



UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos:

Seminario 1. Programación multihebra y semáforos.

Carlos Ureña / Jose M. Mantas / Pedro Villar
2021-22

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

Seminario 1. Programación multihebra y semáforos.

Índice.

1. Concepto e Implementaciones de Hebras
2. Hebras en C++11
3. Sincronización básica en C++11
4. Introducción a los Semáforos
5. Semáforos en C++11

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:

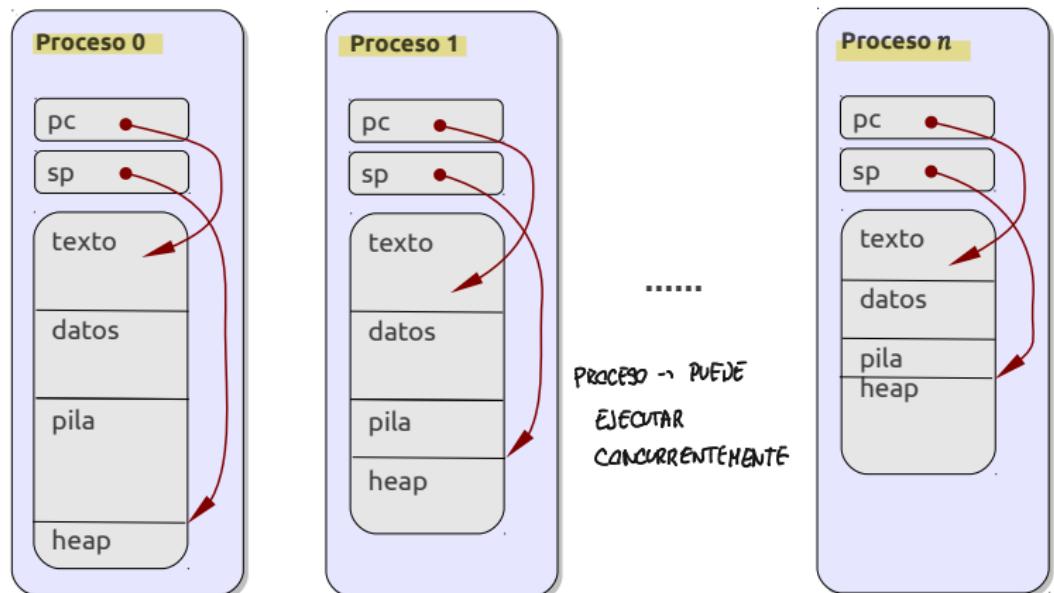
- ▶ **texto**: zona (tamaño fijo) con las instrucciones del programa
- ▶ **datos**: espacio (de tamaño fijo) para variables globales.
- ▶ **pila**: espacio (de tamaño cambiante) para variables locales.
- ▶ **mem. dinámica (*heap*)**: espacio ocupado por variables dinámicas.

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:

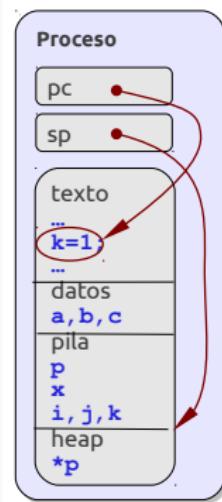


Ejemplo de un proceso

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

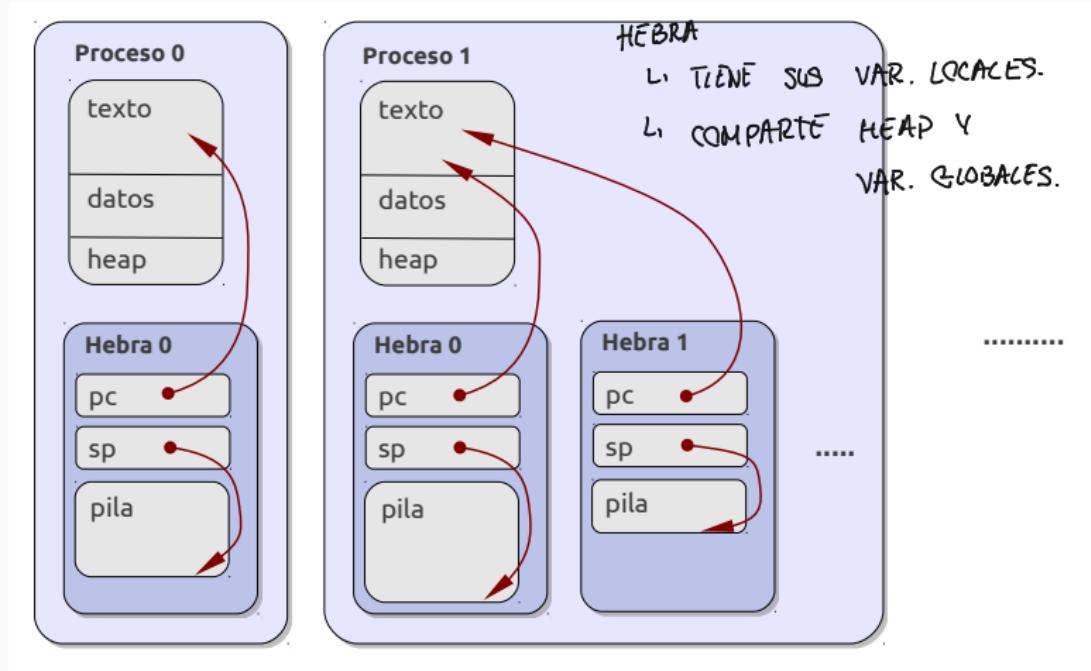
para mayor eficiencia en esta situación se diseñó el concepto de

hebra: ~~HEBRA → FLUJO DE CONTROL EN EL TEXTO (COMÚN) DEL PROCESO AL QUE PERTENECE~~

- ▶ Un proceso puede contener una o varias hebras.
- ▶ Una hebra es un flujo de control en el texto (común) del proceso al que pertenecen.
- ▶ 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

main es una hebra

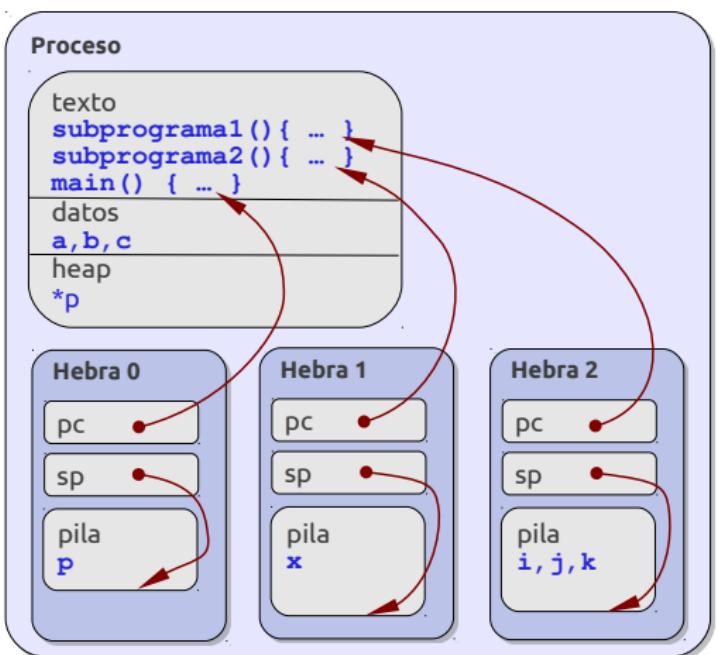
Al inicio de un programa, existe una única hebra (**que ejecuta la función main en C/C++**). Durante la ejecución del programa:

- ▶ 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. Creación y finalización de hebras
- 2.3. Sincronización mediante unión
- 2.4. Paso de parámetros y obtención de un resultado
- 2.5. Vectores de hebras y futuros
- 2.6. Medición de tiempos
- 2.7. 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 dos revisiones posteriores de C++ (2014, 2017), 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 2021-22.

