



UNIVERSIDAD  
DE GRANADA

# Sistemas Concurrentes y Distribuidos:

## Seminario 1. Programación multihebra y semáforos.

---

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

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

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

## Seminario 1. Programación multihebra y semáforos.

### Índice.

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:

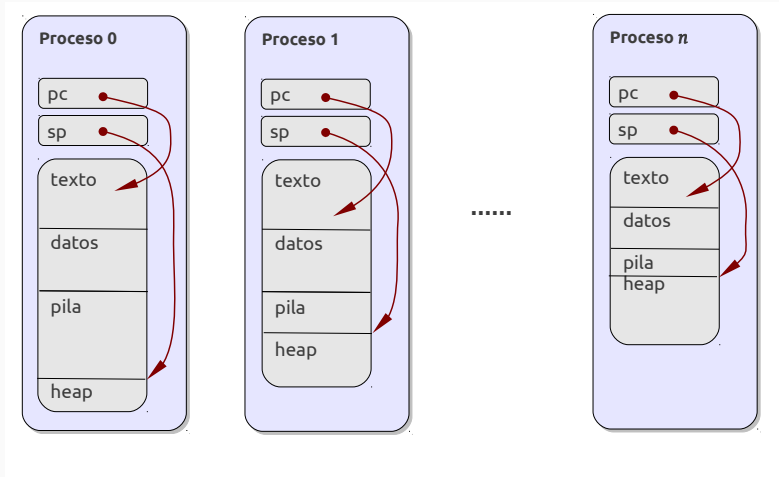
- ▶ **texto:** zona (tamaño fijo) con las instrucciones del programa
- ▶ **datos:** espacio (de tamaño fijo) para variables globales.
- ▶ **pila:** espacio (de tamaño cambiante) para variables locales.
- ▶ **mem. dinámica (*heap*):** espacio ocupado por variables dinámicas.

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

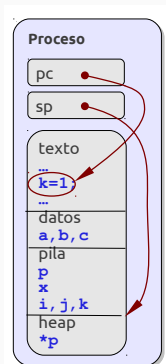
En el siguiente programa escrito en C/C++, el estado del proceso (durante la ejecución de **k=1;**) es el que se ve a la derecha:

```
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

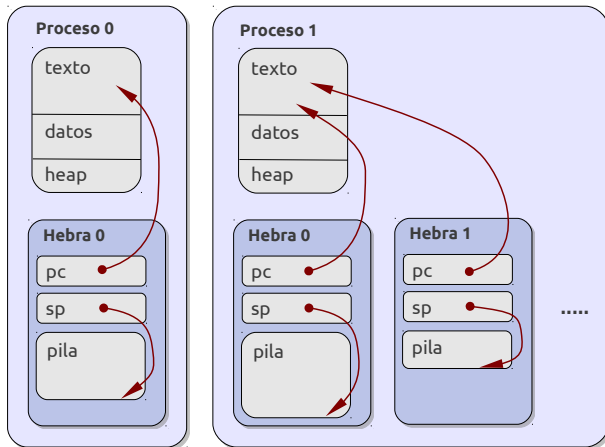
para mayor eficiencia en esta situación se diseñó el concepto de **hebra**:

- ▶ 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

Podríamos visualizarlos (simplificadamente) como sigue:



# Inicio y finalización de hebras

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

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

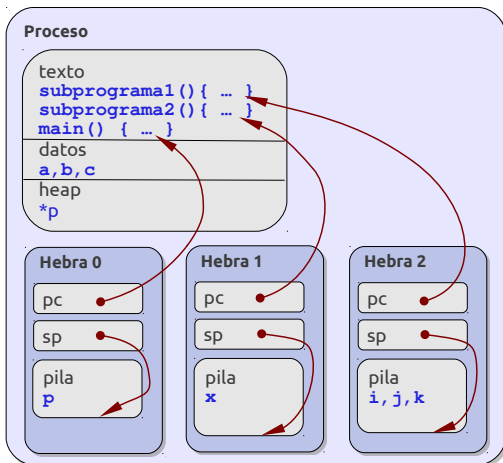
# Ejemplo de estado de un proceso con tres hebras

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

```
int a,b,c ;

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

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





## Sección 2. Hebras en C++11.

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

# El estándar 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 estándar **ISO/IEC 14882:2011**:
  - ▶ Página del estándar en la web de ISO:  
 <https://www.iso.org/standard/50372.html>
  - ▶ Borrador revisado en PDF:  
 <https://github.com/cplusplus/draft/blob/master/papers/n3337.pdf>
- ▶ Hay revisiones posteriores de C++ (2014, 2017, 2020), pero no modifican las características que veremos aquí.
- ▶ Los fuentes C++ que usan este estándar 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 estándar.

