad: medición de tiempos de cálculo concurrente.

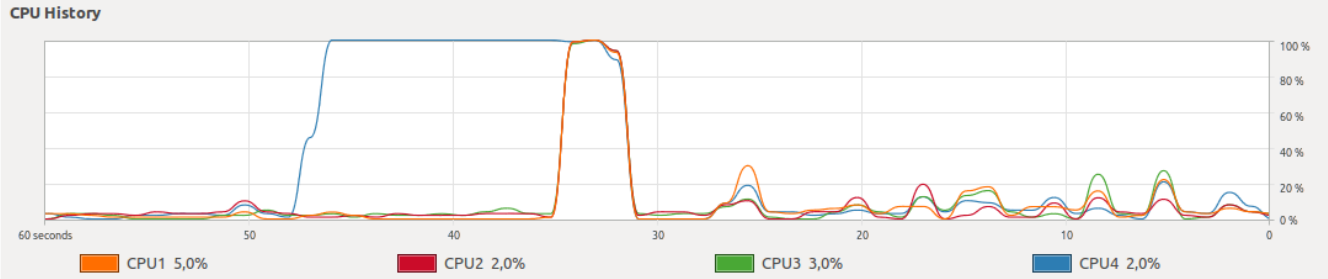
Como actividad se propone copiar la plantilla en `ejemplo09.cpp` y completar en este archivo la implementación del cálculo concurrente del número π, tal y como hemos visto aquí:

- En la salida se presenta el valor exacto de π y el calculado de las dos formas (sirve para verificar si el programa es correcto).
- Asimismo, el programa imprimirá la duración del cálculo concurrente, del secuencial y el porcentaje de tiempo concurrente respecto del secuencial, como se ve aquí:

```
Número de muestras (m)   : 1073741824
Número de hebras (n)     : 4
Valor de PI              : 3.14159265358979312
Resultado secuencial     : 3.14159265358998185
Resultado concurrente    : 3.14159265358978601
Tiempo secuencial        : 11576 milisegundos.
Tiempo concurrente       : 2990.6 milisegundos.
Porcentaje t.conc/t.sec. : 25.83%
```

Resultados de la actividad

En esta figura vemos (en un sistema Ubuntu 16 con 4 CPUs) como van evolucionando los porcentajes de uso de cada CPU a lo largo de la ejecución del programa con 4 hebras (es una captura de pantalla del monitor del sistema (*system monitor*)):



- Parte secuencial: la hebra principal ejecuta la versión secuencial, y ocupa al 100 % una CPU (CPU4, línea azul).
- Parte concurrente: Las 4 hebras creadas por la principal ocupan cada una CPU al 100 %, la hebra principal espera.

Por tanto, el cálculo concurrente tarda **un poco más de la cuarta parte** que el secuencial.

Sección 3.
Sincronización básica en C++11.

3.1. Tipos de datos atómicos
3.2. Objetos *Mutex*

Introducción

En esta sección veremos algunas de las posibilidades básicas que ofrece C++11 para la sincronización de hebras. Son estas dos:

- **Tipos atómicos**: tipos de datos (típicamente enteros) cuyas variables se pueden actualizar de *forma atómica*, es decir, en exclusión mutua.
- **Objetos *mutex***: son variables (objetos) que incluyen operaciones que permiten garantizar la exclusión mutua en la ejecución de trozos de código (secciones críticas)

Existen otros tipos de mecanismos de sincronización en C++11, algunos los veremos más adelante.

Accesos concurrentes a datos compartidos

La interfoliación de las operaciones de consulta y actualización de variables compartidas entre hebras concurrentes puede dar lugar a resultados distintos de los esperados o incorrectos. Por ejemplo:

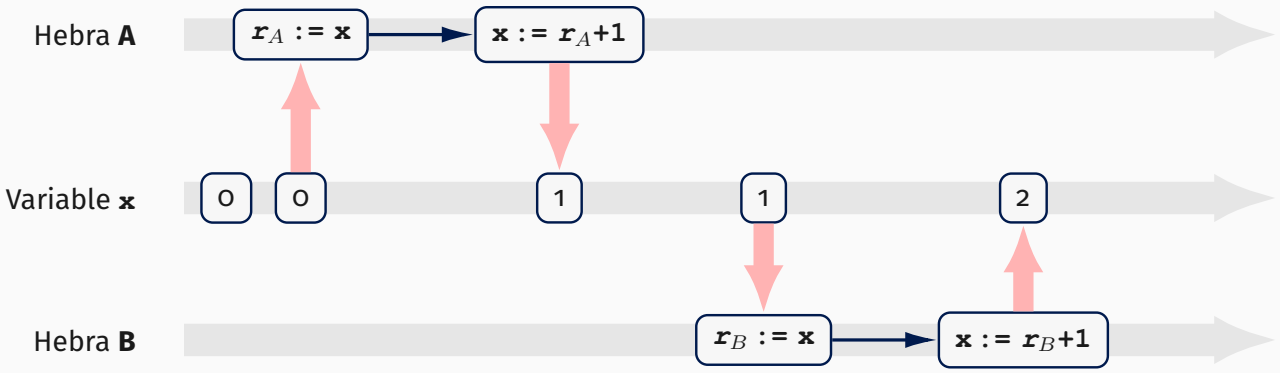
- Incrementar o decrementar una variable entera o flotante se hace en varias instrucciones atómicas distintas. En particular, puede que dos incrementos simultáneos de una variable entera dejen la variable con una unidad más en lugar de dos unidades más, como cabe esperar.
- Insertar o eliminar un nodo de una lista o un árbol. Por ejemplo, puede que dos inserciones simultáneas produzcan que uno de los dos nodos no quede insertado.

En el primer caso, se pueden usar *tipos atómicos*, mientras que en el segundo se pueden usar *objetos mutex*

Ejemplo: incrementos concurrentes de una variable (1/2)

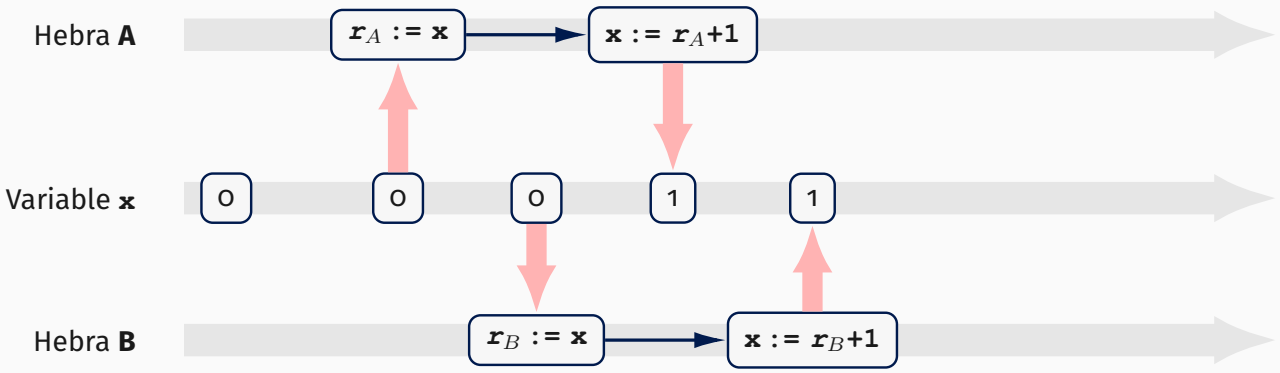
A modo de ejemplo, supongamos que dos hebras A y B ejecutan concurrentemente la sentencia `x++` sobre una variable compartida (global) `x`, que está inicializada a 0. Cada hebra usa su propio registro ($\mathtt{r}_A$ y $\mathtt{r}_B$), y hace el incremento mediante tres instrucciones atómicas (lectura + incremento + escritura)

Si una hebra lee después de que la otra escriba, el valor final de `x` es 2:



Ejemplo: incrementos concurrentes de una variable (2/2)

Sin embargo, si las dos hebras leen antes de que ninguna escriba, el valor final de x es 1



Por tanto, el efecto de los incrementos está indeterminado, ya que puede ocurrir cualquier interfoliación. Esto se debe a que **la sentencia x++ no es atómica**, es decir **no se ejecuta en exclusión mutua**.

Subsección 3.1.
Tipos de datos atómicos.

Tipos atómicos en C++11. Métodos y operadores

Para cada tipo entero T (**int**, **unsigned**, etc..) existe un *tipo atómico* **atomic**<T> (o **atomic_**T), **que es equivalente** a T. Si **a** es una variable **atomic**<T> y e es una expresión de tipo T, entonces evaluar la expresión **a.fetch_add(**e**)** (o la expresión **a.fetch_sub(**e**)**) supone dar estos pasos:

(a) Evaluar la expresión e y obtener su valor, n.

(b) Ejecutar estos tres pasos **de forma atómica**:

1. Leer el valor entero actual de **a**.
2. Incrementar (o decrementar) el valor de **a** en n unidades.
3. Devolver el valor leído en el paso 1.

Se pueden usar operadores tradicionales de C:

a++ ≡ a.fetch_add(1)	++a ≡ a.fetch_add(1)+1
a-- ≡ a.fetch_sub(1)	--a ≡ a.fetch_sub(1)-1
a+=e ≡ a.fetch_add(n)+n	a-=e ≡ a.fetch_sub(n)-n

Atomicidad de las operaciones.

La diferencia clave entre **`atomic`**<T> y T está en la *atomicidad*:

- Estos dos métodos se ejecutan de **de forma atómica**, es decir, **en exclusión mutua** entre todas las hebras que estén operando sobre una misma variable atómica `a`.
- La evaluación de la expresión n ocurre antes de la E.M., no dentro de E.M. (por ejemplo en `a+= x*y+2`).
- Si la arquitectura de la CPU lo permite, los métodos se implementan con **una única instrucción de código máquina**. Son instrucciones específicamente diseñadas para esto.
- Si la arquitectura de la CPU no lo permite, se usan otros métodos para E.M., mucho menos eficientes en tiempo y memoria (p.ej. se pueden usar *objetos mutex*, que veremos a continuación).
- La mayoría de las arquitecturas de CPU modernas incorpora instrucciones de código máquina específicas, al menos para el tipo **`atomic`**`<int>`.

Ejemplo de tipos atómicos (1/2)

Aquí comparamos incrementos atómicos frente a no atómicos (archivo `ejemplo10.cpp`)

```
#include <iostream>
#include <thread>
#include <chrono>
#include <atomic>       // incluye la funcionalidad para tipos atómicos
using namespace std ;
using namespace std::chrono ;

const long  num_iters = 1000000l ; // número de incrementos a realizar
int         contador_no_atom  ;    // contador compartido (no atomico)
atomic<int> contador_atom ;        // contador compartido (atomico)

void funcion_hebra_no_atom(  )  // incrementar el contador no atómico
{ for( long i = 0 ; i < num_iters ; i++ )
    contador_no_atom ++ ; // incremento no atómico de la variable
}
void funcion_hebra_atom(  )    // incrementar el contador atómico
{ for( long i = 0 ; i < num_iters ; i++ )
    contador_atom ++ ; // incremento atómico de la variable
}
....
```

Ejemplo de tipos atómicos (2/2)

```
.....
int main()
{
  // poner en marcha dos hebras que hacen los incrementos atómicos
  contador_atom = 0 ; // inicializa contador atómico compartido
  thread hebra1_atom = thread( funcion_hebra_atom ),
         hebra2_atom = thread( funcion_hebra_atom );
  hebra1_atom.join();
  hebra2_atom.join();

  // poner en marcha dos hebras que hacen los incrementos no atómicos
  contador_no_atom = 0 ; // inicializa contador no atómico compartida
  thread hebra1_no_atom = thread( funcion_hebra_no_atom ),
         hebra2_no_atom = thread( funcion_hebra_no_atom );
  hebra1_no_atom.join();
  hebra2_no_atom.join();

  // escribir resultados
  // .......
}
```

Resultados del ejemplo

Aquí vemos los resultados obtenidos al ejecutar el ejemplo:

```
valor esperado       : 2000000
resultado (atom.)    : 2000000
resultado (no atom.) : 1202969
tiempo atom.         : 35.2199 milisegundos.
tiempo no atom.      : 6.50903 millisegundos.
```

- Los incrementos atómicos producen el valor final de `contador_atom` esperado, esto es, el doble del número de iteraciones (`2*num_iters`).
- Los incremento no atómicos producen un valor final inferior al esperado, esto se debe a las interferencias entre los incrementos concurrentes (muchos incrementos de una hebra no tienen efecto al ser sobreescrita después la variable por la otra hebra).
- Hay una diferencia en los tiempos significativa (¿ a que se debe esta diferencia ?)

