Seminario 1: Programación multihebra y semáforos

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

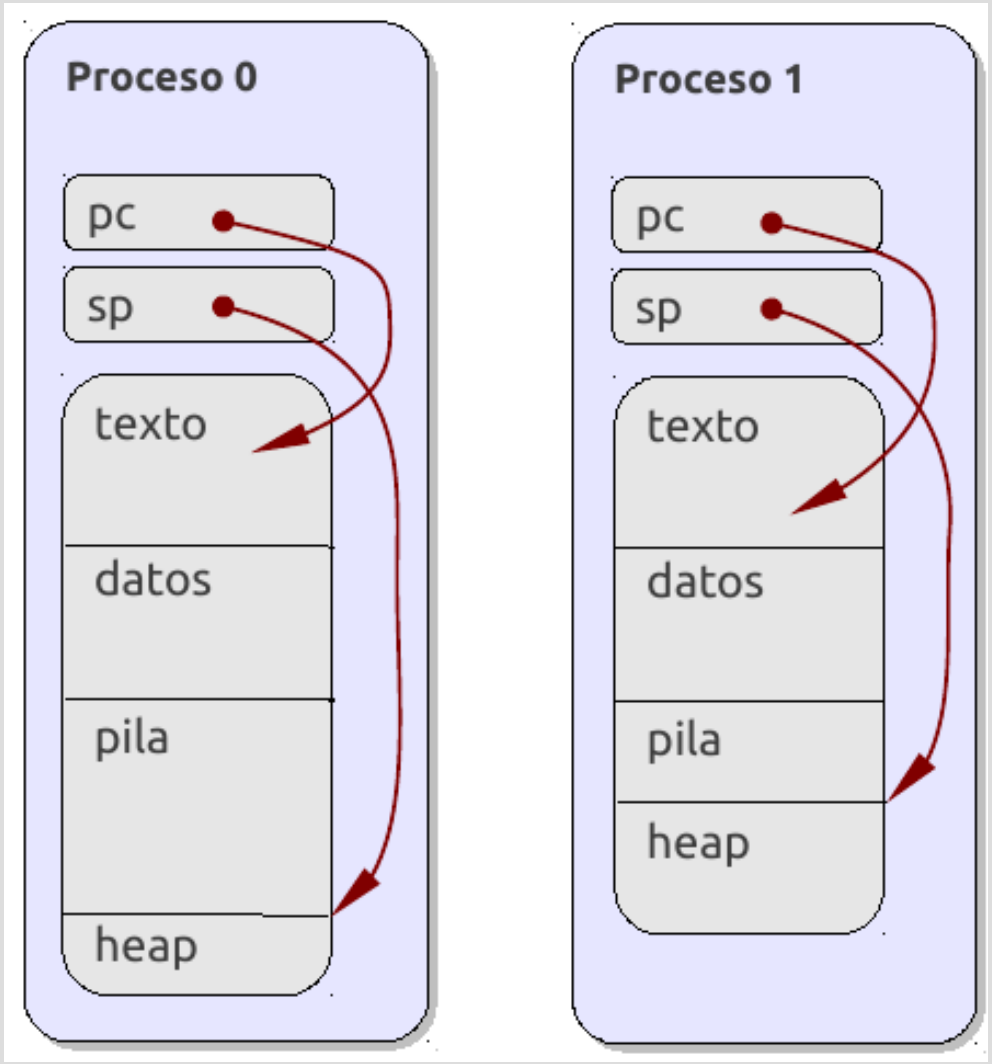
1. Concepto e Implementaciones de Hebras
Procesos: Estructura

Zonas de memoria en un proceso:

- **Texto** (tamaño fijo): Instrucciones programa.
- **Datos** (tamaño fijo): variables globales.
- **Heap** (mem. dinámica): variables dinámicas

Datos asociados

- **Contador de Programa (pc)**: dirección mem. (en texto) de la siguiente instrucción a ejecutar.
- **Puntero de pila (sp)**: dirección última posición ocupada por la pila.