Sistemas Concurrentes y Distribuidos, curso 2024-25.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.1.

Introducción a las hebras en C++11.

# Introducción

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

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



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

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



# Prerequisitos

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

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

# Edición de archivos. Terminal a usar.

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

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

# Codificación de archivos fuente.

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

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

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

# Compilar con la línea de órdenes en Linux y macOS

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

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

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

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

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

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

# Compilar con la línea de órdenes en Windows

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

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

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

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

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

Sistemas Concurrentes y Distribuidos, curso 2024-25.  
Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

## Subsección 2.3. Creación y finalización de hebras.

# Creación de hebras

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

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

# Ejemplo de creación de hebras.

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

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

void funcion_hebra_1( ) // función que va a ejecutar la hebra primera
{ for( unsigned long i = 0 ; i < 5000 ; i++ )
    cout << "hebra 1, i == " << i << endl ;
}

void funcion_hebra_2( ) // función que va a ejecutar la hebra segunda
{ for( unsigned long i = 0 ; i < 5000 ; i++ )
    cout << "hebra 2, i == " << i << endl ;
}

int main()
{
    thread hebra1( funcion_hebra_1 ), // crear hebra1 ejecutando funcion_hebra_1
                 hebra2( funcion_hebra_2 ); // crear hebra2 ejecutando funcion_hebra_2

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

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



# Declaración e inicio separados

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

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

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

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

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

# Finalización de hebras

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

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

Todas las hebras en ejecución de un programa finalizan cuando:

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

Sistemas Concurrentes y Distribuidos, curso 2024-25.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.4.

Sincronización mediante unión.

# Terminación incorrecta

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

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

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

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

# La operación de unión.

C++11 provee diversos mecanismos para sincronizar hebras:

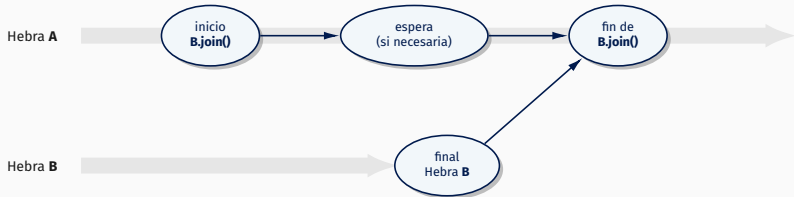
- ▶ Usando la operación de **unión** (*join*).
- ▶ Usando *mutex* o *variables condición*

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

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

# Sincronización asociada a una unión.

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



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

# El método *join* de la clase *thread*

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

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

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

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

# Uso correcto de la unión. Hebras activas.

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

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

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

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

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



Sistemas Concurrentes y Distribuidos, curso 2024-25.

Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.5.

Paso de parámetros y obtención de un resultado.

# Parámetros y resultados

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

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



## Paso de parámetros a hebras

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

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

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

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

o bien, usando declaración e inicio separados:

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

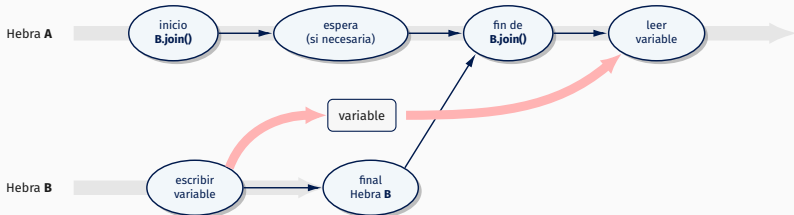
# Métodos de obtención de valores resultado