Sistemas Concurrentes y Distribuidos, curso 2024-25.
Seminario 1. Programación multihebra y semáforos.
Sección **3. Sincronización básica en C++11**

Subsección **3.2.** Objetos *Mutex*.

Introducción

En muchos casos las operaciones complejas sobre estructuras de datos compartidas se deben hacer en *exclusión mutua* en trozos de código llamados *secciones críticas*, y en estos casos no se pueden usar simples operaciones atómicas.

- Para ello podemos usar los **objetos mutex** (también llamados *cerrojos (locks)*).
- El estándard C++11 contempla el tipo o clase `mutex` para esto. Para cada sección crítica (SC), usamos un objeto de este tipo.
- Las variables de tipo mutex suelen residir en memoria compartida, ya cada una de ellas debe ser usada por más de una hebra concurrente.
- Estas variables permiten exclusión mutua mediante espera bloqueda.

Operaciones sobre variables mutex

Las dos únicas operaciones que se pueden hacer sobre un objeto tipo *mutex* (tipo **`std::mutex`**) son **`lock`** y **`unlock`**:

- **`lock`**
  Se invoca al inicio de la SC, y la hebra espera si ya hay otra ejecutando dicha SC. Si ocurre la espera, la hebra no ocupa la CPU durante la misma (queda bloqueada).
- **`unlock`**
  Se invoca al final de la SC para indicar que ha terminado de ejecutar dicha SC, de forma que otras hebras puedan comenzar su ejecución.

Entre las operaciones **`lock`** y **`unlock`**, decimos que la hebra *tiene adquirido* (o *posee*) el mutex. El método **`lock`** permite adquirir el mutex, y el método *unlock* permite liberarlo. Un mutex está libre o adquirido por una única hebra. Una hebra no debe intentar adquirir un mutex que ya posee.

Ejemplo de uso de un objeto mutex

En el ejemplo de un vector de hebras que calculan e imprimen el factorial de un número, las salidas en pantaalla aparecen mezcladas. Esto se puede evitar usando un objeto **mutex** compartido (archivo `ejemplo12.cpp`):

```
#include <iostream>
#include <thread>
#include <mutex>    // incluye clase mutex
using namespace std ;

mutex mtx ; // declaración de la variable compartida tipo mutex

void funcion_hebra_m( int i ) // función que ejecutan las hebras (con mutex)
{
  int fac = factorial( i+1 );
  mtx.lock();  // adquirir el mutex
  cout <<"hebra número " <<i <<", factorial(" <<i+1 <<") = " <<fac <<endl;
  mtx.unlock(); // liberar el mutex
}
```

Eficiencia de los mutex y los tipos atómicos

En el ejemplo (en el archivo `ejemplo11.cpp`) se comparan los tiempos de cálculo del ejemplo del contador, pero ahora usando tambien objetos mutex. Se obtienen estos resultados:

```
valor esperado       : 2000000
resultado (mutex)    : 2000000
resultado (atom.)    : 2000000
resultado (no atom.) : 1222377
tiempo mutex         : 7001.01 milisegundos
tiempo atom.         : 39.5807 milisegundos.
tiempo no atom.      : 7.67227 millisegundos.
```

Como puede observarse, el tiempo para el caso de los objetos mutex es mucho mayor que el uso de instrucciones atómicas. Razona en tu portafolio a que se debe esto

Sección 4.
Introducción a los Semáforos.

4.1. Estructura, operaciones y propiedades
4.2. Espera única
4.3. Exclusión mutua
4.4. Productor-Consumidor (lectura/escritura repetidas)

Semáforos

Los **semáforos** constituyen un mecanismo de nivel medio que permite solucionar los problemas derivados de la ejecución concurrente de procesos no independientes. Sus características principales son:

- permite bloquear los procesos sin mantener ocupada la CPU
- resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos
- se pueden usar para resolver problemas de sincronización (aunque en ocasiones los esquemas de uso son complejos)
- el mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos.

Subsección **4.1.** Estructura, operaciones y propiedades.

Estructura de un semáforo

Un semáforo es un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (se dice que están esperando en el semáforo).
- Un valor natural (entero no negativo), al que llamaremos *valor del semáforo*

Estas estructuras de datos residen en memoria compartida. Al principio de un programa que use semáforos, debe poder inicializarse cada uno de ellos:

- el conjunto de procesos asociados (bloqueados) estará vacío
- se deberá indicar un valor inicial del semáforo

Operaciones sobre los semáforos

Además de la inicialización, solo hay dos operaciones básicas que se pueden realizar sobre una variable de tipo semáforo (que llamamos s) :

- `sem_wait`(s)
  - Si el valor de s es cero, esperar a que el valor sea mayor que cero (durante la espera, el proceso se añade a la lista de procesos bloqueados del semáforo).
  - Decrementar el valor de s en una unidad.
- `sem_signal`(s)
  - Incrementar el valor de s en una unidad.
  - Si hay procesos esperando en la lista de procesos de s, permitir que uno de ellos salga de la espera y continue la ejecución (ese proceso decrementará el valor del semáforo).

Propiedades de los semáforos

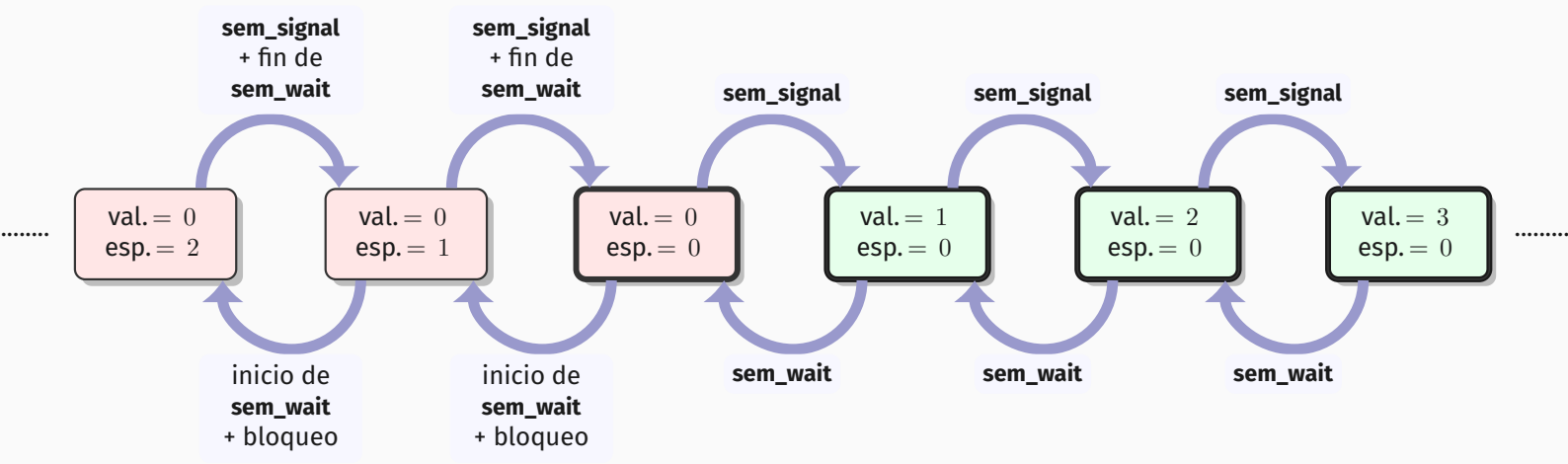
Los semáforos cumplen estas propiedades:

- El valor nunca es negativo (ya que se espera a que sea mayor que cero antes de decrementarlo).
- Solo hay procesos esperando cuando el valor es cero (con un valor mayor que cero, los procesos no esperan en **`sem_wait`**).
- El valor de un semáforo indica cuantas llamadas a **`sem_wait`**(sin **`sem_signal`**entre ellas) podrían ejecutarse en ese momento sin que ninguna haga esperar.

Para cumplir estas propiedades, la implementación debe de asegurar que las operaciones sobre el semáforo deben de ejecutarse en exclusión mutua (excepto cuando un proceso queda bloqueado).

Diagrama de estados de un semáforo

Vemos algunos posibles **estados** de un semáforo, según su número de procesos esperando, (esp.) y su valor (val.), y las posibles **transiciones atómicas** (en E.M.) entre esos estados, provocadas por hebras que invocan **sem_wait** o **sem_signal**



- Los estados con esp. $= 0$ son posibles estados iniciales.
- El color refleja si el semáforo está *en rojo* o *en verde*.

Patrones de solución de problemas de sincronización

Consideramos tres problemas típicos sencillos de sincronización, y vemos como se puede resolver cada uno usando patrones de programación que recurren a semáforos. Los tres problemas son:

- Espera única (Productor/Consumidor con una escritura y una lectura)
- Exclusión mutua.
- Productor/Consumidor con lecturas y escrituras repetidas.

Para poder diseñar las soluciones, en cada caso **relacionamos el valor del semáforo en un momento dado con la interfoliación ocurrida hasta llegar a ese momento**.

Sistemas Concurrentes y Distribuidos, curso 2024-25.
Seminario 1. Programación multihebra y semáforos.
Sección 4. Introducción a los Semáforos

Subsección 4.2. Espera única.

Problema de sincronización

El problema básico de sincronización ocurre cuando:

- Un proceso **P2** no debe pasar de un punto de su código hasta que otro proceso **P1** no haya llegado a otro punto del suyo.
- El caso típico es: **P1** debe escribir una variable compartida y **después P2** debe leerla.

```
{ variables compartidas y valores iniciales }
var compartida  : integer ; { variable compartida: P1 escribe y P2 lee }
```

```
process P1 ;
   var local1 : integer ;
begin
 .....
 local1 := ProducirValor() ;
 compartida := local1; {sentencia E}
 .....
end
```

```
process P2 ;
 var local2 : integer ;
begin
 .....
 local2 := compartida; {sentencia L}
 UsarValor( local2 );
 .....
end
```

Este programa **no funciona correctamente**.

Condición de sincronización

Para que el programa anterior funcione correctamente, la sentencia de escritura (que llamamos E) debe terminar antes de que empiece la sentencia de lectura (que llamamos L).

- La **condición de sincronización** es la siguiente: queremos evitar la interfoliación L, E, y solo permitimos la interfoliación E, L (por la estructura del programa, no hay más opciones).
- Para asegurar que se cumple la condición, hay que introducir una espera bloqueada en **P2**, inmediatamente previa a la lectura L, cuando en ese instante todavía no se haya completado E. La espera, si ocurre, debe durar hasta que se complete E.
- Por tanto, usaremos un semáforo que nos asegure esto. La espera se implementa con una llamada de **P2** a `sem_wait`. Además usamos una llamada desde **P1** a `sem_signal` para desbloquear a **P2** cuando corresponda.

Solución con un semáforo

El valor del semáforo puede ser únicamente 0 o 1. En concreto, será:

- 1 después de que se haya completado E pero antes de que comience L (hay un valor pendiente de leer).
- 0 en cualquier otro caso.

Por tanto:

- El valor inicial del semáforo debe ser 0, ya que al inicio no se ha completado E todavía.
- Inmediatamente **después de** E, **P1** debe invocar a **`sem_signal`**, para incrementar el valor del semáforo desde 0 a 1 (y después desbloquear a **P2**, si está esperando).
- Inmediatamente **antes de** L, **P2** debe invocar a **`sem_wait`**, para esperar a que el valor sea 1 (si no lo era al entrar) y entonces decrementar el valor del semáforo desde 1 a 0.