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* y/o *variables condición*.
- ▶ 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).

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.2.

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 ocurre en el ejemplo visto).

Compilar con la línea de órdenes

Si queremos compilar y enlazar con **g++** en la línea de órdenes un fuente (compuesto posiblemente de varios archivos .cpp, llamados $f_1.cpp$, $f_2.cpp \dots f_n.cpp$) debemos de escribir:

```
g++ -std=c++11 -pthread -o ejecutable  $f_1.cpp$   $f_2.cpp \dots f_n.cpp$ 
```

- ▶ Se puede usar el compilador **clang++** en lugar de **g++** (en macOS, **clang++** es el compilador del sistema de desarrollo XCode de Apple).

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  $f_1.cpp$ 
g++ -std=c++11 -pthread -c  $f_2.cpp$ 
.....
g++ -std=c++11 -pthread -c  $f_n.cpp$ 
g++ -std=c++11 -pthread -o ejecutable_exe  $f_1.o$   $f_2.o \dots f_n.o$ 
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.3.

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:

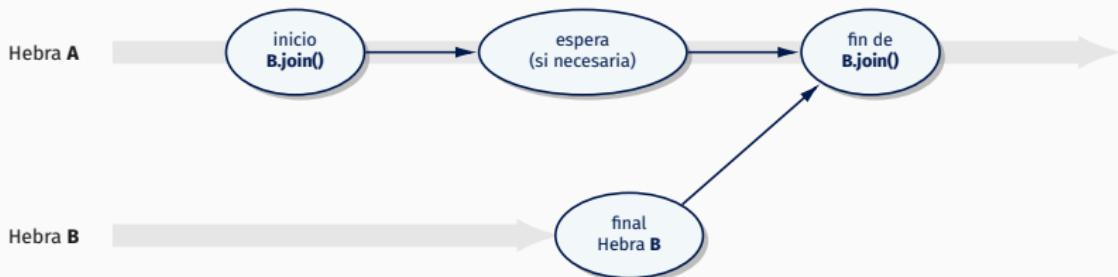
- ▶ Usando la operación de unión (*join*). -> *mecanismo de unión*.
- ▶ Usando *mutex* o *variables condición*

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.

El método **joinable** devuelve un valor lógico que indica si una hebra está activa (devuelve **true**) o no lo está (**false**).

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.4.

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:

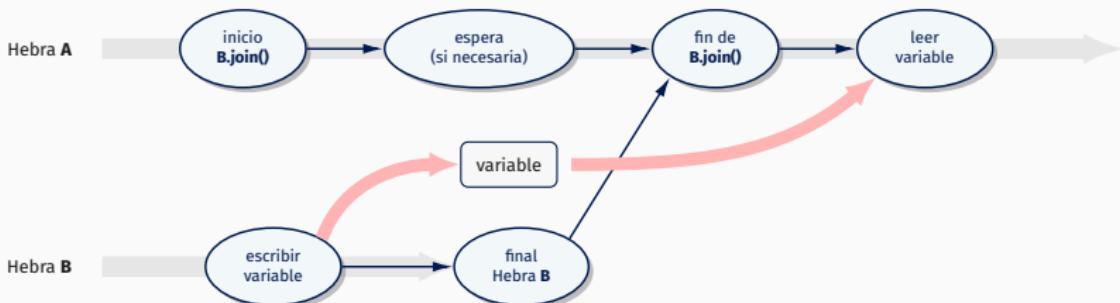
QUIERO QUE MI HEBRA A EXPULSE UN VALOR Y LA HECHA B OTRA HEBRA QUE LO USA EN F

- ▶ 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-

ESTA
ES UNA
QUE
USAMOS

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>      // declaración 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 resultados:  
    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 resultados:  
    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 ;  
}
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.5.

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 código.

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 ;  
}
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.6.

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 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_precision_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 ;
}
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.7.

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:

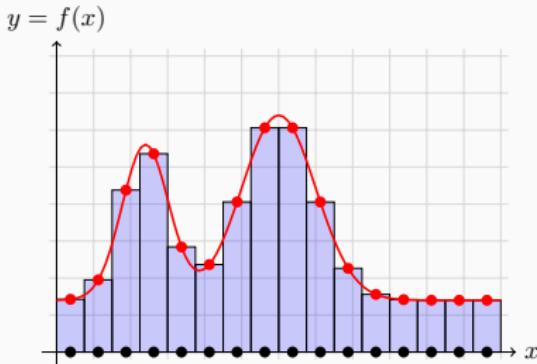
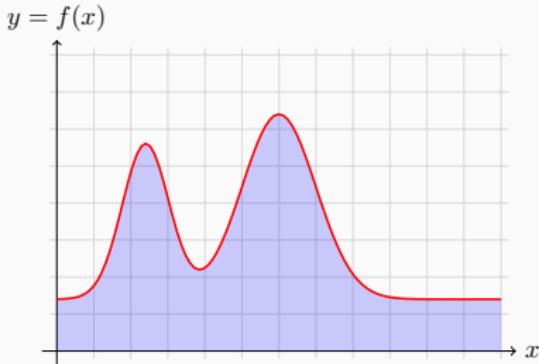
$$I = \int_0^1 f(x) dx$$

- ▶ 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:

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

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 :