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

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

# Resultado en variable compartida. Sincronización.

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



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

# Obtención de valores resultado

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

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

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

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

# Uso de variables globales

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

```
.....  
// variables globales donde se escriben los resultados  
long resultado1, resultado2 ;  
  
// funciones que ejecutan las hebras  
void funcion_hebra_1( int n ) { resultado1 = factorial( n ) ; }  
void funcion_hebra_2( int n ) { resultado2 = factorial( n ) ; }  
  
int main()  
{  
    // iniciar las hebras  
    thread hebra1( funcion_hebra_1, 5 ), // calcula factorial(5) en resultado1  
               hebra2( funcion_hebra_2, 10 ); // calcula factorial(10) en resultado2  
  
    // esperar a que terminen las hebras,  
    hebra1.join() ; hebra2.join() ;  
  
    // imprimir los 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 2024-25.  
Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

Subsección 2.6.

Vectores de hebras y futuros.

# Hebras idénticas

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

- ▶ Es necesario que cada hebra reciba parámetros distintos, lo cual permite que operen sobre datos distintos.
- ▶ Un caso muy común es que cada hebra reciba un número de orden o identificador de la hebra distinto, empezando en 0.
- ▶ Por simplicidad, se puede usar un vector de variables de tipo hebra o variables de tipo futuro.
- ▶ Lo anterior permite, además, que el número de hebras sea un parámetro configurable, sin cambiar el 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 2024-25.  
Seminario 1. Programación multihebra y semáforos.

Sección 2. Hebras en C++11

## Subsección 2.7. Medición de tiempos.

# Medición de tiempo real

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

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



# Relojes

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

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


Para medir tiempos, usaremos el reloj **steady\_clock**

# Medición de tiempos con el reloj monotónico

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

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

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



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

# Cálculo numérico de integrales

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

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

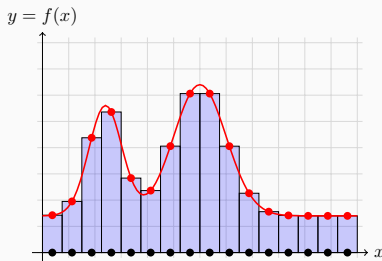
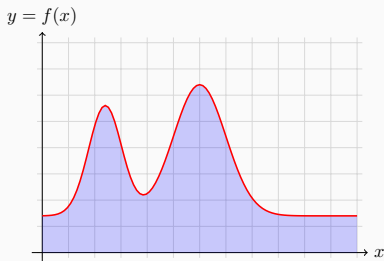
$$I = \int_0^1 f(x) dx$$

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

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

# 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 $\pi$

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  $\pi$ :

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

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
}
```

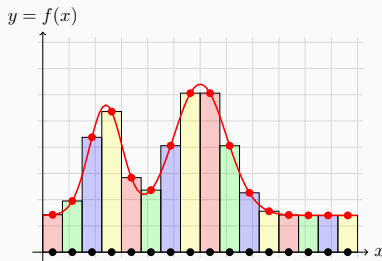
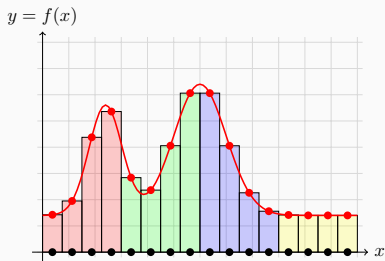
# Versión concurrente de la integración

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 evalúa  $f$  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.

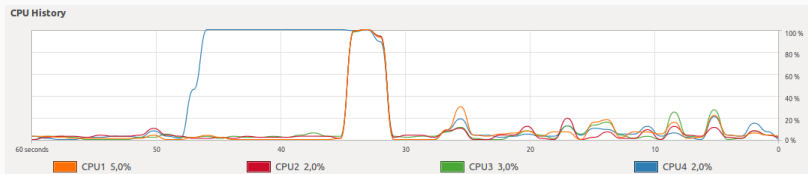
Como actividad se propone copiar la plantilla en `ejemplo09.cpp` y completar en este archivo la implementación del cálculo concurrente del número  $\pi$ , tal y como hemos visto aquí:

- ▶ En la salida se presenta el valor exacto de  $\pi$  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:

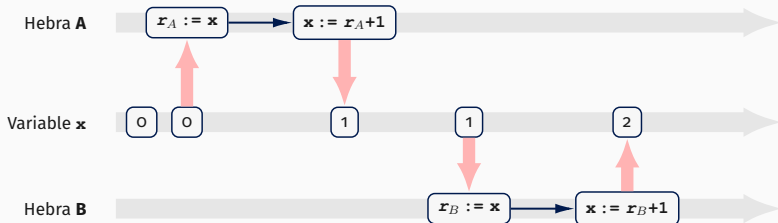
- ▶ Incrementar o decrementar una variable entera o flotante se hace en varias instrucciones atómicas distintas. En particular, puede que dos incrementos simultáneos de una variable entera dejen la variable con una unidad más en lugar de dos unidades más, como cabe esperar.
- ▶ Insertar o eliminar un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