Pseudo-código de la solución

Llamamos al semáforo **puede_leer**, y entonces el programa con la sincronización correcta mediante este semáforo será este:

```
{ variables compartidas y valores iniciales }
var compartida  : integer ; { var. compartida: P1 escribe y P2 lee }
var puede_leer  : semaphore := 0 ; { 1 si var. pte. de leer, 0 si no }
```

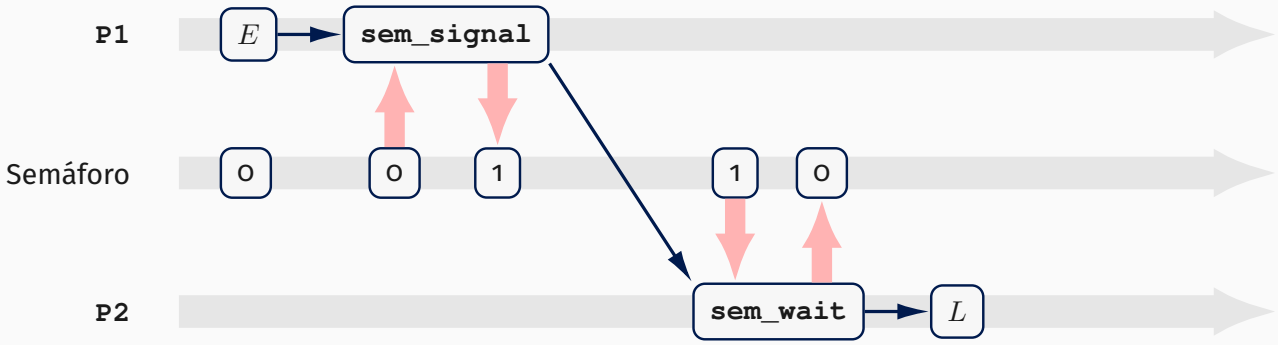
```
process P1 ;
   var local1 : integer ;
begin
   .....
   local1 := ProducirValor() ;
   compartida := local1 ; { E }
   sem_signal( puede_leer ) ;
   .....
end
```

```
process P2 ;
   var local2 : integer ;
begin
   .....
   sem_wait( puede_leer ) ;
   local2 := compartida ; { L }
   UsarValor( local2 );
   .....
end
```

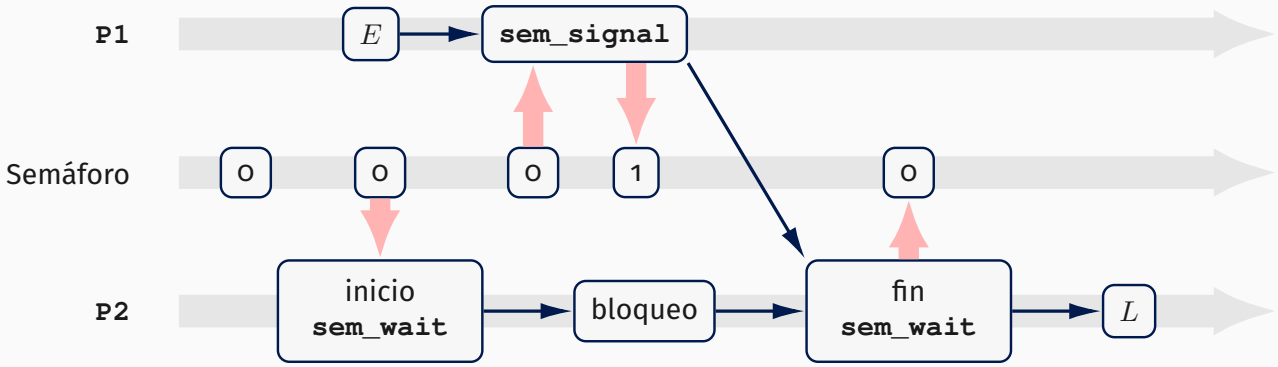
El programa introduce una espera que asegura que L se ejecuta después que E. Es decir **se cumple la condición de sincronización**

Posibles trazas: esquema

Si **P1** inicia **sem_signal** antes de que **P2** inicie **sem_wait**:



Si **P1** inicia **sem_signal** después de que **P2** inicie **sem_wait**:



Posibles trazas: explicación

Solo hay dos posibles interfoliaciones de las sentencias relevantes:

Si **P1** inicia **`sem_signal`** **antes** de que **P2** inicie **`sem_wait`**:

1. **P1** ejecuta **`sem_signal`** y el semáforo queda a 1 (**P1** ya ha ejecutado E antes).
2. **P2** ejecuta **`sem_wait`**, ve el semáforo a 1, lo baja a 0 y termina el **`sem_wait`**.
3. **P2** ejecuta L

Si **P1** inicia **`sem_signal`** **después** de que **P2** inicie **`sem_wait`**:

1. **P2** inicia **`sem_wait`**, ve el semáforo a 0 y se bloquea.
2. **P1** ejecuta **`sem_signal`** y el semáforo queda a 1 (**P1** ya ha ejecutado E antes).
3. **P2** se desbloquea, pone el semáforo a 0 y termina **`sem_wait`**.
4. **P2** ejecuta L.

Subsección 4.3. Exclusión mutua.

El problema de la exclusión mutua

En este caso, queremos que en cada instante de tiempo **solo haya un proceso como mucho ejecutando un trozo de código**, que llamamos *sección crítica*. Es un código que se traduce en dos o más instrucciones atómicas, a la primera de ellas la llamamos I y a la última F

```
process ProcesosEM[ i : 0..n-1 ] ;
begin
   while true do begin
      { inicio de sección crítica (sentencia I) }
      .......
      { fin de sección crítica (sentencia F) }
      ....... { RS: resto de sentencias  }
   end
end
```

Las sentencias atómicas I y F sirven para expresar la condición de sincronización, como veremos a continuación.

Condición de sincronización

La **condición de sincronización** es la siguiente: que en cada instante **solo haya un proceso como mucho que haya ejecutado I pero no el correspondiente F**:

- La única interfoliación permitida es $I, F, I, F, I, F, \ldots$ (en una traza en la cual ignoramos todas las instrucciones distintas de I y de F, que son las relevantes).
- Para asegurarlo hay que introducir una espera bloqueada de los procesos, inmediatamente previa al inicio de la S.C. (inmediatamente previa a I), cuando en ese instante ya haya un proceso en la S.C. (es decir, que ya haya ejecutado I pero no el correspondiente F).
- Por tanto, usaremos un semáforo que nos asegure esto. La espera se implementa con una llamada a **`sem_wait`**, y además usamos una llamada a **`sem_signal`** para desbloquear a los procesos cuando corresponda.

Solución con un semáforo

El valor del semáforo puede ser únicamente 0 o 1. En concreto, será:

- 0 cuando haya un proceso en la S.C. (es decir, cuando haya un proceso que ha ejecutado I pero no el correspondiente F).
- 1 cuando no haya ninguno.

Por tanto:

- El valor inicial del semáforo debe ser 1, ya que al inicio no hay ningún proceso que esté ejecutando la sección crítica.
- Inmediatamente **antes de** I, cada proceso debe invocar a `sem_wait`, para esperar a que el valor sea 1 (si no lo era al entrar) y entonces decrementar el valor del semáforo desde 1 a 0.
- Inmediatamente **después de** F, cada proceso debe invocar a `sem_signal`, para incrementar el valor del semáforo desde 0 a 1 (y después desbloquear a otro proceso, si hay alguno esperando).

Pseudo-código con un semáforo

Llamamos al semáforo **sc_libre**, y entonces el programa con la sincronización correcta mediante este semáforo será este:

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { 1 si S.C. libre, 0 si S.C. ocupada }

process ProcesosEM[ i : 0..n-1 ] ;
begin
   while true do begin
      sem_wait( sc_libre );   { esperar hasta que "sc_libre" sea 1 }
      { aquí va la sección crítica: I .... F }
      sem_signal( sc_libre ); { desbloquear o poner "sc_libre" a 1 }
      { resto de sentencias: .......}
   end
end
```

Este programa introduce una espera que asegura que en cada instante solo haya un proceso como mucho ejecutando la sección crítica. Es decir **se cumple la condición de sincronización**.

Subsección 4.4.
Productor-Consumidor (lectura/escritura repetidas).

El problema del productor-consumidor

El problema del Productor-Consumidor es similar al problema de la lectura y escritura únicas, pero repetidas en un bucle.

```
{ variables compartidas }
var x : integer ; { contiene cada valor producido }
```

```
Process Productor ; { calcula "x" }
   var a : integer ;
begin
   while true begin
      a := ProducirValor() ;
      x := a ; { escritura (E) }
   end
end
```

```
Process Consumidor ; { lee "x" }
   var b : integer ;
begin
   while true do begin
      b := x ; { lectura (L) }
      UsarValor(b) ;
   end
end
```

Este programa es claramente incorrecto, ya que no se evita que un valor escrito no llegue a ser leído, o bien que un valor escrito sea leído más de una vez.

Condición de sincronización

La **condición de sincronización** que queremos imponer consiste en sincronizar los procesos de forma que: **cada valor escrito por el productor sea leído exactamente una vez por el consumidor**. Por tanto:

- No se pueden permitir dos lecturas seguidas sin una escritura intermedia.
- No se pueden permitir dos escrituras seguidas sin una lectura intermedia.
- La primera operación debe ser escritura (no se puede leer un valor antes de que se haya escrito).
- Esto significa que: se permite una interfoliación de las sentencias E y L de la forma $E, L, E, L, E, L, \ldots$. Cualquier otra no se permite.

Uso de semáforos para este problema (1/3)

Para lograr la sincronización requerida, usaremos dos semáforos:

- Un semáforo para que el consumidor espere, antes de L, a que se haya producido una escritura previa. Evita que L se ejecute dos veces seguidas.
- Un semáforo para que el productor espere, antes de E, a que se haya producido una lectura previa (excepto la primera vez). Evita que E se ejecute dos veces seguidas.

Esto hace necesario

- que el productor haga **`sem_wait`** de uno de los semáforos inmediatamente antes de E,
- que el consumidor haga **`sem_wait`** del otro semáforo inmediatamente antes de L, y
- que se hagan los **`sem_signal`** correspondientes y que los valores iniciales sean correctos.

Uso de semáforos para este problema (2/3)

El semáforo donde espera el productor se llamará **`puede_escribir`**:

- Vale 1 cuando aún no ha habido ninguna escritura, o después de que un valor ya se haya leído, pero antes de que se escriba el siguiente valor.
- Vale 0 en otro caso.
- Inicialmente vale 1, pues al inicio no hay ningún valor pendiente de leer.

El semáforo donde espera el consumidor se llamará **`puede_leer`**:

- Vale 1 después de que se haya escrito un valor, pero antes de que se haga la lectura de dicho valor.
- Vale 0 en otro caso.
- Inicialmente vale 0, pues al inicio no se ha escrito ningún valor.

Uso de semáforos para este problema (3/3)

Con estos semáforos, el productor debe:

- Hacer `sem_wait( puede_escribir )` antes de escribir un valor, para esperar si todavía no se ha leído el anterior valor.
- Hacer `sem_signal( puede_leer )` después de escribir un valor, para desbloquear al consumidor (si estaba esperando)

Por otro lado, el consumidor debe:

- Hacer `sem_wait( puede_leer )` antes de leer un valor, para esperar si todavía no se ha escrito un valor nuevo.
- Hacer `sem_signal( puede_escribir )` después de leer un valor, para desbloquear al productor (si estaba esperando)

Uso de semáforos para sincronización

Con todo lo dicho, el programa con la sincronización correcta mediante estos semáforos será este:

```
{ variables compartidas }
var
 x              : integer ;         { contiene cada valor producido }
 puede_leer     : semaphore := 0 ; { 1 si se puede leer x, 0 si no }
 puede_escribir : semaphore := 1 ; { 1 si se puede escribir x, 0 si no }
```

```
Process Productor ; { calcula "x" }
   var a : integer ;
begin
   while true begin
      a := ProducirValor() ;
      sem_wait( puede_escribir  );
      x := a ; { escritura (E) }
      sem_signal( puede_leer ) ;
   end
end
```

```
Process Consumidor ; { lee "x" }
   var b : integer ;
begin
   while true do begin
      sem_wait( puede_leer ) ;
      b := x ; { lectura (L) }
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

Sección 5.
Semáforos en C++11.

5.1. Introducción: operaciones y compilación
5.2. Implementación del ejemplo productor/consumidor

Subsección 5.1.
Introducción: operaciones y compilación.

Tipo de datos y operaciones

El estándar C++11 no contempla funcionalidad alguna para semáforos. Sin embargo, se ha diseñado un tipo de datos (una clase) que ofrezca dicha funcionalidad, usando características de C++11:

- El tipo se denomina **`Semaphore`**.
- Las únicas operaciones posibles sobre las variables del tipo son:
  - Inicialización, obligatoriamente en la declaración:

```
Semaphore s1 = 34, s2 = 0 ;
Semaphore s3(34), s4(5) ;
```

  - Funciones (o métodos) **`sem_wait`** y **`sem_signal`**:

```
sem_wait( s1 );      s1.sem_wait();
sem_signal( s1 );    s1.sem_signal();
```

- Cuando hay varias hebras esperando, **`sem_signal`** despierta a la primera de ellas que entró al **`sem_wait`** (es FIFO).