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

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 xj
    suma += f( xj ); // añadir f(xj) a suma
  }
  return suma/m ; // devolver valor promedio de f
}
```

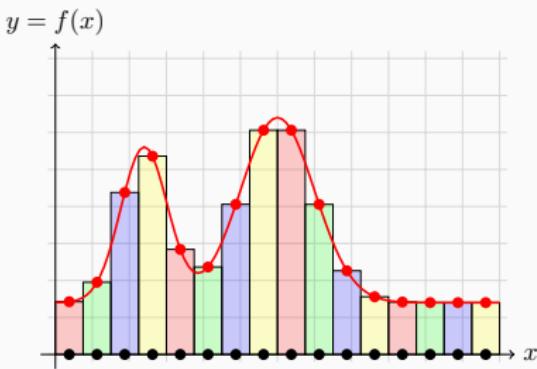
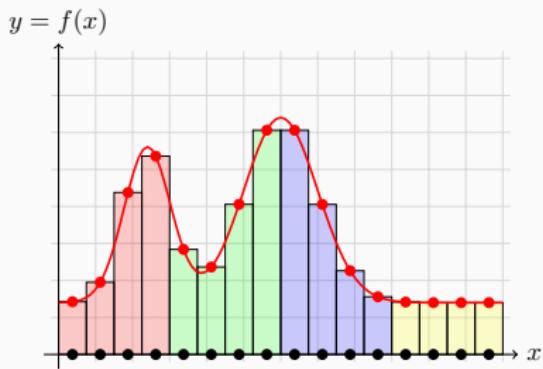
Versión concurrente de la integración

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 $i \bmod n$ (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  $\pi$  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.

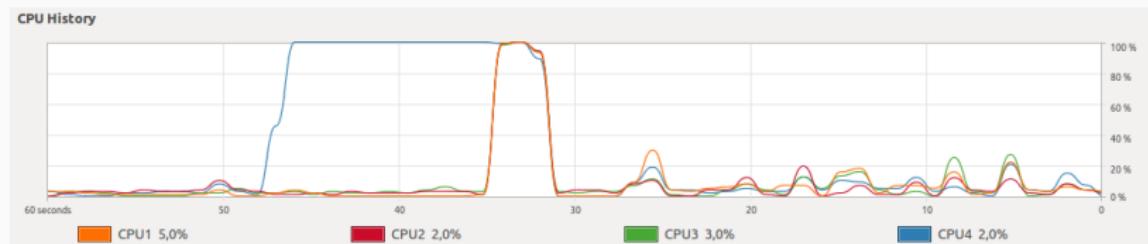
Como actividad se propone copiar la plantilla en `ejemplo09.cpp` y completar en este archivo la implementación del cálculo concurrente del número π , tal y como hemos visto aquí:

- ▶ 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:

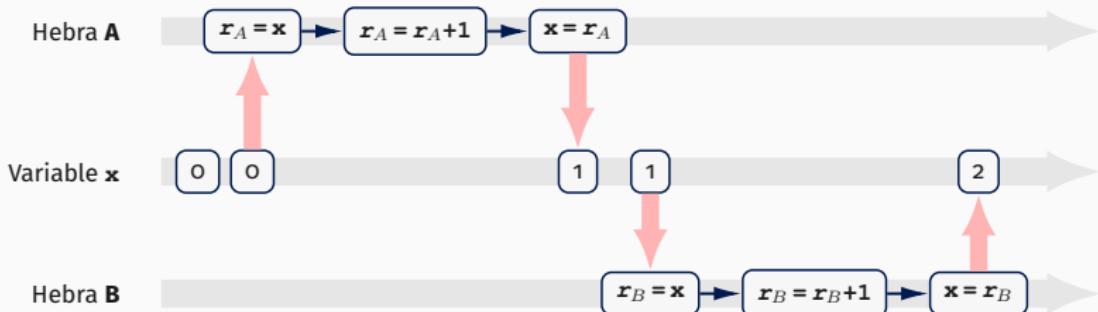
- ▶ **Incrementar o decrementar una variable entera** o flotante se hace en varias instrucciones atómicas distintas. En particular, puede que dos incrementos simultáneos de una variable entera dejen la variable con una unidad más en lugar de dos unidades más, como cabe esperar.
- ▶ **Insertar o eliminar 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)

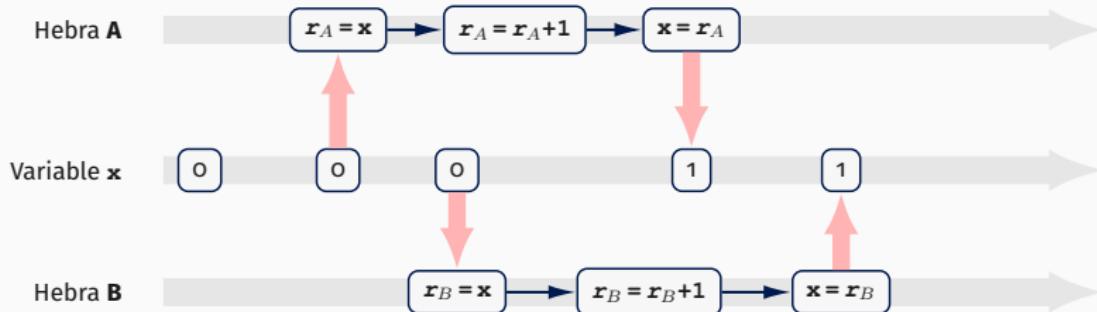
A modo de ejemplo, supongamos que dos hebras A y B ejecutan concurrentemente la sentencia $x++$ sobre una variable compartida (global) x , que está inicializada a 0. Cada hebra usa su propio registro (r_A y r_B), y hace el incremento mediante tres instrucciones atómicas (lectura + incremento + escritura)

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.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 3. Sincronización básica en C++11

Subsección 3.1.

Tipos de datos atómicos.

Tipos atómicos en C++11

Para cada tipo entero posible (**char**, **int**, **long**, **unsigned**, etc...) existe un correspondiente *tipo atómico*, llamado **atomic**<T> o bien **atomic_T** (donde T es el tipo entero original).

- ▶ Para los enteros, las operaciones se suelen implementar con instrucciones hardware atómicas específicas del juego de instrucciones del procesador
- ▶ Si k es una de estas variables, las siguientes operaciones se hacen de forma atómica:
 - ▶ Asignación de un valor: `k=expresion;`.
 - ▶ Incrementos o decrementos de la forma: `k++;` `k--;`
`k+=expresion;` `k-=expresion;`

(si la expresión no es un simple literal, su evaluación previa no ocurre atómicamente junto con la actualización).

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 2021-22.

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 pantalla 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
}
```

SC ↴

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.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 4. Introducción a los Semáforos

Subsección 4.1.

Estructura, operaciones y propiedades.

Estructura de un semáforo

Un semáforo es una 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

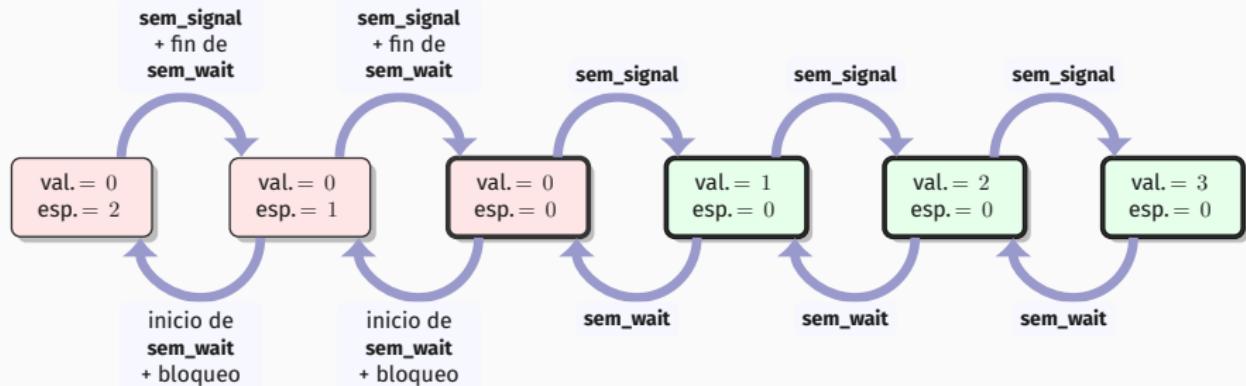
Los semáforos cumplen estas propiedades:

- ▶ 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 $\text{esp.} = 0$ 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 2021-22.

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).
- ▶ Por tanto, la condición que queremos cumplir es $\#L \leq \#E$, es decir, en cualquier estado:

$$0 \leq \#E - \#L$$

En adelante, para cualquier sentencia S de un programa concurrente y para cualquier estado durante la ejecución del programa, llamamos $\#S$ al número de veces que se ha completado la ejecución de S , por cualquier proceso, desde el inicio hasta llegar a ese estado.

Solución con un semáforo

Usamos un semáforo, cuyo valor será $\#E - \#L$, que es el valor que queremos mantener no negativo al lo largo del tiempo. Para que el semáforo tenga ese valor, se deben usar estas operaciones:

- ▶ Inicializar el semáforo a 0 (ya que en el estado inicial la expresión $\#E - \#L$ vale 0, pues $\#E$ y $\#L$ valen ambas 0).
- ▶ Inmediatamente después de E , incrementamos el valor del semáforo con **sem_signal**, ya que $\#E$ aparece con signo positivo en la expresión $\#E - \#L$.
- ▶ Inmediatamente antes de L , decrementamos el valor del semáforo, con **sem_wait**, ya que $\#L$ aparece con signo negativo en la expresión $\#E - \#L$. Se debe hacer **antes** de L para que el **sem_wait** espere si es necesario y así evitar que la expresión $\#E - \#L$ tome un valor negativo.