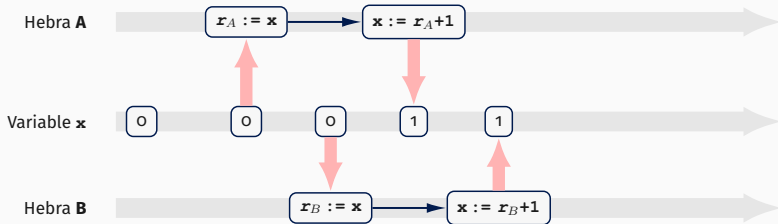
Si una hebra lee después de que la otra escriba, el valor final de  $x$  es 2:





## 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 2024-25.

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. Métodos y operadores

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

- (a) Evaluar la expresión  $e$  y obtener su valor,  $n$ .
- (b) Ejecutar estos tres pasos **de forma atómica**:
  1. Leer el valor entero actual de  $a$ .
  2. Incrementar (o decrementar) el valor de  $a$  en  $n$  unidades.
  3. Devolver el valor leído en el paso 1.

Se pueden usar operadores tradicionales de C:

<code>a++</code> $\equiv$ <code>a.fetch_add(1)</code>	<code>++a</code> $\equiv$ <code>a.fetch_add(1)+1</code>
<code>a--</code> $\equiv$ <code>a.fetch_sub(1)</code>	<code>--a</code> $\equiv$ <code>a.fetch_sub(1)-1</code>
<code>a+=e</code> $\equiv$ <code>a.fetch_add(n)+n</code>	<code>a-=e</code> $\equiv$ <code>a.fetch_sub(n)-n</code>

# Atomicidad de las operaciones.

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

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

# Ejemplo de tipos atómicos (1/2)

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

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

const long  num_iters = 10000001 ; // número de incrementos a realizar
int         contador_no_atom ;    // contador compartido (no atómico)
atomic<int>  contador_atom ;      // contador compartido (atómico)

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 milisegundos.
```

- ▶ Los incrementos atómicos producen el valor final de **contador\_atom** esperado, esto es, el doble del número de iteraciones ( **$2 * \text{num\_iters}$** ).
- ▶ 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 sobrescrita después la variable por la otra hebra).
- ▶ Hay una diferencia en los tiempos significativa (¿ a que se debe esta diferencia ?)

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

## Subsección 3.2. Objetos *Mutex*.



# Introducción

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

- ▶ Para ello podemos usar los **objetos mutex** (también llamados *cerrojos* (*locks*)).
- ▶ El estándar 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 bloqueada.

# 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
}
```