Variables y arrays de tipo semáforo

Una variable semáforo no se puede copiar sobre otra, ya que eso haría falso el invariante de la variable destino y además carece de sentido copiar la lista de hebras esperando:

```
Semaphore s1 = 0, s2 = 34, s3 = Semaphore(34) ; // ok
s1 = s2 ;              // error: es ilegal copiar un semáforo sobre otro
Semaphore s5 = s1 ; // error: no se puede copiar ni siquiera para crear un nuevo.
```

En C++11 se pueden declarar arrays de semáforos, de tamaño fijo:

```
const int N = 4 ; Semaphore s[N] = { 1, 56, 78, 0 } ; // ok
Semaphore t[2] = { 0, 2 } ; // ok
Semaphore u[3] = { 4, 5 } ; // error: falta un valor
```

Si el tamaño no es conocido al compilar o es muy grande, se puede usar un vector STL (añadiéndole entradas antes de lanzar hebras):

```
std::vector<Semaphore> s ; // se crea sin ningún semáforo (vacío)
...
for( i = 0 ; i < N ; i++ )
   s.push_back( Semaphore(0) ); // añadir entrada nueva
```

Estructura de los programas con semáforos

Los programas que usan semáforos típicamente declaran variables globales de tipo **Semaphore** compartidas entre las hebras, que los usan.

- Es necesario hacer **#include** y **using** en la cabecera:

```
#include <iostream>
#include <thread>
#include "scd.h" // incluye tipo scd::Semaphore
using namespace std ; // permite acortar la notación (abc en lugar de std::abc)
using namespace scd ; // permite 'Semaphore' en lugar de scd::Semaphore
```

- Se debe de disponer de los archivos **scd.h** y **scd.cpp** en el directorio de trabajo.
- Se debe de compilar y enlazar el archivo **scd.cpp**, junto con los fuentes que usan los semáforos (p.ej. **f1.cpp**):

```
g++ -std=c++11 -pthread -I. -o ejecutable_exe f1.cpp scd.cpp
```

Subsección 5.2.
Implementación del ejemplo productor/consumidor.

Cabecera del programa

Usando el tipo **Semaphore**, implementaremos el ejemplo del productor/consumidor (con una única hebra productora y una única consumidora) que ya hemos visto en pseudo-código (archivo `ejemplo13-s.cpp`)

```
#include <iostream>
#include <thread>
#include "scd.h"       // incluye tipo Semaphore

using namespace std ; // permite acortar la notación (abc en lugar de std::abc)
using namespace scd ; // permite usar Semaphore en lugar de scd::Semaphore

// constantes y variables enteras (compartidas)
const int num_iter          = 100 ; // número de iteraciones
int       valor_compartido,         // variable compartida entre prod. y cons.
          contador          = 0 ;   // contador usado en ProducirValor

// semáforos compartidos
Semaphore puede_escribir = 1 ,      // 1 si no hay valor pendiente de leer
          puede_leer     = 0 ;      // 1 si hay valor pendiente de leer

.....
```

Funciones para producir y consumir valores

Las funciones **producir_valor** y **consumir_valor** se usan para simular la acción de generar valores y de consumirlos:

```
....

// función que, cada vez que se invoca, devuelve el siguiente entero:
int producir_valor()
{
  contador++ ;  // incrementar el contador
  cout << "producido: " << contador << endl ;
  return contador ;
}

// función que simula la consumición de un valor (simplemente lo imprime)
void consumir_valor( int valor )
{
  cout << "                    consumido: " << valor << endl  ;
}

.....
```

Hebra productora

La función que ejecuta la hebra productora usa los dos semáforos compartidos para escribir en la variable compartida

```
.....
// función que ejecuta la hebra productora (escribe la variable)
// (escribe los valores desde 1 hasta num_iters, ambos incluidos)
void funcion_hebra_productora(  )
{
   for( unsigned long i = 0 ; i < num_iter ; i++ )
   {
      int valor_producido = producir_valor(); // generar valor
      sem_wait( puede_escribir ) ;
      valor_compartido = valor_producido ; // escribe el valor
      cout << "escrito: " << valor_producido << endl ;
      sem_signal( puede_leer ) ;
   }
}
.....
```

Hebra consumidora

La función que ejecuta la hebra consumidora usa los mismos dos semáforos compartidos para leer de la variable compartida

```
.....
// función que ejecuta la hebra consumidora (lee la variable)
void funcion_hebra_consumidora(  )
{
   for( unsigned long i = 0 ; i < num_iter ; i++ )
   {
      sem_wait( puede_leer ) ;
      int valor_leido = valor_compartido ; // lee el valor generado
      cout << "                    leído: " << valor_leido << endl  ;
      sem_signal( puede_escribir ) ;
      consumir_valor( valor_leido );
   }
}
.....
```

Hebra principal

Como es habitual, la hebra principal, en main, pone en marcha y espera a las otras:

```
.....
// hebra principal (pone las otras dos en marcha)
int main()
{
  // crear y poner en marcha las dos hebras
  thread hebra_productora( funcion_hebra_productora ),
         hebra_consumidora( funcion_hebra_consumidora );

  // esperar a que terminen todas las hebras
  hebra_productora.join();
  hebra_consumidora.join();
}
```

Fin de la presentación.

2.2. Seminario 2

UNIVERSIDAD DE GRANADA

Sistemas Concurrentes y Distribuidos: Seminario 2. Introducción a los monitores en C++11.

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2024-25 (archivo generado el 15 de octubre de 2024)

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

Seminario 2. Introducción a los monitores en C++11.

Índice.

Introducción

El objetivo básico de este seminario es introducir las herramientas que usaremos en C++11 para implementar monitores con semántica *Señalar y Espera Urgente*, junto con un ejemplo sencillo. En concreto:

- Hacemos un breve repaso de los monitores con semántica **Señalar y Espera Urgente** (en adelante SU)
- Vemos como el mecanismo de las clases puede usarse para implementar monitores, en general.
- Introducimos la clase **`HoareMonitor`**, que sirve para implementar monitores SU en C++11.
- Vemos cómo se puede adaptar el ejemplo del productor y consumidor para monitores SU con la citada clase.

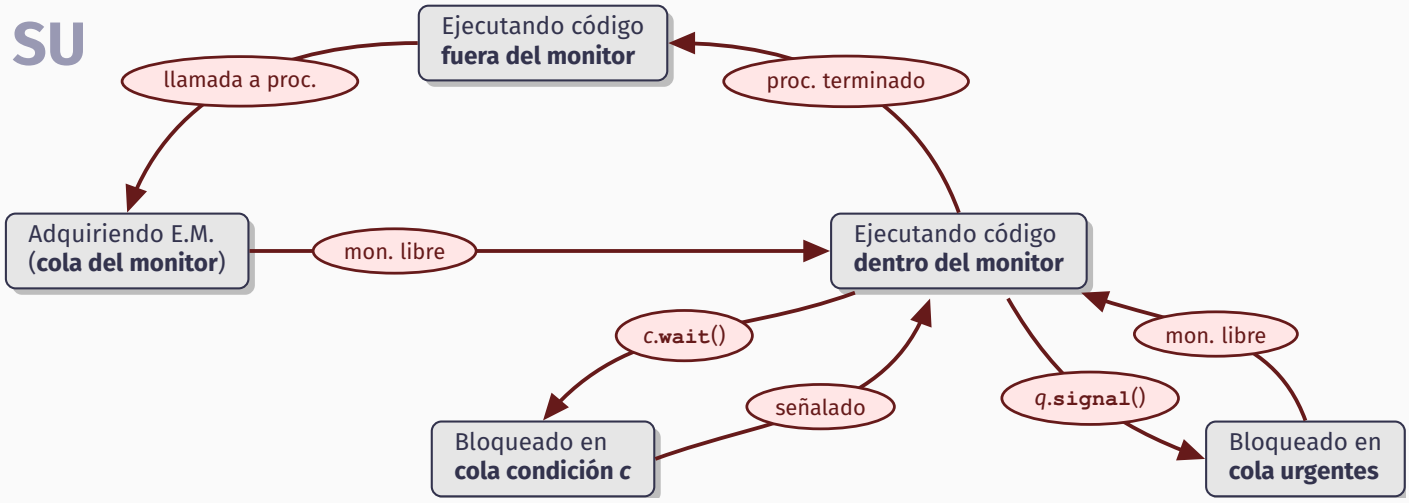
Sección 1.
La semántica SU (monitores Hoare).

La semántica *Señalar y Espera Urgente*. Características

El término **Señalar y Espera Urgente** (SU en adelante) hace referencia a una de las estrategias (semánticas) de implementación de monitores. En ella, si un proceso *señalador* hace **`signal`** sobre un proceso *señalado* (que estaba bloqueado en un **`wait`**):

- El señalador se bloquea justo al ejecutar la operación **`signal`** en una cola de procesos que esperan para acceder al monitor, que llamamos *cola de procesos urgentes*.
- El señalado sale del **`wait`** y entra de forma inmediata en el monitor, ejecutando las sentencias posteriores a **`wait`**. Por tanto, encuentra el monitor exactamente en el mismo estado en que lo dejo el señalador al hacer **`signal`**.
- Los procesos en la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los procesos que esperan en la cola del monitor.

Semántica SU: diagrama de estados de un proceso



- **Señalador:** se bloquea en la **cola de urgentes** hasta readquirir la E.M. y ejecutar código del monitor tras **`signal`**. Para readquirir E.M. tiene más prioridad que los procesos en la cola del monitor.
- **Señalado:** reanuda inmediatamente la ejecución de código del monitor tras **`wait`**.

Ventajas de la semántica SU

La semántica SU presenta diversas ventajas frente a las demás:

- Es posible incluir sentencias después de **`signal`** (el proceso señalador no abandona el monitor tras **`signal`**).
- Las sentencias tras **`signal`** se ejecutan con prioridad frente las hebras de la cola del monitor (el proceso señalador no tiene que esperar en dicha cola otra vez para ejecutar lo que siga a **`signal`**).
- Si un estado del monitor permite desbloquear una hebra con **`signal`**, esa hebra señalada se desbloquea en ese mismo estado y puede avanzar inmediatamente. Se evita que la señalada tenga que volver a comprobar el estado, en bucle.

Sección 2.
Monitores en C++11..

2.1. Los monitores como clases C++
2.2. Encapsulamiento.

Subsección 2.1.
Los monitores como clases C++.

Clases y monitores

El mecanismo de definición de clases de C++ es la forma más natural de implementar el concepto de monitor en este lenguaje:

- Los procedimientos exportados son **métodos publicos**. Son los únicos que se pueden invocar desde fuera.
- Las variables permanentes del monitor se implementan como **variables de instancia no públicas**.
- La inicialización ocurre en los **métodos constructores** de la clase.

Los mecanismos de concurrencia de C++11 permiten:

- Asegurar la exclusión mutua.
- Implementar las variables condición.

Implementación de monitores en C++11

El lenguaje C++11 y la librería estándar no incluye la posibilidad de definir monitores directamente (no hay una clase para eso).

- En C++11, para implementar monitores se pueden usar diversas clases o tipos *nativos* (proporcionados por el lenguaje) como son: *objetos mutex* (tipo `mutex`), *guardas de cerrojo* (tipo `lock_guard`) y *variables condición* (tipo `condition_variable`).
- La implementación puede tener una semántica tipo
  - *Señalar y Continuar* (SC), usando esos tipos nativos directamente.
  - *Señalar y Espera Urgente* (SU, o monitores tipo *Hoare*), de forma indirecta mediante una biblioteca de funciones o clases, las cuales a su vez usan los tipos nativos.

Subsección 2.2. Encapsulamiento..

Ejemplo sencillo

Partimos de un diseño de un monitor sencillo para garantizar EM en los accesos a una variable permanente entera:

```
monitor MContador1 ;                { declaración del nombre del monitor}
 var cont : integer;                { variable permanente (no accesible) }
 export incrementa, leer_valor;     { nombres de métodos que podemos llamar}

 procedure incrementa( );           { procedimiento: incrementa valor actual}
 begin
   cont := cont+1 ;                 {    (añade 1 al valor actual) }
 end;
 function leer_valor() : integer;   { función: devuelve el valor: }
 begin
   return cont ;                    {    el resultado es el valor actual  }
 end;
begin                               { código de inicialización:  }
  cont := 0 ;                       {    pone la variable a cero }
end
```

En este ejemplo, el código del monitor se ejecuta en exclusión mutua. No hay variables condición.