Pseudo-código de la solución

Aquí vemos la solución. El semáforo valdrá:

- ▶ 1 si **P1** ha ejecutado *E*, pero **P2** no ha comenzado *L*. Es decir, cuando la variable tiene un valor pendiente de leer.
- ▶ 0 en cualquier otro caso (antes de terminar *E* y después de terminar *L*).

```
{ 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
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 4. Introducción a los Semáforos

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**.

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

Las sentencias (ficticias) *I* y *F* constituyen el inicio y final de la sección crítica, y se introducen exclusivamente para expresar la condición de sincronización.

Condición de sincronización

Queremos que solo haya un proceso como mucho que haya terminado de ejecutar I pero que no haya terminado el correspondiente F :

- ▶ La única interfoliación permitida es I, F, I, F, I, F, \dots
- ▶ El número de procesos en sección crítica es $\#I - \#F$. Este valor solo puede ser 0 o 1, luego se debe cumplir:

$$0 \leq \#I - \#F \leq 1$$

- ▶ Debido a que cada proceso solo ejecuta F una vez después de cada I , tenemos garantizado que siempre $\#F \leq \#I$, o lo que es lo mismo: $0 \leq \#I - \#F$
- ▶ Así, únicamente hay que asegurar que $\#I - \#F \leq 1$, es decir:

$$0 \leq 1 + \#F - \#I$$

(equivale a que no se ejecute dos veces I de forma consecutiva sin $\#F$ entre ellas)

Solución con un semáforo

Para solucionar el problema, usamos un semáforo cuyo valor es $1 + \#F - \#I$. Este valor es el número de procesos que pueden iniciar la sección crítica, y solo puede valer 0 o 1 (ya que se cumplen las dos condiciones anteriores).

```
{ 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 }
        { inicio de sección crítica (sentencia I) }
        { aquí va la sección crítica: ..... }
        { fin de sección crítica (sentencia F) }
        sem_signal( sc_libre ); { desbloquear o poner "sc_libre" a 1 }
        { resto sección: .....}
    end
end
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 4. Introducción a los Semáforos

Subsección 4.4.

Productor-Consumidor (lectura/escritura repetidas).

Uso de semáforos para sincronización

El problema del Productor-Consumidor es similar al problema de la lectura/escritura, 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
```

Sincronización en el ejemplo productor-consumidor

En el ejemplo anterior del productor-consumidor:

- ▶ Se permite una interfoliación de las sentencias E y L de la forma E, L, E, L, E, L, \dots . Cualquier otra no se permite.
- ▶ El número de valores escritos en la variable compartida y pendientes de leer solo puede ser 0 o 1, y ese número coincide con $\#E - \#L$. Por tanto se debe cumplir

$$0 \leq \#E - \#L \leq 1$$

- ▶ Lo anterior equivale a dos condiciones independientes que se deben de cumplir, son las siguientes:

$$0 \leq 1 + \#L - \#E \quad \text{and} \quad 0 \leq \#E - \#L$$

2 SEMAFOROS

Uso de semáforos para sincronización

El problema se soluciona con dos semáforos:

- ▶ Semáforo **puede_escribir**: vale $1 + \#L - \#E$
- ▶ Semáforo **puede_leer**: vale $\#E - \#L$.

```
{ 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

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 5. Semáforos en C++11

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
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 1. Programación multihebra y semáforos.

Sección 5. Semáforos en C++11

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.



UNIVERSIDAD
DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar
2021-22

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

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

Índice.

1. Encapsulamiento y exclusión mutua
2. Monitores nativos tipo *Señalar* y *Continuar* (SC)
3. Monitores tipo *Señalar* y *Espera Urgente* (SU)

Introducción

Clases y monitores

monitor -> evolución semáforo.

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 públicos**. 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, implementando la cola del monitor **como una variable de instancia (privada)** de tipo **mutex** o *cerrojo* (lo llamamos *cerrojo del monitor*)
- ▶ Implementar las variables condición usando el tipo **condition_variable**

Estructura del seminario

En este seminario veremos:

- ▶ Un ejemplo de un monitor sencillo (sin variables condición), que ofrece **encapsulamiento y exclusión mutua**.
- ▶ Monitores basados directamente en las características de C++11, que permite colas condición con semántica **señalar y continuar (SC)**.
- ▶ Monitores basados en una biblioteca que permiten usar la semántica **señalar y espera urgente (SU)**

En la práctica 2 veremos ejemplos adicionales de monitores SU.

Sección 1. Encapsulamiento y exclusión mutua.

- 1.1. Encapsulamiento mediante clases C++
- 1.2. Exclusión mutua mediante uso directo de cerrojos
- 1.3. Exclusión mutua mediante guardas de cerrojo

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 1. Encapsulamiento y exclusión mutua

Subsección 1.1.

Encapsulamiento mediante clases C++.

Ejemplo sencillo

Ejemplo de un monitor que permite accesos concurrentes a una variable permanente de tipo entero.

```
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 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 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 puntero 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, &monitor ),
          hebra2( funcion_hebra_M1, &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.
}
```

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, y por tanto, cada una puede sobreescribir incrementos realizados por la otra (el valor obtenido es menor que el esperado por regla general).

- ▶ Para solucionar el problema, usaremos un objeto **mutex** asociado a cada instancia del monitor (una variable de instancia privada de tipo **std::mutex**)
- ▶ A esa variable le llamamos **cerrojo del monitor**.
- ▶ Al inicio de cada método público, adquirimos el cerrojo (con **lock**), y al final lo liberamos (con **unlock**).
- ▶ De esta forma, el código del monitor se ejecuta en exclusión mutua.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 1. Encapsulamiento y exclusión mutua

Subsección 1.2.

Exclusión mutua mediante uso directo de cerrojos.

Declaración de la clase con un cerrojo

Declaramos una clase (**MContador2**) similar a la anterior, pero que ahora incorpora como variable de instancia el cerrojo del monitor (también en el archivo **monitor_em.cpp**):

```
class MContador2 // nombre de la clase: MContador2
{
    private:          // elementos privados (usables internamente):
        int cont ;      // variable de instancia (contador)
        mutex cerrojo_mon ; // cerrojo del monitor (tipo mutex)

    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 ) // el constructor es similar
{
    cont = valor_ini ;
}
```

Métodos con exclusión mutua

En los dos métodos públicos, llamamos a **lock** y **unlock**

```
void MContador2::incrementa()
{
    cerrojo_mon.lock();      // accede a exclusión mutua
    cont ++ ;                // incrementar variable
    cerrojo_mon.unlock();    // liberar exclusión mutua
}

int MContador2::leer_valor()
{
    cerrojo_mon.lock();      // accede a exclusión mutua
    int resultado = cont;    // copiar cont en el resultado
    cerrojo_mon.unlock();    // liberar a exclusión mutua
    return resultado ;       // devolver el resultado
} ;
```

Es necesario usar la variable local **resultado** en **leer_valor**, ya que si se liberara el cerrojo y luego hicieramos **return cont**, se leería **cont** fuera de la exclusión mutua.

Inconvenientes de lock/unlock

El esquema que hemos visto antes presenta varios inconvenientes:

- ▶ En los casos de **return** que devuelven un valor, hay que hacer la copia previa en una variable local.
- ▶ En métodos con varias sentencias **return**, en varios puntos de su código, hay que repetir la llamada a **unlock** antes de cada **return**.
- ▶ Si se produce una excepción durante la ejecución del método (en E.M.), no se liberaría el cerrojo (no se ejecuta **unlock**), dejando al monitor en un estado incorrecto que lleva después a interbloqueo (ninguna hebra está ejecutando código pero ninguna puede entrar y todas esperan).

El librería estándar de C++11 incluye un mecanismo para solventar estas dificultades.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 1. Encapsulamiento y exclusión mutua

Subsección 1.3.

Exclusión mutua mediante guardas de cerrojo.