# 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 también 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 milisegundos.
```

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 2024-25.  
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 un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

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

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

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



# Operaciones sobre los semáforos

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

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

# Propiedades de los semáforos

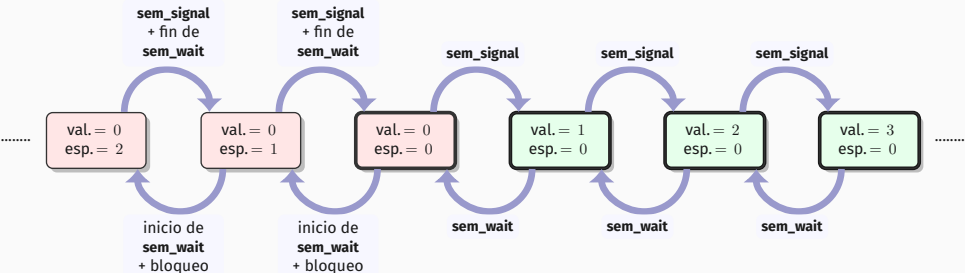
Los semáforos cumplen estas propiedades:

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

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

# Diagrama de estados de un semáforo

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



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

# Patrones de solución de problemas de sincronización

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

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

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

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

## Subsección 4.2. Espera única.

# Problema de sincronización

El problema básico de sincronización ocurre cuando:

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

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

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

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

Este programa **no funciona correctamente**.

# Condición de sincronización

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

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

# Solución con un semáforo

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

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

Por tanto:

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



# Pseudo-código de la solución

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

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

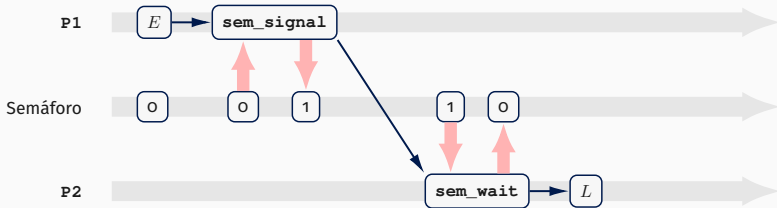
```
process P1 ;  
    var local1 : integer ;  
begin  
    .....  
    local1 := ProducirValor() ;  
    compartida := local1 ; { E }  
    sem_signal( puede_leer ) ;  
    .....  
end
```

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

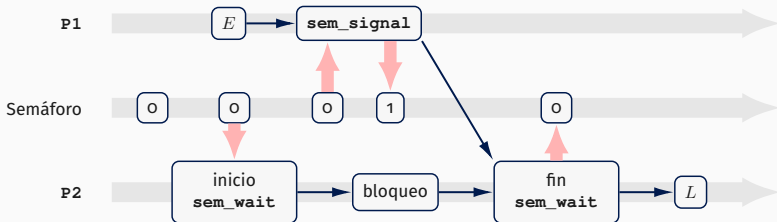
El programa introduce una espera que asegura que *L* se ejecuta después que *E*. Es decir **se cumple la condición de sincronización**

# Posibles trazas: esquema

Si **P1** inicia **sem\_signal** antes de que **P2** inicie **sem\_wait**:



Si **P1** inicia **sem\_signal** después de que **P2** inicie **sem\_wait**:



# Posibles trazas: explicación

Solo hay dos posibles interfoliaciones de las sentencias relevantes:

Si **P1** inicia **sem\_signal** antes de que **P2** inicie **sem\_wait**:

1. **P1** ejecuta **sem\_signal** y el semáforo queda a 1 (**P1** ya ha ejecutado *E* antes).
2. **P2** ejecuta **sem\_wait**, ve el semáforo a 1, lo baja a 0 y termina el **sem\_wait**.
3. **P2** ejecuta *L*

Si **P1** inicia **sem\_signal** después de que **P2** inicie **sem\_wait**:

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

Sistemas Concurrentes y Distribuidos, curso 2024-25.  
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*. Es un código que se traduce en dos o más instrucciones atómicas, a la primera de ellas la llamamos *I* y a la última *F*

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

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

# Condición de sincronización

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

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

# Solución con un semáforo

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

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

Por tanto:

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

# Pseudo-código con un semáforo

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

```
{ variables compartidas y valores iniciales }  
var sc_libre : semaphore := 1 ; { 1 si S.C. libre, 0 si S.C. ocupada }  
  
process ProcesosEM[ i : 0..n-1 ] ;  
begin  
    while true do begin  
        sem_wait( sc_libre ); { esperar hasta que "sc_libre" sea 1 }  
        { aquí va la sección crítica: I .... F }  
        sem_signal( sc_libre ); { desbloquear o poner "sc_libre" a 1 }  
        { resto de sentencias: ..... }  
    end  
end
```

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



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

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

# El problema del productor-consumidor

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

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

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

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

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

# Condición de sincronización

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

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

# Uso de semáforos para este problema (1/3)

Para lograr la sincronización requerida, usaremos dos semáforos:

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

Esto hace necesario

- ▶ que el productor haga **sem\_wait** de uno de los semáforos inmediatamente antes de  $E$ ,
- ▶ que el consumidor haga **sem\_wait** del otro semáforo inmediatamente antes de  $L$ , y
- ▶ que se hagan los **sem\_signal** correspondientes y que los valores iniciales sean correctos.

## Uso de semáforos para este problema (2/3)

El semáforo donde espera el productor se llamará **puede\_escribir**:

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

El semáforo donde espera el consumidor se llamará **puede\_leer**:

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

## Uso de semáforos para este problema (3/3)

Con estos semáforos, el productor debe:

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

Por otro lado, el consumidor debe:

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

# Uso de semáforos para sincronización

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

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

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

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

## Sección 5. Semáforos en C++11.

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



Sistemas Concurrentes y Distribuidos, curso 2024-25.  
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
```

## 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.