Clase `Mcontador1` para un monitor: declaración

En el archivo `monitor_em.cpp` vemos la declaración de la clase:

```
class MContador1 // nombre de la clase: MContador1
{
  private:        // elementos privados (usables internamente):
    int cont ;    // variable de instancia (contador)
  public:         // elementos públicos (usables externamente):
    MContador1( int valor_ini ); // declaración del constructor
    void incrementa();           // método que incrementa el valor actual
    int  leer_valor() ;          // método que devuelve el valor actual
} ;
MContador1::MContador1( int valor_ini )
{
  cont = valor_ini ;
}
void MContador1::incrementa()
{
   cont ++ ;
}
int MContador1::leer_valor()
{
  return cont ;
}
```

Clase `Mcontador1` para un monitor: uso

El monitor se usa por dos hebras concurrentes (en **`test_1`**)

```
const int num_incrementos = 10000; // número de incrementos por hebra

void funcion_hebra_M1( MContador1 & monitor ) // recibe referencia al monitor
{
   for( int i = 0 ; i < num_incrementos ; i++ )
      monitor.incrementa();
}
void test_1( ) // (se invoca desde main)
{
  // declarar instancia del monitor (inicialmente, cont==0)
  MContador1 monitor(0) ;
  // lanzar las hebras
  thread hebra1( funcion_hebra_M1, ref(monitor) ),
         hebra2( funcion_hebra_M1, ref(monitor) );
  // esperar que terminen las hebras
  hebra1.join();
  hebra2.join();
  // imprimir el valor esperado y el obtenido:
  cout<< "Valor obtenido: " << monitor.leer_valor() << endl // valor final
      << "Valor esperado: " << 2*num_incrementos << endl ;  // valor o.k.
}
```

Ausencia de exclusión mutua

Al ejecutar el programa anterior, vemos que el valor obtenido no es igual al esperado, ya que:

- Las dos hebras acceden concurrentemente a la variable compartida.
- Cada una puede sobreescribir incrementos realizados por la otra (el valor obtenido es menor que el esperado, generalmente).

Para solucionar el problema, usaremos una biblioteca que incluye una clase base para monitores:

- Los métodos se ejecutan en exclusión mutua (el valor final será el esperado).
- Se pueden declarar variables condición.
- Se implementa la semántica *Señalar y Espera Urgente* (SU).

Sección 3.
La clase `HoareMonitor` para monitores SU.

La clase `HoareMonitor` para monitores SU en C++11

La clase `HoareMonitor`

- Se implementa usando las características nativas de C++11.
- Es una clase base de la cual derivamos (con herencia) otras clases para implementar diseños concretos de monitores.
- Los métodos de la clase se ejecutan en exclusión mutua, de forma prácticamente transparente al programador.
- Se proporciona una clase para las variables condición (`CondVar`), con los métodos para señalar y para esperar.
- Garantiza que siempre habrá orden FIFO en la cola del monitor, la de urgentes, y en las colas de las variables condición.

Uso de la clase `HoareMonitor`

Para definir un monitor tipo Hoare (SU) con esta clase, debemos:

- Hacer **`#include`** del archivo **`scd.h`**
- Definir la clase del monitor como derivada (tipo **`public`**) de **`HoareMonitor`**.
- Declarar los procedimientos exportados y el constructor como públicos (el resto de elementos son privados)
- Al inicio del programa, se debe crear una instancia del monitor con la función **`Create`**.
- Guardamos una referencia o puntero al monitor en una variable de tipo **`MRef`** (por *Monitor* ***Ref****erence*).
- Hacer que las hebras usen esa referencia para invocar los métodos del monitor.

Subsección 3.1. Exclusión mutua.

Declaración de la clase MContador2

La clase **MContador2** es similar, pero derivada de **HoareMonitor**

```
class MContador2 : public HoareMonitor
{
  private:       // elementos privados (usables internamente):
    int cont;    // variable de instancia (contador)
  public:        // elementos públicos (usables externamente):
    MContador2( int valor_ini ); // declaración del constructor
    void incrementa();           // método que incrementa el valor actual
    int  leer_valor() ;          // método que devuelve el valor actual
} ;
MContador2::MContador2( int valor_ini )
{
  cont = valor_ini ;
}
void MContador2::incrementa()
{
   cont ++ ;
}
int MContador2::leer_valor()
{
  return cont ;
}
```

Exclusión mutua implícita mediante referencias MRef

La exclusión mutua en los métodos está asegurada implícitamente. Si M es una clase derivada de **HoareMonitor**, entonces:

- Se debe usar la función **Create**<M> para crear una instancia del monitor (de tipo **MRef**<M>), debemos de pasar como argumentos los requeridos por el constructor de M:

```
MRef<M> monitor = Create<M>( ... );
```

- Las funciones que ejecutan las hebras reciben como parámetro un objeto `r` de tipo **MRef**<M>, y usan el operador `->` para invocar los métodos del monitor, con `r->metodo(...)`.
- **Nunca** debemos usar directamente un objeto `m` de tipo M mediante `m.metodo(...)`, ni tampoco un puntero `p` de tipo M`*`, mediante `p->metodo(...)` (lo métodos no se ejecutarían en EM).

Creación y uso de una instancia de MContador2

Código que usa **MContador2** (en **test_2**):

```
void funcion_hebra_M2( MRef<MContador2>  monitor )
{
   for( int i = 0 ; i < num_incrementos ; i++ )
      monitor->incrementa();
}
void test_2()
{
   // declarar instancia del monitor (inicialmente, cont==0)
   MRef<MContador2> monitor = Create<MContador2>( 0 );
   // lanzar las hebras
   thread hebra1( funcion_hebra_M2, monitor ),
          hebra2( funcion_hebra_M2, monitor );
   // esperar que terminen las hebras
   hebra1.join();
   hebra2.join();
   // imprimir el valor esperado y el obtenido:
     cout<< "Valor obtenido: " << monitor->leer_valor() << endl // valor fi-
nal
         << "Valor esperado: " << 2*num_incrementos << endl ;  // valor o.k.
}
```

Actividad

Realiza estas actividades:

- Compila y ejecuta `monitor_em.cpp`. Verás que ejecuta las funciones `test_1` y`test_2`, cada una de ellas usa cada uno de los dos monitores descritos.
- Verifica que el valor obtenido es distinto del esperado en el caso del monitor sin exclusión mutua. Verifica que en el otro monitor (con EM), el valor obtenido coincide con el esperado.
- Añade código para conocer los tiempos que tardan cada par de hebras desde que se lanzan hasta que terminan, sin contar la creación del monitor ni los `cout`. Justifica los resultados que obtienes.

Subsección 3.2. Variables condición.

El tipo `CondVar` para variables condición

En las clases derivadas de **`HoareMonitor`** se puede usar el tipo **`CondVar`** para las *variables condición*

- Deben ser variables de instancia privadas de la clase monitor (nunca se pueden usar fuera de los métodos de dicha clase).
- En el constructor del monitor, **cada una** de esas variables debe crearse explícitamente usando la función **`newCondVar`** (deben inicializarse después de que ya exista el objeto monitor, ya que guardan un puntero al mismo).
- Si alguna variable de este tipo se declara pero no se inicializa en el constructor, usarla es un error que aborta el programa.
- En estas colas condición el orden de salida es siempre FIFO (si hay más de una hebra bloqueda, la primera en salir será la primera que entró).

Operaciones sobre variables condición

Las variables condición (de tipo **`CondVar`**) tienen definidas estas operaciones (métodos), que se invocan desde los métodos del monitor:

- **`wait`**`()`: la hebra que invoca espera bloqueada hasta que otra hebra haga **`signal`** sobre la cola.
- **`signal`**`()`: si la cola está vacía, no se hace nada, en otro caso se libera una hebra de las que esperan, y la hebra que invoca se bloquea en cola de urgentes. La hebra liberada **será siempre la primera que llamó** a **`wait`** en esa cola (si hay más de una esperando).
- **`get_nwt`**`()`: devuelve el número de hebras esperando en la cola.
- **`empty`**`()`: devuelve **`true`** si no hay hebras esperando, o **`false`** si hay al menos una.

Sección 4.
Productor/Consumidor únicos en monitores SU.

El productor/consumidor en monitores

En estas sub-sección veremos cómo diseñar e implementar una solución al problema del productor/consumidor (con una hebra en cada rol), usando un monitor SU. Para ello nos basamos en la solución que vimos con semáforos en la práctica 1:

- Las funciones para producir un dato y consumir un dato son exactamente iguales que en la versión de semáforos.
- Diseñamos un monitor SU que encapsula el buffer y define métodos de acceso.
- La función que ejecuta la hebra de productor invocan el método del monitor **`insertar`** para añadir un nuevo valor en el buffer.
- La función que ejecuta la hebra consumidora invoca el método (función) del monitor **`extraer`** para leer un valor del buffer y eliminarlo del mismo.
- El tamaño o capacidad del buffer es un valor constante conocido, que llamamos k.

Hebras productora y consumidora

La hebra productora produce un total de m items, los mismos que consume la consumidora. El valor m es una constante cualquiera, superior al tamaño del buffer ($k < m$).

El pseudo-código de los procesos es como sigue:

```
Monitor ProdConsSU1
  .......
end
```

```
Process Productor ;
  var dato : integer ;
begin
  for i := 1 to m do begin
    dato := ProducirDato();
    ProdConsSU1.insertar( dato );
  end
end
```

```
Process Consumidor ;
  var dato : integer ;
begin
  for i := 1 to m do begin
    dato := ProdConsSU1.extraer();
    ConsumirDato( dato );
  end
end
```

Diseño del monitor: condiciones de espera

Llamamos n al número de entradas del buffer ocupadas con algún valor pendiente de leer. El diseño del monitor debe de asegurar que:

- La hebra productora espera (en **`insertar`**) hasta que hay al menos un hueco para insertar el valor (es decir, espera hasta que $n < k$)
- La hebra consumidora espera (en **`extraer`**) hasta que hay al menos una celda ocupada con un valor pendiente de leer (es decir, hasta que $0 < n$).

Como consecuencia, en el monitor debemos incluir:

- Una variable permanente que contenga el valor de n
- Una cola condición llamada **`libres`**, cuya condición asociada es $n < k$, y donde espera la hebra productora cuando $n = k$.
- Otra, llamada **`ocupadas`**, cuya condición asociada es $0 < n$, y donde espera la hebra consumidora cuando $n = 0$.

Diseño del monitor: accesos al buffer

Los accesos al buffer se pueden hacer usando el esquema LIFO o el FIFO que ya vimos para el diseño con semáforos. Hay que tener en cuenta que:

- Si se usa la opción LIFO, basta con tener la variable permanente **`primera_libre`**, cuyo valor coincide con n. Esta variable sirve tanto para acceder al buffer como para comprobar las condiciones de espera.
- Si se usa la opción FIFO, son necesarias las variables **`primera_libre`** y **`primera_ocupada`** para acceder al buffer. Estas dos variables no permiten saber cuántas celdas hay ocupadas, por tanto necesitamos una variable adicional con el valor de n.

Pseudocódigo del monitor

Por todo lo dicho, el monitor SU (opcion LIFO) puede tener, por tanto, este diseño:

```
Monitor ProdConsSU1

var
  { array con los datos insertados pendientes extraer }
  buffer : array[ 0..k-1 ] of integer ;

  { variables permanentes para acceso y control de ocupación }
  primera_libre   : integer := 0;  { celda de siguiente inserción (== n) }

  { colas condición }
  libres          : Condition ;    { cola de espera hasta n < k (prod.) }
  ocupadas        : Condition ;    { cola de espera hasta n > 0 (cons.) }

  { procedimientos exportados del monitor }
  procedure insertar( dato : integer ) begin ..... end
  function  extraer ( ) : integer      begin .... end

end
```

Operaciones de insertar y extraer

Su diseño es de esta forma:

```
procedure insertar( dato : integer )
begin
  if primera_libre == k then          { si buffer lleno (n == k) }
    libres.wait() ;                   {  esperar que haya celdas libres }
  buffer[primera_libre]:= dato ;      { escribir dato en buffer }
  primera_libre:=  primera_libre+1 ;  { incrementa n (ahora n > 0) }
  ocupadas.signal();                  { despertar consum., si esperaba }
end
```

```
function extraer( ) : integer
   var result ;
begin
  if primera_libre == 0 then          { si buffer vacío (n == 0) }
    ocupadas.wait() ;                 {  esperar que haya celdas ocupadas }
  primera_libre := primera_libre-1 ;  { decrementa n (ahora n < k) }
  result := buffer[primera_libre] ;   { leer del buffer (antes de signal) }
  libres.signal();                    { despertar produ., si esperaba }
  return result ;                     { devolver valor del buffer }
end
```

Implementación en C++11. Actividad.