Guardas de cerrojo

C++11 incluye las denominadas **guardas de cerrojo (lock guards)**:

- ▶ Una guarda de cerrojo es una variable local a cada método exportado del monitor, variable que contiene una referencia al cerrojo del monitor.
- ▶ La guarda se declara al inicio del método, y en la creación (en su constructor) se gana la exclusión mutua (se llama a **lock**)
- ▶ Al ser una variable local, la guarda se destruye automáticamente al final del método (independientemente de si el método se acaba por llegar el flujo de control al final, por **return** o por una excepción).
- ▶ Durante la destrucción de la guarda, se invoca automáticamente **unlock**

De esta forma, el programador no tiene que escribir explícitamente sentencias **unlock**, que se ejecutan siempre antes de acabar el método.

Uso de guardas de cerrojo

Las guardas son variables del tipo `unique_lock<mutex>`. El constructor de esas variables recibe como parámetro una referencia al cerrojo del monitor. La implementación de `Contador3` es similar a `Contador2` excepto que los métodos públicos quedan así:

```
void MContador3::incrementa()
{
    unique_lock<mutex> guarda( cerrojo_mon ); // gana exclusión mutua
    cont ++ ;      // incrementar variable, después liberar exclusión mutua
}
int MContador3::leer_valor()
{
    unique_lock<mutex> guarda( cerrojo_mon ); // gana exclusión mutua
    return cont ;      // devolver valor de x, después liberar exclusión mutua
} ;
```

Tambien se puede usar `lock_guard` en lugar de `unique_lock`, sería totalmente equivalente en este ejemplo, pero usamos `unique_lock` ya que `unique_lock` es compatible con las variables condición, pero `lock_guard` no.

Actividad

Realiza estas actividades:

- ▶ Compila y ejecuta `monitor_em.cpp`. Veras que ejecuta las funciones `test_1`, `test_2` y `test_3`, cada una de ellas usa uno de los tres monitores descritos.
- ▶ Verifica que el valor obtenido es distinto del esperado en el caso del monitor sin exclusión mutua. Verifica que los otros dos monitores (con EM), el valor obtenido coincide con el esperado.
- ▶ Prueba a quitar el `unlock` de `MContador2::incrementa`.
Describe razonadamente que ocurre.

Sección 2. Monitores nativos tipo *Señalar* y *Continuar* (SC).

- 2.1. Monitor de barrera simple
- 2.2. Monitor de barrera parcial
- 2.3. Solución del Productor/Consumidor con monitores SC

Introducción

En esta sección veremos como añadir variables condición a las clases que implementan monitores. Junto con las guardas, estas variables condición permiten implementar monitores con semántica señal y continuar.

- ▶ El tipo de datos para las variables condición se denomina **variable_condition**.
- ▶ Una variable condición contiene una lista (posiblemente vacía) de hebras bloqueadas en espera de que sea cierta una determinada condición (una función lógica de las variables permanentes).
- ▶ Existen métodos de **variable_condition** para esperar y señalar.
- ▶ Las variables de este tipo se inicializan automáticamente en su declaración, es decir, al declarar **variable_condition v**, la variable **v** es inmediatamente usable.

Operaciones sobre variables condición

La clase `variable_condition` tiene los siguientes métodos:

- ▶ `wait(guarda)`, para hacer espera bloqueada, liberando durante la espera el cerrojo del monitor (se pasa como parámetro una guarda del cerrojo del monitor). La hebra que llama a `wait` debe poseer el cerrojo al llamar.
- ▶ `notify_one()`, para despertar una hebra de las que esperan (si hay alguna). La hebra señaladora (la que llama a `notify_one`) continua la ejecución tras esa llamada, si hay alguna hebra señalada se pone en espera en la cola del monitor para readquirir el cerrojo.
- ▶ `notify_all()`, para despertar todas las hebras que esperan. La hebra señaladora continua la ejecución, y las hebras señaladas esperan en la cola del monitor.

Características de las operaciones

Si hay más de una hebra esperando en una v.c., entonces una llamada a **notify_one** despierta a **una cualquiera de ellas**

- ▶ El estándar C++11 no especifica ninguna política concreta para seleccionar la hebra que reanuda la ejecución.
- ▶ El programador no debe asumir que se está usando ningún criterio concreto, y en particular no puede asumir que se selecciona la más antigua (respecto del orden de entrada en el wait).
- ▶ Las implementaciones son libres de usar cualquier criterio equitativo.
- ▶ El diseño debe hacerse de forma que todas las hebras esperando en una v.c. deben ser igualmente capaces de realizar el trabajo computacional que sigue a la reanudación.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 2. Monitores nativos tipo *Señalar* y *Continuar* (SC)

Subsección 2.1.

Monitor de barrera simple.

Un ejemplo sencillo: barrera simple

Vamos a diseñar una solución para un monitor sencillo llamado **Barrera** con una única cola condición. Los requerimientos son:

- ▶ Hay n procesos usando el monitor, los procesos ejecutan un bucle, en cada iteración realizan una actividad (un retraso aleatorio) y luego llaman al único procedimiento exportado del monitor, llamado **cita**.
- ▶ Cuando una hebra llama a **cita** en su iteración número k , no terminará la llamada antes de que todas y cada una de las otras hebras hayan hecho también su llamada número k .

Esta especificación implica que todos las hebras se esperan entre ellas tras cada iteración de su bucle, es decir, avanzan de forma **síncrona**.

Diseño de la solución

En cada iteración, cada hebra debe de esperar a que todas las demás invoquen a la cita, excepto la última en llegar, que debe despertar a las demás. Por tanto, necesitamos una variable permanente del monitor que nos indique cuantas hebras han llegado a la cita en la presente iteración. La llamamos `c`:

- ▶ Debe estar inicializada a 0.
- ▶ Cuando una hebra llega a la cita, `c` debe ser incrementado en una unidad (ya que hay una nueva hebra que ha llegado).
- ▶ Tras incrementar, si $c < n$ entonces la hebra debe esperar a las demás que faltan por llegar (faltan $n - c > 0$). Sin embargo, si $c = n$ entonces la hebra es la última y debe despertar a todas las demás.

Por tanto, debemos usar claramente una cola condición cuya condición asociada es $c == n$.

Pseudo-código de la solución

Así que el monitor puede tener esta estructura:

```
monitor MonitorBarrera1 ;  
  
var c      : integer;           { número de hebras que han llegado a cita }  
      n      : integer ;          { núm. total de hebras usando el monitor }  
      cola : Condition ;        { cola de hebras esperando  $c == n$  }  
  
procedure cita( )  
begin  
    c := c+1 ;                  { modificar estado del monitor }  
    if c < n then              { comprobar condición, si no es cierta: }  
        cola.wait();           {   esperar a que lo sea }  
    else begin  
        for i := 0 to n-2 do    {   para cada una de las otras hebras }  
            cola.signal();      {       despertar a la hebra que esperaba }  
        c := 0 ;                 {   reinicializar c para siguiente iter. }  
    end  
end  
begin                         { inicialización: }  
    c := 0 ;                   {     hay 0 hebras en la cita }  
    n := ..... ;              {     número de hebras, arbitrario, ( $n > 1$ ) }  
end
```

Implementación en C++11: declaración de la clase

Ahora veremos como implementar esta solución en C++11, usando guardas de cerrojo para E.M. y variables condición (en el archivo **barrera1_sc.cpp**)

La declaración de la clase tiene esta forma:

```
#include ..... // includes varios (iostream,thread,mutex,random,...)
#include <condition_variable> // tipo std::condition_variable
using namespace std ;

class MBarreraSC
{
    private:                                // variables privadas:
        int           cont,                  // contador de hebras en cita
                    num_hebras ;          // número total de hebras
    mutex         cerrojo_monitor;        // cerrojo del monitor
    condition_variable cola ;             // cola de hebras esperando en cita

    public:                                 // metodos públicos:
        MBarreraSC( int p_num_hebras ) ;   // constructor (inicialización)
        void cita( int num_hebra );        // método de cita
};
```