En el archivo `prodcons1_su.cpp` puedes encontrar una implementación del monitor **`ProdConsSU1`** descrito, con semántica SU (versión LIFO).

- Esta implementación, al finalizar, comprueba que cada valor entero producido está entre 0 y $m-1$, ambos incluidos, y además que es producido y consumido una y solo una sola vez. Si esto no ocurre, se produce un mensaje de error al final.
- Modifica la implementación para usar la opción FIFO. Verifica que funciona correctamente.

Fin de la presentación.

2.3. Seminario 3

UNIVERSIDAD DE GRANADA

Sistemas Concurrentes y Distribuidos: Seminario 3. Introducción a paso de mensajes con MPI.

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2024-25 (archivo generado el 17 de junio de 2024)

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

Seminario 3. Introducción a paso de mensajes con MPI.
Índice.

Introducción

- El objetivo de esta práctica es familiarizar al alumno con el uso de la interfaz de paso de mensajes MPI y la implementación OpenMPI de esta interfaz.
- Se indicarán los pasos necesarios para compilar y ejecutar programas usando OpenMPI.
- Se presentarán las principales características de MPI y algunas funciones básicas de comunicación entre procesos.

Enlaces para acceder a información complementaria

- Web oficial de OpenMPI.
- Instalación de OpenMPI en Linux.
- Ayuda para las funciones de MPI.
- Tutorial de MPI.

Sección 1.
Message Passing Interface (MPI).

¿Qué es MPI?

MPI es un estándar que define una API para programación paralela mediante paso de mensajes, que permite crear programas portables y eficientes.

- Proporciona un conjunto de funciones que pueden ser utilizadas en programas escritos en C, C++, Fortran y Ada.
- MPI-2 contiene más de 150 funciones para paso de mensajes y operaciones complementarias (con numerosos parámetros y variantes).
- Muchos programas paralelos se pueden construir usando un conjunto reducido de dichas funciones (hay 6 funciones básicas).

Modelo de Programación en MPI

El esquema de funcionamiento implica un número fijo de procesos que se comunican mediante llamadas a funciones de envío y recepción de mensajes.

- El modelo básico es SPMD (*Single Program Multiple Data*): todos los procesos ejecutan un mismo programa.
- Permite el modelo MPMD (*Multiple Program Multiple Data*): cada proceso puede ejecutar un programa diferente.
- La creación e inicialización de procesos no está definida en el estándar, depende de la implementación. En OpenMPI sería:

```
mpirun -oversubscribe -np 4 -machinefile maquinas prog1_mpi_exe
```

  - Comienza 4 copias del ejecutable `prog1_mpi_exe`.
  - El archivo `maquinas` define la asignación de procesos a ordenadores del sistema distribuido.

Aspectos de implementación

- Hay que hacer: `#include <mpi.h>`: define constantes, tipos de datos y los prototipos de las funciones MPI.
- Las funciones devuelven un código de error.
  - `MPI_SUCCESS`: Ejecución correcta.
- `MPI_Status` es un tipo estructura con los metadatos de los mensajes:
  - `status.MPI_SOURCE`: proceso fuente.
  - `status.MPI_TAG`: etiqueta del mensaje.
- Constantes para representar tipos de datos básicos de C/C++ (para los mensajes en MPI): `MPI_CHAR`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, etc.
- Comunicador: es tanto un grupos de procesos como un contexto de comunicación. Todas las funciones de comunicación necesitan como argumento un comunicador.

Sección 2.
Compilación y ejecución de programas MPI.

La librería OpenMPI. Instalación en Linux y macOS

OpenMPI (☞ open-mpi.org) es una implementación portable y de código abierto del estándar MPI-2, llevada a cabo por una serie de instituciones de ámbito tanto académico y científico como industrial.

- Es la mejor opción para compilar y ejecutar programas MPI en Linux y macOS (en Windows usamos otra, ver más abajo).
- En **Linux** podemos instalarlo fácilmente usando el gestor de paquetes **apt**, con esta orden:

```
sudo apt install libopenmpi-dev
```

- En **macOS** podemos usar el gestor de paquetes **Homebrew** (☞ brew.sh/index_es), instalamos con:

```
brew install open-mpi
```

- La documentación de OpenMPI puede consultarse aqui:
  ☞ docs.open-mpi.org

Compilación y ejecución de programas con OpenMPI

OpenMPI ofrece varios *scripts* necesarios para trabajar con programas aumentados con llamadas a funciones de MPI. Los más importantes son estos dos:

- `mpicxx`: compila y/o enlazar programas C++ con MPI.
- `mpirun`: ejecuta programas MPI.

Se compila con las opciones habituales, pero usando usando `mpicxx` en lugar de `g++`. Por ejemplo:

```
mpicxx -std=c++11  -c ejemplo.cpp
mpicxx -std=c++11  -o ejemplo_mpi_exe  ejemplo.o
```

También se puede compilar directamente (sin crear `.o`):

```
mpicxx -std=c++11  -o ejemplo_mpi_exe  ejemplo.cpp
```

Ejecución de programas con OpenMPI

La forma más usual de ejecutar un programa MPI con OpenMPI es :

```
mpirun  -oversubscribe  -np 4  ./ejemplo_mpi_exe
```

- El argumento `-np` sirve para indicar cuántos procesos ejecutarán el programa ejemplo. En este caso, se lanzarán cuatro procesos ejecutando `ejemplo_mpi_exe`.
- Como no se indica la opción `-machinefile`, OpenMPI lanzará dichos 4 procesos en el mismo ordenador donde se ejecuta `mpirun`.
- Con la opción `-machinefile`, podríamos realizar asociaciones de procesos a distintos ordenadores.
- La opción `-oversubscribe` puede ser necesaria si el número de procesadores disponibles en algún ordenador es inferior al número de procesos que se quieren lanzar en ese ordenador

Compilación y ejecución con MS-MPI en Windows.

Microsoft proporciona una implementación de MPI-1 para Windows (MS-MPI). Para usarla descargamos e instalamos los archivos `.exe` y `.msi` disponibles aquí:

☞ www.microsoft.com/en-us/download/details.aspx?id=100593

En el *Developer PowerShell* añadimos las carpetas de *includes* y librerías a las correspondientes variables del compilador:

```
$env:INCLUDE += ";$env:MSMPI_INC."
$env:LIB     += ";$env:MSMPI_LIB64."
```

Compilamos los programas igual, pero añadiendo la librería MS-MPI:

```
cl /EHsc /Fe:ejemplo_mpi  ejemplo_mpi.cpp  msmpi.lib
```

Ejecutamos con `mpiexec` indicando el número N de procesos:

```
mpiexec -n N .\ejemplo_mpi
```

Sección 3.
Funciones MPI básicas.

Funciones MPI básicas

Hay 6 funciones básicas en MPI:

- **`MPI_Init`**: inicializa el entorno de ejecución de MPI.
- **`MPI_Finalize`**: finaliza el entorno de ejecución de MPI.
- **`MPI_Comm_size`**: determina el número de procesos de un comunicador.
- **`MPI_Comm_rank`**: determina el identificador del proceso en un comunicador.
- **`MPI_Send`**: operación básica para envío de un mensaje.
- **`MPI_Recv`**: operación básica para recepción de un mensaje.

Inicializar y finalizar un programa MPI

Se usan estas dos sentencias:

```
int MPI_Init( int *argc, char ***argv )
```

- Llamado antes de cualquier otra función MPI.
- Si se llama más de una vez durante la ejecución da un error.
- Los argumentos `argc, argv` son los argumentos de la línea de orden del programa.

```
int MPI_Finalize( )
```

- Llamado al fin de la computación.
- Realiza tareas de limpieza para finalizar el entorno de ejecución

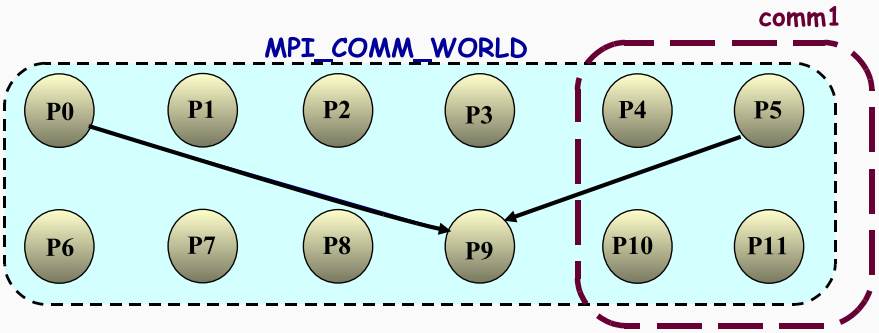
Subsección 3.1.
Introducción a los comunicadores.

Introducción a los comunicadores (1)

Un **Comunicador MPI** es una variable de tipo **`MPI_Comm`**. Está constituido por:

- Grupo de procesos: Subconjunto de procesos (pueden ser todos).
- Contexto de comunicación: Ámbito de paso de mensajes en el que se comunican dichos procesos. Un mensaje enviado en un contexto sólo puede ser recibido en dicho contexto.

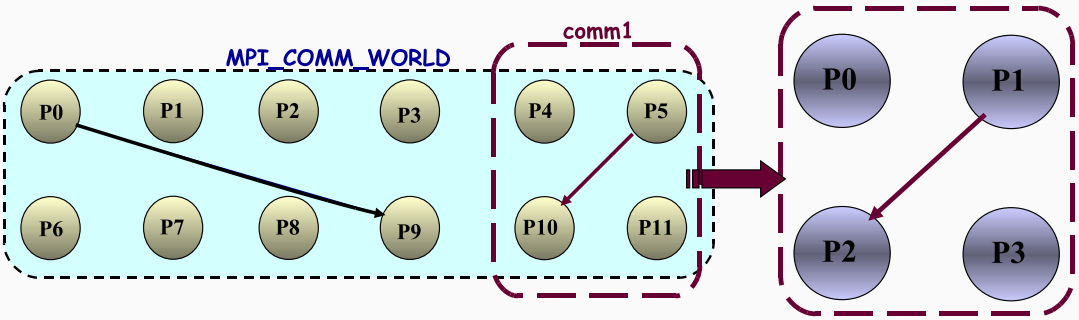
Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.



Introducción a los comunicadores (2)

La constante **MPI_COMM_WORLD** hace referencia al **comunicador universal**, está predefinido e incluye todos los procesos lanzados:

- La identificación de los procesos participantes en un comunicador es unívoca:
- Un proceso puede pertenecer a diferentes comunicadores.
- Cada proceso tiene un identificador: desde 0 a $P-1$ (P es el número de procesos del comunicador).
- Mensajes destinados a diferentes contextos de comunicación no interfieren entre sí.



Consulta del número de procesos

La función **MPI_Comm_size** tiene esta declaración:

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

- Escribe en el entero apuntado por **size** el número total de procesos que forman el comunicador **comm**.
- Si usamos el comunicador universal, podemos saber cuantos procesos en total se han lanzado en una aplicación, por ejemplo:

```
int num_procesos ; // contendrá el total de procesos de la aplic.
MPI_Comm_size( MPI_COMM_WORLD, &num_procesos );
cout << "El numero total de procesos es: " << num_procesos << endl ;
```

Consulta del identificador del proceso

La función **MPI_Comm_rank** está declarada como sigue:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

- Escribe en el entero apuntado por **rank** el número de proceso que llama. Este número es el número de orden dentro del comunicador **comm** (empezando en 0). Ese número se suele llamar *rank* o *identificador* del proceso en el comunicador.
- Se suele usar al inicio de una aplicación MPI, con el comunicador universal, como en este ejemplo:

```
int id_propio ; // contendrá el número de proceso que llama
MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
cout << "mi número de proceso es:"  << id_propio << endl ;
```

Normalmente, usaremos este número de orden en el comunicador universal para identificar cada proceso.

Ejemplo de un programa simple

En el archivo **holamundo.cpp** vemos un ejemplo sencillo:

```
#include <mpi.h>
#include <iostream>
using namespace std;

int main( int argc, char *argv[] )
{
  int id_propio, num_procesos ;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  cout <<"Hola desde proceso " <<id_propio <<" de " <<num_procesos <<endl;
  MPI_Finalize();
  return 0;
}
```

Si se compila y ejecuta con 4 procesos obtenemos esta salida:

```
$mpicxx -std=c++11 -o hola holamundo.cpp
$mpirun -np 4 ./hola
```

```
Hola desde proc. 0 de 4
Hola desde proc. 2 de 4
Hola desde proc. 1 de 4
Hola desde proc. 3 de 4
```

Subsección 3.2.
Funciones básicas de envío y recepción de mensajes.

Envío asíncrono seguro de un mensaje (`MPI_Send`)

Un proceso puede enviar un mensaje usando **`MPI_Send`**:

```
int MPI_Send( void *buf_emi, int num, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
```

- Envía los datos (**`num`** elementos de tipo **`datatype`** almacenados a partir de **`buf_emi`**) al proceso **`dest`** dentro del comunicador **`comm`**.
- El entero **`tag`** (*etiqueta*) se transfiere junto con el mensaje, y suele usarse para clasificar los mensajes en distintas categorías o tipos, en función de sus etiquetas. Es no negativo.
- Implementa envio **asíncrono seguro**: tras acabar **`MPI_Send`**
  - MPI ya ha leído los datos de **`buf_emi`**, y los ha copiado a otro lugar, por tanto podemos volver a escribir sobre **`buf_emi`** (el envío es **seguro**).
  - el receptor no necesariamente ha inciado ya la recepción del mensaje (el envío es **asíncrono**)

Recepción segura síncrona de un mensaje (`MPI_Recv`)

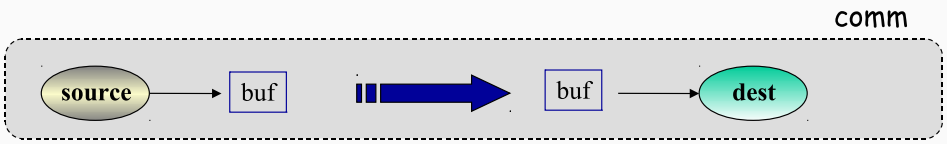
Un proceso puede recibir un mensaje usando **`MPI_Recv`**, que se declara como sigue:

```
int MPI_Recv( void *buf_rec, int num, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
```

- Espera hasta recibir un mensaje del proceso **`source`** dentro del comunicador **`comm`** con la etiqueta **`tag`**, y escribe los datos en posiciones contiguas desde **`buf_rec`**.
- Puesto que se espera a que el emisor envíe, es una recepción **síncrona**. Puesto que al acabar ya se pueden leer en **`buf_rec`** los datos transmitidos, es una recepción **segura**.
- Se pueden dar valores especiales o *comodín*:
  - Si **`source`** es **`MPI_ANY_SOURCE`**, se puede recibir un mensaje de cualquier proceso en el comunicador
  - Si **`tag`** es **`MPI_ANY_TAG`**, se puede recibir un mensaje con cualquier etiqueta.

Envío y recepción de mensajes (2)

Los datos se copian desde **buf_emi** hacia **buf_rec**:



- Los argumentos **num** y **datatype** determinan la longitud en bytes del mensaje. El objeto **status** es una estructura con el emisor (campo **MPI_SOURCE**), la etiqueta (campo **MPI_TAG**).

Para obtener la cuenta de valores recibidos, usamos **status**

```
int MPI_Get_count( MPI_Status *status, MPI_Datatype dtype, int *num );
```

- Escribe en el entero apuntado por **num** el número de items recibidos en una llamada **MPI_Recv** previa. El receptor debe conocer y proporcionar el tipo de los datos (**dtype**).

Ejemplo sencillo (1): estructura del programa

Dos procesos: emisor y receptor (archivo `sendrecv1.cpp`)

```
#include <iostream>
#include <mpi.h>  // incluye declaraciones de funciones, tipos y ctes. MPI
using namespace std;
const int id_emisor             = 0,  // identificador de emisor
          id_receptor           = 1,  // identificador de receptor
          num_procesos_esperado = 2;  // numero de procesos esperados

int main( int argc, char *argv[] )
{
  int id_propio, num_procesos_actual; // identificador propio, núm.procesos
  MPI_Init( &argc, &argv );           // inicializa MPI
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );           // lee mi ident.
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual ); // lee num.procs.
  if ( num_procesos_esperado == num_procesos_actual )    // si n.procs. ok
  {
     ..... // hacer envío o recepción (según id_propio)
  }
  else if ( id_propio == 0 ) // si error, el primer proceso informa
    cerr<< "Esperados 2 procs., hay: "<< num_procesos_actual<< endl;
  MPI_Finalize();  // terminar MPI: debe llamarse siempre por cada proceso.
  return 0;        // terminar proceso
}
```

Ejemplo sencillo (2): envío y recepción

Cuando el número de procesos es correcto, el comportamiento de cada proceso depende de su rol, es decir, de su identificador propio (**id_propio**). En este caso, uno emite (**id_emisor**) y otro recibe (**id_receptor**):

```
if ( id_propio == id_emisor ) // soy emisor: enviar
{
  int valor_enviado = 100 ; // buffer del emisor (tiene 1 entero: MPI_INT)
  MPI_Send( &valor_enviado, 1, MPI_INT, id_receptor, 0, MPI_COMM_WORLD );
  cout << "Emisor ha enviado: " << valor_enviado << endl ;
}
else // soy receptor: recibir
{
  int valor_recibido; // buffer del receptor (tiene 1 entero: MPI_INT)
  MPI_Status estado;  // estado de la recepción
  MPI_Recv( &valor_recibido,1,MPI_INT,id_emisor,0,MPI_COMM_WORLD,&estado);
  cout << "Receptor ha recibido: " << valor_recibido << endl ;
}
```

Emparejamiento de operaciones de envío y recepción

En MPI, una operación de envío (con etiqueta e) realizada por un proceso emisor A **encajará** con una operación de recepción realizada por un proceso receptor B si y solo si se cumplen cada una de estas tres condiciones:

- A nombra a B como receptor y e como etiqueta.
- B especifica `MPI_ANY_SOURCE`, o bien nombra explíctamente a A como emisor
- B especifica `MPI_ANY_TAG`, o bien nombra explícitamente e como etiqueta

Si al iniciar una operación de recepción se determina que encaja con varias operaciones de envío ya iniciadas:

- Se seleccionará de entre esos varios envíos **el primero que se inició** (esto facilita al programador garantizar propiedades de equidad).

Interpretación de bytes transferidos

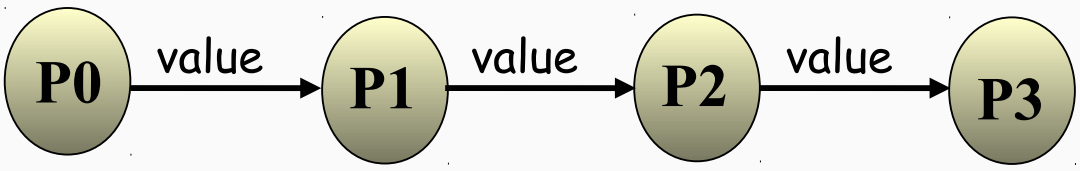
Es importante tener en cuenta que para determinar el emparejamiento MPI **no tiene en cuenta el tipo de datos ni la cuenta de items**. Es responsabilidad del programador asegurarse de que, en el lado del receptor:

- **Los bytes transferidos se interpretan con el mismo tipo de datos** que el emisor usó en el envío (de otra forma los valores leídos son indeterminados).
- **Se sabe exactamente cuantos items de datos** se han recibido (en otro caso el receptor podría leer valores indeterminados de zonas de memoria no escritas por MPI).
- **Se ha reservado memoria suficiente** para recibir todos los datos (de no hacerse, MPI escribiría erróneamente fuera de la memoria correspondiente a la variable especificada en el receptor).

Ejemplo: Difusión de mensaje en una cadena de procesos

En este ejemplo

- Funciona con un número de procesos como mínimo igual a 2.
- Todos los procesos ejecutan un bucle, en cada iteración:
  - El primer proceso (identificador $= 0$) pide un valor entero por teclado.
  - El resto de procesos (identificador > 0), reciben cada uno un valor del proceso anterior.
  - Todos los procesos (excepto el último) envían su valor al siguiente proceso.
- El bucle acaba cuando se ha recibido o leído un valor negativo.



Difusión en cadena: estructura del programa

El ejemplo está en el archivo `sendreceive2.cpp`:

```
const int num_procesos_min = 2 ; // número mínimo de procesos

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_min <= num_procesos_actual ) // núm. procs. ok
  {
     ........ // bucle de envío/recepción
  }
  else if ( id_propio == 0 ) // si error, el primero proceso informa
    cerr << "Número de procesos menor que mínimo ("
         << num_procesos_min << ")" << endl;

  MPI_Finalize();
  return 0;
}
```

Difusión en cadena: bucle de envío/recepción

Si el número de procesos es correcto, cada proceso hace un bucle. El código tiene esta forma:

```
const int  id_anterior  = id_propio-1,  // ident. proceso anterior
           id_siguiente = id_propio+1 ; // ident. proceso siguiente
int        valor;  // valor recibido o leído, y enviado
MPI_Status estado; // estado de la recepción

do
{
  if ( id_anterior < 0 ) // si soy el primer proceso (id_anterior es -1):
    cin >> valor ;       //    pedir valor por teclado
  else                   // si no soy el primero: recibir valor de anterior
    MPI_Recv( &valor, 1,MPI_INT, id_anterior, 0, MPI_COMM_WORLD, &estado);

  cout<< "Proc."<< id_propio<< ": recibido/leído: "<< valor<< endl ;

  if ( id_siguiente < num_procesos_actual ) // so no soy último: enviar
    MPI_Send( &valor, 1, MPI_INT, id_siguiente, 0, MPI_COMM_WORLD );
}
while( valor >= 0 ); // acaba cuando se teclea un valor negativo
```

Sección 4.
Paso de mensajes síncrono en MPI.

Envío en modo síncrono (y seguro) (`MPI_Ssend`)

En MPI existe una función de envío **síncrono** (siempre es **seguro**):

```
int MPI_Ssend( void* buf_emi, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm );
```

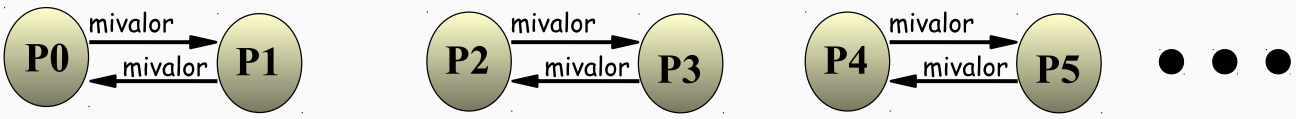
- Inicia un envío, lee datos y espera el inicio de la recepción, con los mismos argumentos que **`MPI_Send`**.
- Es **síncrono y seguro**. Tras acabar **`MPI_Ssend`**
  - ya se ha iniciado en el receptor una operación de recepción que encaja con este envío (es **síncrono**),
  - los datos ya se han leído de **`buf_emi`** y se han copiado a otro lugar. Por tanto se puede volver a escribir en **`buf_emi`** (es **seguro**)
- Si la correspondiente operación de recepción usada es **`MPI_Recv`**, la semántica del paso de mensajes es puramente síncrona (existe una cita entre emisor y receptor).

Ejemplo de intercambio síncrono

En este ejemplo hay un número par de procesos:

- Los procesos se agrupan por parejas. Cada proceso enviará un dato a su correspondiente pareja o vecino.
- Los envíos se hacen usando envío síncrono (con **`MPI_Ssend`**).
- Si todos los procesos hacen envío seguido de recepción (o todos lo hacen al revés), **habría interbloqueo con seguridad**.
- Para evitarlo, los procesos pares hacen envío seguido de recepción y los procesos impares recepción seguida de envío.