Implementación en C++11: constructor y cita

El constructor y el método **cita** se implementan así:

```
MBarreraSC::MBarreraSC( int p_num_hebras )
{
    num_hebras = p_num_hebras ;
    cont       = 0 ;
}
void MBarreraSC::cita( int num_hebra )
{
    unique_lock<mutex> guarda( cerrojo_monitor ); // ganar E.M.

    cont ++ ;
    const int orden = cont ; // copia local del contador (para la traza)
    cout << "Llega hebra " << num_hebra << " (" << orden << ")." << endl ;
    if ( cont < num_hebras )
        cola.wait( guarda ); // wait accede al cerrojo del monitor
    else
    {   for( int i = 0 ; i < num_hebras-1 ; i++ )
        cola.notify_one() ;
        cont = 0 ;
    }
    cout << "      Sale hebra " << num_hebra << " (" << orden << ")." << endl ;
}
```

Implementación en C++11: código de las hebras

La función de las hebras recibe como parámetros

- ▶ un puntero al monitor (**mon_barrera1**)
- ▶ el número de hebra (comenzando en 0, hasta **num_hebras-1**)

```
void funcion_hebra( MBarreraSC * monitor, int num_hebra )
{
    while( true )
    {
        const int ms = aleatorio<10,100>();
        this_thread::sleep_for( chrono::milliseconds( ms ) );
        monitor->cita( num_hebra );
    }
}
```

Implementación en C++11: hebra principal

La hebra principal crea el monitor y pone en marcha el resto de hebras:

```
int main()
{
    const int num_hebras = 10 ; // número total de hebras

    // crear el monitor
    MBarreraSC monitor( num_hebras );

    // crear y lanzar hebras
    thread hebra[num_hebras];
    for( unsigned i = 0 ; i < num_hebras ; i++ )
        hebra[i] = thread( funcion_hebra, &monitor, i );

    // esperar a que terminen las hebras (no ocurre nunca)
    for( unsigned i = 0 ; i < num_hebras ; i++ )
        hebra[i].join();
}
```

Características de los monitores SC

Hay que tener en cuenta estos aspectos relativos los monitores SC en general:

- ▶ El método **wait** de **condition_variable** accede al cerrojo del monitor a través de la variable local **guarda** (que tiene dentro una referencia a dicho cerrojo).
- ▶ El método **wait** libera temporalmente el cerrojo del monitor (mientras que la hebra espera). Cuando la hebra es señalada, espera en la cola del monitor hasta que pueda readquirir el cerrojo, entonces continua la ejecución de las sentencias que haya después de la llamada a **wait**.
- ▶ La hebra señaladora, tras ejecutar **notify_one**, continua su ejecución y tiene el cerrojo hasta que sale del monitor (termina el método que está ejecutando).

Compilación de programas con monitores nativos

Para compilar los programas con monitores nativos no es necesario compilar también el archivo **scd.cpp**, es decir, para compilar un archivo *f.cpp*, y crear el ejecutable **ejecutable_exe**, bastaría poner:

```
g++ -std=c++11 -pthread -o ejecutable_exe f.cpp
```

Además, los programas que usen la plantilla de función **aleatorio** deben de:

- ▶ hacer **#include** de **scd.h**
- ▶ hacer **using** del **namespace scd**

Traza del programa

La salida del programa (para $n = 6$) en una iteración de todas las hebras es típicamente así:

```
Llega hebra 5 (1).
Llega hebra 1 (2).
Llega hebra 3 (3).
Llega hebra 2 (4).
Llega hebra 0 (5).
Llega hebra 4 (6).
    Sale hebra 4 (6).
    Sale hebra 5 (1).
    Sale hebra 3 (3).
    Sale hebra 2 (4).
    Sale hebra 1 (2).
    Sale hebra 0 (5).
```

Los números entre paréntesis expresan los números de orden de entrada a la cita.

Actividades relativas al monitor Barrera1 (SC)

Describe razonadamente en tu portafolio el motivo de cada uno de estos tres hechos:

1. La hebra que entra la última al método **cita** (la hebra señaladora) es siempre la primera en salir de dicho método.
2. El orden en el que las hebras señaladas logran entrar de nuevo al monitor no siempre coincide con el orden de salida de **wait** (se observa porque los números de orden de entrada no aparecen ordenados a la salida).
3. El constructor de la clase no necesita ejecutarse en exclusión mutua.

Prueba a usar **notify_all** en lugar de **notify_one**. Describe razonadamente en tu portafolio si se observa o no algún cambio importante en la traza del programa.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 2. Monitores nativos tipo *Señalar* y *Continuar* (SC)

Subsección 2.2.

Monitor de barrera parcial.

Requerimientos

Para ilustrar mejor las características de la semántica SC, vamos a analizar una modificación del monitor barrera simple que acabamos de ver:

- ▶ En este nuevo monitor, vamos a suponer que la espera de la cita (en cada iteración) no será una espera de todas las hebras en ejecución, sino solo de un subconjunto de ellas (por eso es una barrera *parcial*)
- ▶ Suponemos ahora que hay un total de n hebras en ejecución, mientras que en la cita solo se espera a que lleguen m hebras ($m < n$). En las pruebas faremos $m = 10$ y $n = 100$.
- ▶ Por tanto, cada conjunto de m hebras que llegan de forma consecutiva a la cita se esperan entre ellas.

Diseño de la solución

Por tanto, la solución con un monitor es similar a la barrera simple que ya hemos visto, solo que substituyendo n (número total de hebras) por m (número de hebras en la cita)

- ▶ Ahora creamos una nueva versión del monitor, en la cual el constructor del monitor recibe como parámetro el valor de m , en lugar de n . La cola tiene asociada la condición $c = m$, en lugar de $c = n$.
- ▶ En la implementación, sustituimos `num_hebras` por `num_hebras_cita` en la declaración de la clase monitor y en la implementación de sus métodos.

Como actividad, puedes probar a compilar y ejecutar el monitor **MBarreraParSC**, en el archivo `barrera2_sc.cpp`.

Traza de la barrera parcial

Podemos analizar la traza de la barrera parcial, observamos esto:

- ▶ Al igual que en la barrera simple, la hebra señaladora (última de cada grupo), es siempre la primera en salir (de ese grupo)
- ▶ Igualmente, el orden de salida no coincide necesariamente con el de entrada (en cada grupo)
- ▶ Además, ahora las hebras señaladas compiten por el cerrojo con otras hebras que quieren comenzar a ejecutar **cita**, así que las salidas de un grupo no son consecutivas: se alternan con las entradas de otras hebras que no son del grupo.

Propiedades adicionales

Supongamos que añadimos nuevas propiedades a la barrera parcial:

- ▶ Hasta que no termina de salir un grupo completo de m hebras, no se permitirá que ninguna otra hebra fuera del grupo pueda comenzar a ejecutar **cita** (las entradas y salidas de un grupo se hacen consecutivas).
- ▶ El orden de salida de **cita** debe coincidir con el orden de entrada a la misma.

Esto se puede conseguir fácilmente, usando el mismo diseño de solución, pero con un **monitor tipo Hoare**, es decir, un monitor **con semántica SU** y además con **orden FIFO garantizado** (lo veremos más adelante).

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 2. Monitores nativos tipo *Señalar* y *Continuar* (SC)

Subsección 2.3.

Solución del Productor/Consumidor con monitores SC.

El productor/consumidor en monitores

En estas sub-sección veremos como diseñar e implementar una solución al problema del productor/consumidor (con una hebra en cada rol), usando un monitor SC. 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 SC 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 ProdConsSC

```
.....  
end
```

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

Process Consumidor ;
var dato : integer ;
begin
for i := 1 to m do begin
dato := ProdConsSC.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 cuantas celdas hay ocupadas, por tanto necesitamos una variable adicional con el valor de n .