Por tanto, el esquema de funcionamiento se puede ver así:



Intercambio síncrono: estructura del programa

La estructura de **main** (en **intercambio_sincrono.cpp**) es esta:

```
#include <iostream>
#include <mpi.h>
using namespace std;

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_actual % 2 == 0 ) // si número de procesos correcto (par)
   {
      .... // envío/recepción hacia/desde proceso vecino
   }
   else if ( id_propio == 0 ) // si n.procs. impar, el primero da error
      cerr << "Error: se esperaba un número par de procesos" << endl ;

   MPI_Finalize();
   return 0;
}
```

Intercambio síncrono: envío/recepción

El envío y recepción con el proceso vecino se puede hacer así:

```
MPI_Status estado ;
int valor_env = id_propio*(id_propio+1),// dato a enviar (cualquiera vale)
    valor_rec,                          // valor recibido
    id_vecino ;                         // identificador de vecino

if ( id_propio % 2 == 0 )  // si proceso par: enviar y recibir
{
   id_vecino = id_propio+1 ; // el vecino es el siguiente
   MPI_Ssend( &valor_env,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD );
   MPI_Recv ( &valor_rec,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD,&estado );
}
else  // si proceso impar: recibir y enviar
{
   id_vecino = id_propio-1 ; // el vecino es el anterior
   MPI_Recv ( &valor_rec,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD,&estado );
   MPI_Ssend( &valor_env,1,MPI_INT,id_vecino,0,MPI_COMM_WORLD );
}
cout << "El proceso " << id_propio << " ha recibido " << valor_rec
     << " del proceso  " << id_vecino << endl ;
```

Sección 5.
Sondeo de mensajes.

Sondeo de mensajes

MPI incorpora dos operaciones que permiten a un proceso receptor averiguar si hay algún mensaje pendiente de recibir (en un comunicador), y en ese caso obtener los metadados de dicho mensaje. Esta consulta:

- no supone la recepción del mensaje.
- se puede restringir a mensajes de un emisor.
- se puede restringir a mensajes con una etiqueta.
- cuando hay mensaje, permite obtener los metadatos: emisor, etiqueta y número de items (el tipo debe ser conocido).

Las dos operaciones son

- **`MPI_Iprobe`**: consultar si hay o no algún mensaje pendiente en este momento.
- **`MPI_Probe`**: esperar bloqueado hasta que haya al menos un mensaje.

Espera bloqueada con `MPI_Probe`

La función **`MPI_Probe`** tiene esta declaración:

```
int MPI_Probe( int source, int tag, MPI_Comm comm,
               MPI_Status *status );
```

El proceso que llama queda bloqueado hasta que haya al menos un mensaje enviado a dicho proceso (en el comunicador **`comm`**) que encaje con los argumentos.

- **`source`** puede ser un identificador de emisor o **`MPI_ANY_SOURCE`**
- **`tag`** puede ser una etiqueta o bien **`MPI_ANY_TAG`**.
- **`status`** permite conocer los metadados del mensaje, igual que se hace tras **`MPI_Recv`**.
- Si hay más de un mensaje disponible, los metadatos se refieren al primero que se envió.

Consulta previa a recepción, con `MPI_Probe`

En el archivo `ejemplo_probe.cpp` vemos un programa en el cual los procesos mandan cadenas de texto a un proceso receptor, que las imprime al recibirlas. El receptor reserva justo la memoria necesaria para cada cadena:

```
int num_chars_rec ; // número de caracteres del mensaje (sin el cero al final)
MPI_Status estado ; // contiene metadatos del mensaje

// esperar mensaje y leer la cuenta de caracteres (sin recibirlo)
MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &estado );
MPI_Get_count( &estado, MPI_CHAR, &num_chars_rec );

// reservar memoria para el mensaje y recibirlo
char * buffer = new char[num_chars_rec+1] ;
MPI_Recv( buffer, num_chars_rec, MPI_CHAR, estado.MPI_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD, &estado );
buffer[num_chars_rec] = 0 ; // añadir un cero al final

// imprimir el mensaje y liberar la memoria que ocupa
cout << buffer << endl ;
delete [] buffer ;
```

Consulta no bloqueante con `MPI_Iprobe`

La función **`MPI_Iprobe`** tiene esta declaración:

```
int MPI_Iprobe( int source, int tag, MPI_Comm comm, int *flag,
                MPI_Status *status );
```

Al terminar, el entero apuntado por **`flag`** será mayor que 0 solo si hay algún mensaje enviado al proceso que llama, y que encaje con los argumentos (en el comunicador **`comm`**). Si no hay mensajes, dicho entero es 0.

- No supone bloqueo alguno.
- La consulta se refiere a los mensajes pendientes en el momento de la llamada.
- Los parámetros (excepto **`flag`**) se interpretan igual que en **`MPI_Probe`**.

Recepción con prioridad usando `MPI_Iprobe`

Aquí (`ejemplo_iprobe.cpp`) un proceso receptor determina si hay mensajes pendientes de recibir de los emisores con idents. desde `id_min` hasta `id_max`, ambos incluidos.

Si hay más de uno, recibe del emisor con identificador más bajo. Si no hay mensajes pendientes, queda a la espera hasta recibir el primero de cualquier emisor:

```
MPI_Status estado ; int hay_mens, id_emisor, valor ;

// comprobar si hay mensajes, en orden creciente de los posibles emisores
for( id_emisor = id_min ; id_emisor <= id_max ; id_emisor++ )
{
  MPI_Iprobe( id_emisor, MPI_ANY_TAG, MPI_COMM_WORLD, &hay_mens, &estado);
  if ( hay_mens ) break ; // si hay mensaje: terminar consulta
}
if ( ! hay_mens )                 // si no hay mensaje:
   id_emisor = MPI_ANY_SOURCE ;   // aceptar de cualquiera

// recibir el mensaje del emisor concreto o de cualquiera
MPI_Recv( &valor, 1,MPI_INT, id_emisor, 0, MPI_COMM_WORLD, &estado );
```

Sección 6.
Comunicación insegura.

Operaciones inseguras.

MPI ofrece la posibilidad de usar **operaciones inseguras** (asíncronas). Permiten el inicio de una operación de envío o recepción, y después el emisor o el receptor puede continuar su ejecución de forma concurrente con la transmisión:

- **`MPI_Isend`**: inicia envío pero retorna antes de leer el buffer.
- **`MPI_Irecv`**: inicia recepción pero retorna antes de recibir.

En algún momento posterior se puede comprobar si la operación ha terminado o no, se puede hacer de dos formas:

- **`MPI_Wait`**: espera bloqueado hasta que acabe el envío o recepción.
- **`MPI_Test`**: comprueba si el envio o recepción ha finalizado o no. no supone espera bloqueante.

Operaciones inseguras

Se pueden usar estas dos funciones:

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request *request );

int MPI_Irecv( void* buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request );
```

Los argumentos son similares a **MPI_Send**, excepto:

- **request** es un ticket que permitirá después identificar la operación cuyo estado se pretende consultar o se espera que finalice.
- La recepción no incluye argumento **status** (se obtiene con las operaciones de consulta de estado de la operación).

Cuando ya no se va a usar una variable **MPI_Request**, se puede liberar la memoria que usa con **MPI_Request_free**, declarada así:

```
int  MPI_Request_free( MPI_Request *request )
```

Consulta de estado de operaciones inseguras

La función **MPI_Test** comprueba la operación identificada por un ticket (**request**) y escribe en **flag** un número > 0 si ha acabado, o bien 0 si no ha acabado:

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )
```

MPI_Wait sirve para esperar bloqueado hasta que termine una operación:

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

En ambas funciones, una vez terminada la operación referenciada por el ticket:

- podemos usar el objeto **status** para consultar los metadatos del mensaje.
- la memoria usada por **request** es liberada por MPI (no hay que llamar a **MPI_Request_free**).

Ventajas y seguridad en operaciones inseguras

Las operaciones inseguras:

- Permiten simultanear trabajo útil en el emisor y/o receptor con la lectura, transmisión y recepción del mensaje.
- Aumentan el paralelismo potencial y por tanto pueden mejorar la eficiencia en tiempo.

Las operaciones de consulta de estado (`MPI_Wait` y `MPI_Test`) permiten saber cuando **es seguro** volver a usar el buffer de envío o recepción, ya que nos dicen que la operación ha acabado cuando

- se han leído y copiado los datos del buffer del emisor (si el ticket se refiere a una operación `MPI_Isend`).
- se han recibido los datos en el buffer del receptor (si el ticket se refiere a una operación `MPI_Irecv`).

Una operación insegura se puede emparejar con una operación segura y/o síncrona.

Intercambio de mensajes con operaciones inseguras

A modo de ejemplo, vemos una solución (archivo `intercambio_nobloq.cpp`) con operaciones inseguras que evita el interbloqueo asociado al intercambio síncrono:

```
int         valor_enviado = id_propio*(id_propio+1), // dato a enviar
            valor_recibido, id_vecino ;
MPI_Status  estado ;
MPI_Request ticket_envio, ticket_recepcion;

if ( id_propio % 2 == 0 ) id_vecino = id_propio+1 ;
else                      id_vecino = id_propio-1 ;

// iniciar ambas operaciones simultáneamente (el orden es irrelevante)
MPI_Irecv( &valor_recibido, 1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
           &ticket_recepcion );
MPI_Isend( &valor_enviado,  1, MPI_INT, id_vecino, 0, MPI_COMM_WORLD,
           &ticket_envio );

// esperar hasta que acaben ambas operaciones
MPI_Wait( &ticket_envio, &estado );
MPI_Wait( &ticket_recepcion, &estado );
```

Fin de la presentación.

Todos los archivos ya sean ejecutables o en cualquier otro formato necesarios para la realización de las prácticas se encuentran en:

- Materiales de la Asignatura SCD

3 Examenes

3.1. Primer Parcial / Simulacro

3.1.1. Enunciado

SCD. Parcial de Práctica hasta la Práctica 2

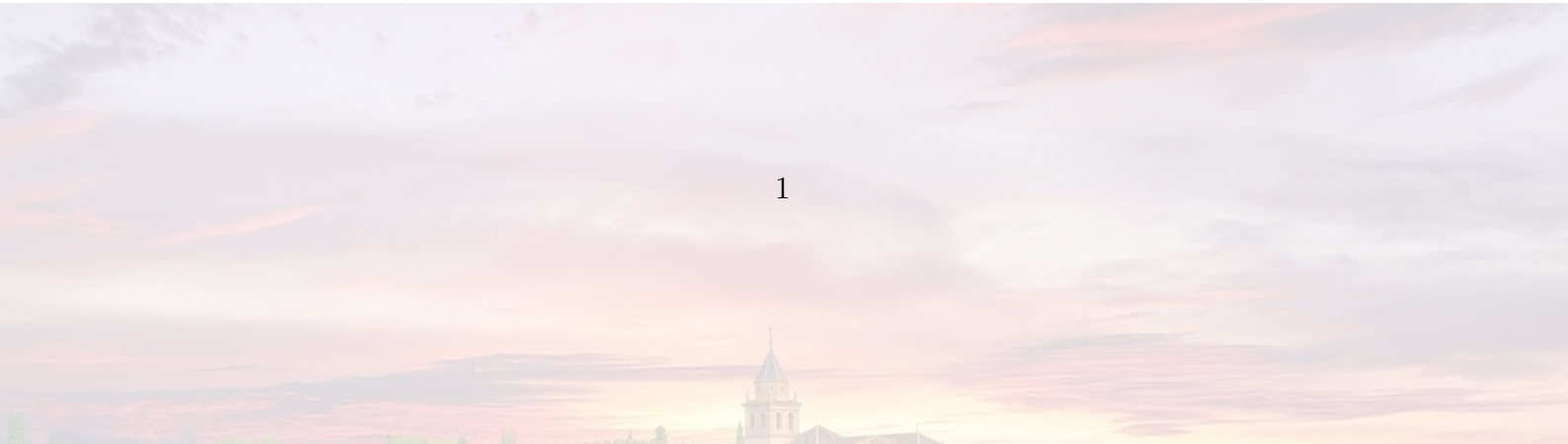
Ismael Sallami Moreno

- **Asignatura:** SCD
- **Grado:** Doble Grado en Ingeniería Informática + ADE .
- **Tiempo:** 90 minutos.
- **Contenidos evaluables:** Semáforos y Monitores.

---

1. Preguntas

- Realizar el mismo juego del simulacro de los corazones, pero hay ciertas diferencias a implementar:
  - Hay entre 2-8 jugadores.
  - Hay que tener en cuenta la dirección que adopte el jugador.
  - El corazón se coloca en una dirección.
  - Direcciones posibles: izq, der, arriba y abajo.
  - Varios jugadores pueden mirar hacia la misma dirección, el primero que llega gana 5 puntos, el 2º gana 4 puntos y el 3º 2 puntos, el resto 1 punto.
  - Como máximo 10 rondas



3.1.2. Solución

Solucion y Material Extra

3.1.3. Primer Parcial Solucion Detallada

Solucion Detallada

4 Referencias

- Diapositivas de clase.