Pseudocódigo del monitor

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

Monitor ProdConsSC

```
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_sc.cpp` puedes encontrar una implementación del monitor **ProdConsSC** descrito, con semántica SC (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. Describe razonadamente en tu portafolio los cambios que esto supone.

Actividad: múltiples productores y consumidores

Copia el archivo `prodcons1_sc.cpp` en `prodcons2_sc.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 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, 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).

Semántica SC en múltiples prod/cons: implicaciones

La versión de múltiples productores y consumidores mantiene la verificación final de que cada item es consumido una y solo una vez:

- ▶ Para que el programa sea correcto, debes de cambiar las sentencias **if** en **extraer** e **insertar**, por bucles **while** (manteniendo la condición).
- ▶ Comprueba que esto es realmente así, es decir, observa que con el uso de **if** se produce un error en las verificaciones que el programa hace, pero con **while** se ejecuta correctamente.
- ▶ Describe razonadamente en tu portafolio a que se debe que (con semántica SC), la versión para múltiples productores y consumidores deba usar **while**, mientras que la versión para un único productor y consumidor puede usar simplemente **if**.

Sección 3. Monitores tipo *Señalar* y *Espera Urgente* (SU) .

- 3.1. Monitor de barrera parcial con semántica SU
- 3.2. Productor/Consumidor con semántica SU

Monitores Señalar y Espera Urgente (SU)

En los monitores **con semántica Señalar y Espera Urgente (SU)**, cuando una hebra señaladora hace **signal** en una cola donde hay una hebra señalada esperando:

- ▶ La hebra señalada adquiere el cerrojo del monitor, y continua ejecutando las sentencias que siguen al **wait**.
- ▶ La hebra señaladora pasa a esperar a una **cola especial, llamada cola de urgentes.**

Cuando una hebra libera el cerrojo del monitor (bien por entrar en **wait**, bien por salir de un procedimiento del monitor):

- ▶ Si **hay hebras esperando en la cola de urgentes, la que antes entró se libera y continua ejecutando código tras el signal.**
- ▶ Si **no hay hebras en la cola de urgentes, pero hay en la cola del monitor, una de ellas puede acceder al mismo.**

Monitores SU: propiedades

Los monitores SU suelen permitir diseños más simples en muchos casos:

- ▶ La hebra señalada no tiene que competir con otras hebras para adquirir el cerrojo del monitor y reanudar la ejecución.
- ▶ Cada hebra espera en la cola del monitor como mucho una vez como consecuencia de una llamada a un procedimiento del monitor.
- ▶ Está garantizado que la hebra señalada reanuda su ejecución inmediatamente tras **signal**, ninguna otra puede acceder.
- ▶ Como consecuencia, la hebra señalada tiene garantizado que, al salir de **wait**, se cumple la condición que espera.

Todo esto permite en muchos casos diseños más simples. Veremos el ejemplo del monitor de barrera parcial.

Monitores SU en C++11

El lenguaje C++11 y la librería estándar no incluye la posibilidad de definir monitores SU directamente. Pero se pueden usar los cerrojos y las variables condición para construir estos monitores.

- ▶ Se ha construido una clase base para monitores SU.
- ▶ La adquisición y liberación de la E.M. se hace de forma transparente al programador.
- ▶ Esta clase se implementa usando *objetos mutex, guardas de cerrojo y variables condición*, todo ello características nativas de C++11.
- ▶ Esta implementación garantiza que siempre habrá orden FIFO en las colas de las variables condición.

Clases para monitores SU

Para construir monitores SU hay que:

- ▶ 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)
- ▶ Declarar las variables condición como variables de instancia de tipo **CondVar**
- ▶ En el constructor, inicializar cada variable condición llamando a **newCondVar** e inicializar cada variable permanente al valor adecuado.
- ▶ En **main**, crear una instancia del monitor.

Veremos el ejemplo de la barrera parcial usando un monitor SU.

Operaciones sobre variables condición

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

- ▶ Método **wait()** (sin argumentos): la hebra que invoca espera bloqueada hasta que otra hebra haga **signal** sobre la cola.
- ▶ Método **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).
- ▶ Método **get_nwt()**: devuelve el número de hebras esperando en la cola.
- ▶ Método **empty()**: devuelve **true** si no hay hebras esperando, o **false** si hay al menos una.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 3. Monitores tipo *Señalar* y *Espera Urgente* (SU)

Subsección 3.1.

Monitor de barrera parcial con semántica SU.

Declaración de la clase

Veremos el **ejemplo del monitor de barrera parcial** (en el archivo **barrera2_su.cpp**): partimos del diseño usado para SC y lo trasladamos al monitor SU. La clase se declara usando la clase base **HoareMonitor** y la clase **CondVar**:

```
#include ..... // iostream, random, etc...
#include "scd.h"

using namespace std ;
using namespace scd ;

class MBarreraParSU : public HoareMonitor // clase derivada (pública)
{
    private:
        int      cont,           // contador de hebras en cita
                num_hebras_cita ; // número total de hebras en cita
        CondVar cola ;          // cola de hebras esperando en cita

    public:
        MBarreraParSU( int p_num_hebras_cita ); // constructor
        void cita( int num_hebra );             // método de cita
} ;
```

Constructor y métodos

Usamos **newCondVar**, **wait** y **signal**:

```
MBarreraParSU::MBarreraParSU( int p_num_hebras_cita )
{
    num_hebras_cita = p_num_hebras_cita ; // total de hebras en cita (> 1)
    cont            = 0 ;                // hebras actualmente en cita
    cola            = newCondVar();     // cola de espera
}
void MBarreraParSU::cita( int num_hebra )
{
    cont ++ ;                      // una hebra más ha llegado a la cita
    const int orden = cont ;        // guarda numero de orden de llegada
    cout << "Llega hebra " <<num_hebra << " (" <<orden <<")." <<endl ;

    if ( cont < num_hebras_cita ) // si no han llegado todas la hebras:
        cola.wait();           //     espera bloqueado en la cola
    else                         // si ya han llegado todas las demás:
    {   for( int i = 0 ; i < num_hebras_cita-1 ; i++ ) // para cada una:
        cola.signal();          //         reanudar hebra
        cont = 0 ;               // ini. contador
    }
    cout << " Sale hebra " <<num_hebra << " (" <<orden <<")." <<endl ;
}
```

Función que ejecutan las hebras

La función que ejecutan las hebras recibe como parámetro una referencia a la instancia del monitor que van a usar (se usa el tipo **MRef**: encapsula un puntero o referencia al monitor).

Esta referencia permite gestionar automáticamente la exclusión mutua del monitor (el código de **cita** no menciona el cerrojo del monitor)

```
void funcion_hebra( MRef<MBarreraParSU> monitor, int num_hebra )
{
    while( true )
    { const int ms = aleatorio<0,30>();           // duración aleatoria
        this_thread::sleep_for( chrono::milliseconds(ms) ); // espera bloqueada
        monitor->cita( num_hebra );                      // invocar cita con ->
    }
}
```

Hebra principal

La hebra principal crea una instancia del monitor, y usa un objeto de tipo **MRef** que referencia a dicha instancia (y que se pasa a las hebras como parámetro)

```
int main()
{
    const int num_hebras      = 100, // número total de hebras
            num_hebras_cita = 10 ; // número de hebras en cita

    // crear monitor ('monitor' es una referencia al mismo, de tipo MRef<...>)
    MRef<MBarreraParSU> monitor = Create<MBarreraParSU>( num_hebras_cita );

    // crear y lanzar todas las hebras (se les pasa ref. a monitor)
    thread hebra[num_hebras];
    for( unsigned i = 0 ; i < num_hebras ; i++ )
        hebra[i] = thread( funcion_hebra, monitor, i );

    // esperar a que terminen las hebras (no pasa nunca)
    for( unsigned i = 0 ; i < num_hebras ; i++ )
        hebra[i].join();
}
```

Compilar programas con monitores SU

Para compilar en la línea de órdenes programas con monitores SU, se hace igual que antes, solo que ahora añadimos la unidad de compilación **HoareMonitor.cpp**. Para compilar el ejemplo de la barrera parcial (archivo **barrera2_su.cpp**) escribimos (en una sola línea de órdenes)

```
g++ -std=c++11 -pthread -o barrera2_su_exe barrera2_su.cpp scd.cpp
```

Esta orden genera el archivo ejecutable **barrera2_su_exe**.

En la misma carpeta del programa es necesario tener disponibles los archivos **scd.cpp** y **scd.h**, al igual que en el seminario 1 y la práctica 1.

Traza de la barrera parcial con semántica SU

Si ejecutamos `barrera2_su_exe`, ahora vemos una salida como esta:

```
Llega hebra 88 ( 1).
Llega hebra 49 ( 2).
Llega hebra 31 ( 3).
Llega hebra 52 ( 4).
Llega hebra 32 ( 5).
Llega hebra 94 ( 6).
Llega hebra 35 ( 7).
Llega hebra 86 ( 8).
Llega hebra 25 ( 9).
Llega hebra 66 (10).

        Sale hebra 88 ( 1).
        Sale hebra 49 ( 2).
        Sale hebra 31 ( 3).
        Sale hebra 52 ( 4).
        Sale hebra 32 ( 5).
        Sale hebra 94 ( 6).
        Sale hebra 35 ( 7).
        Sale hebra 86 ( 8).
        Sale hebra 25 ( 9).
        Sale hebra 66 (10).
```

Propiedades de la barrera parcial con semántica SU

De la traza se deduce lo siguiente:

- ▶ Hasta que todas las hebras de un grupo han salido de la cita, ninguna otra hebra que no sea de ese grupo logra entrar.
- ▶ El orden de salida de la cita coincide siempre con el orden de entrada a la misma.

Describe razonadamente en tu portafolio a que se debe que ahora, con la semántica SU, se cumplan estas propiedades.

Sistemas Concurrentes y Distribuidos., curso 2021-22.

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

Sección 3. Monitores tipo *Señalar* y *Espera Urgente* (SU)

Subsección 3.2.

Productor/Consumidor con semántica SU.

Actividad

A modo de actividad, puedes partir de tu implementación de la solución al problema del productor/consumidor (con múltiples productores/consumidores y con semántica SC), y construir una solución equivalente con semántica SU:

- ▶ Adapta la versión SC para SU, usando como referencia el código de la barrera parcial SU.
- ▶ Implementa la solución LIFO y la FIFO.
- ▶ Comprueba que la verificación final es correcta en los dos casos.
- ▶ Verifica si en el caso de la semántica SU es también necesario poner las operaciones **wait** dentro de un bucle **while**, o bien podemos sustituir dichos bucles por sentencias **if**.
- ▶ Describe razonadamente en tu portafolio a que se debe el resultado que has obtenido en el punto anterior.

Fin de la presentación.



UNIVERSIDAD
DE GRANADA

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

Carlos Ureña / Jose M. Mantas / Pedro Villar
2021-22

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

Seminario 3. Introducción a paso de mensajes con MPI.

Índice.

1. Message Passing Interface (MPI)
2. Compilación y ejecución de programas MPI
3. Funciones MPI básicas
4. Paso de mensajes síncrono en MPI
5. Sondeo de mensajes
6. Comunicación insegura

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.

uso MPI
+ implementacion

OPEN MPI .

Sistemas Concurrentes y Distribuidos., curso 2021-22.
Seminario 3. Introducción a paso de mensajes con 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 grupo 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.

Compilación y ejecución de programas en OpenMPI

OpenMPI 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.

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, p.ej:

```
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
```

Compilación y ejecución de programas MPI

La forma más usual de ejecutar un programa MPI 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

Sección 3. Funciones MPI básicas.

3.1. Introducción a los comunicadores

3.2. Funciones básicas de envío y recepción de mensajes

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

Sistemas Concurrentes y Distribuidos., curso 2021-22.
Seminario 3. Introducción a paso de mensajes con MPI.
Sección 3. Funciones MPI básicas

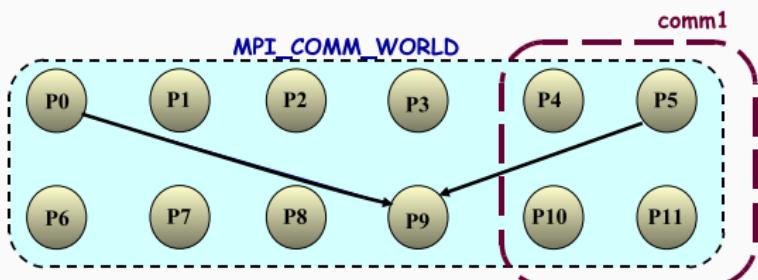
Subsección 3.1.
Introducción a los comunicadores.

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.

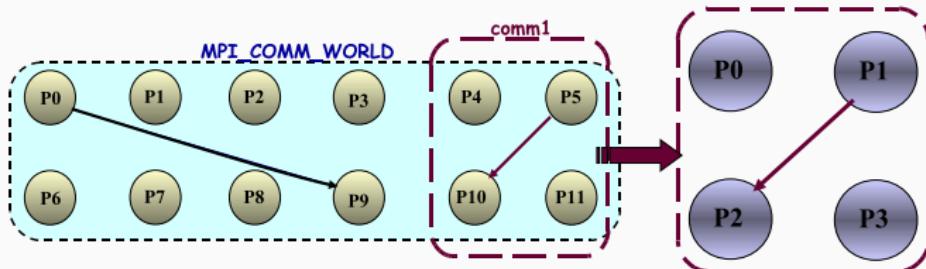
Todas las funciones de comunicación de MPI necesitan como argumento un comunicador.



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 **única**:
- ▶ 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
```

Sistemas Concurrentes y Distribuidos., curso 2021-22.

Seminario 3. Introducción a paso de mensajes con MPI.

Sección 3. Funciones MPI básicas

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 envío **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 iniciado ya la recepción del mensaje (**el envío es asíncrono**)

Recepción segura síncrona de un mensaje (MPI_Recv)

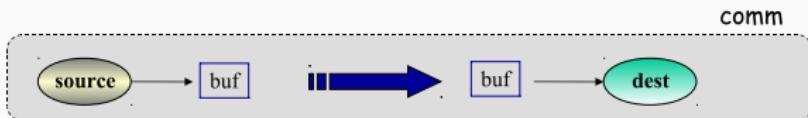
Un proceso puede recibir un mensaje usando **MPI_Recv**, que se declara como sigue:

```
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, n m.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 env o o recepci n (seg n 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 explícitamente 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

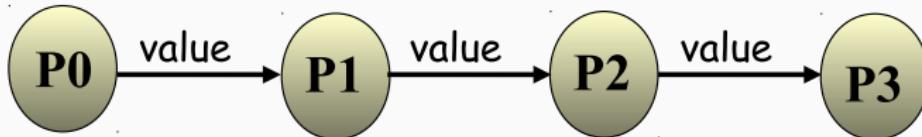
Es importante tener en cuenta que para determinar el emparejamiento MPI **no tiene en cuenta el tipo de datos ni la cuenta de items**. Es responsabilidad del programador asegurarse de que, en el lado del receptor:

- ▶ **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 = id_propio; // 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 metadatos 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( unsigned 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 envío 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.