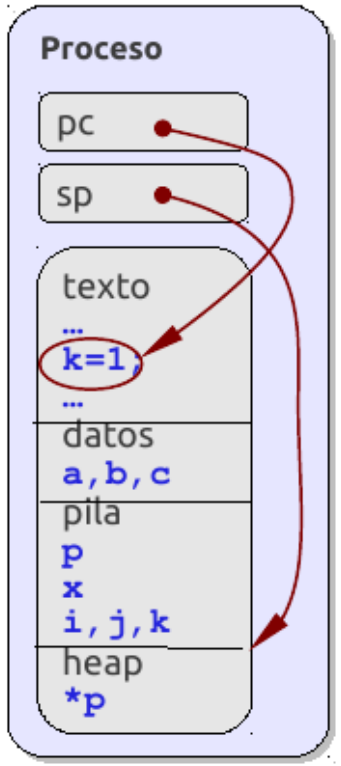


1. Concepto e Implementaciones de Hebras
Ejemplo de proceso en ejecución

Estado de un proceso al ejecutar k=1

```
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() ;
}
```



1. Concepto e Implementaciones de Hebras
Motivación del uso de hebras

Gestión de varios procesos cooperantes es muy útil pero **consume mucho** tiempo y memoria del SO: reparto CPU, datos y comunicaciones entre procesos.

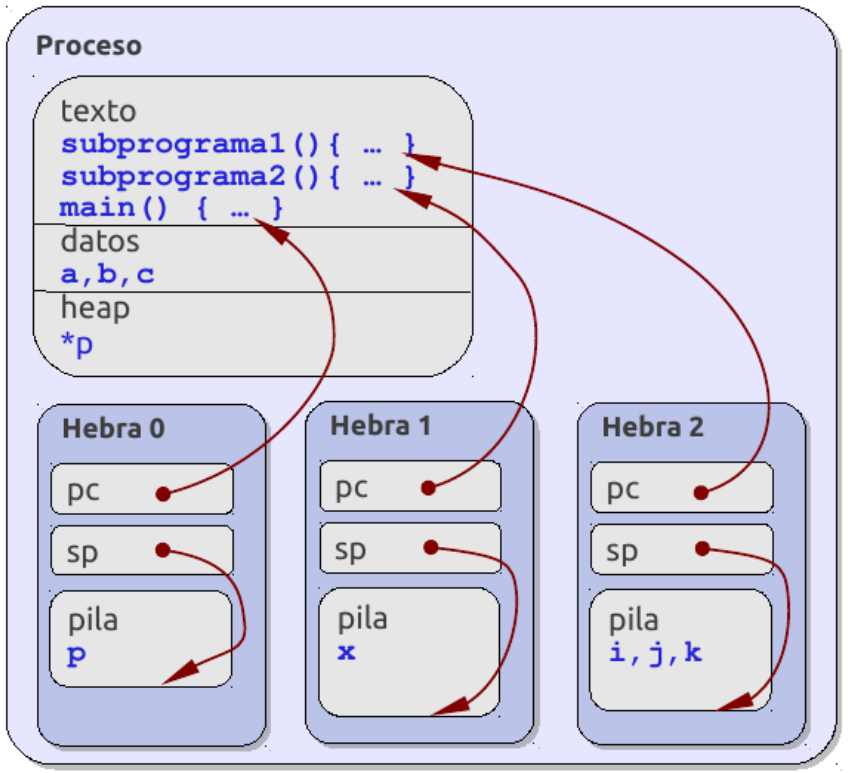
Uso de hebras permite mejorar eficiencia:

- **1 proceso** puede contener 1 o **varias hebras**.
- **Hebra: flujo de control** en texto del proceso.
- Cada hebra tiene su **propia pila**, vacía al inicio.
- Hebras del mismo proceso **comparten datos** + **heap**.

```
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)
 // ...
}
```



1. Concepto e Implementaciones de Hebras
Inicio y finalización de hebras

Inicio ejecución programa, existe una **única hebra** (ejecutaría main en C/C++).

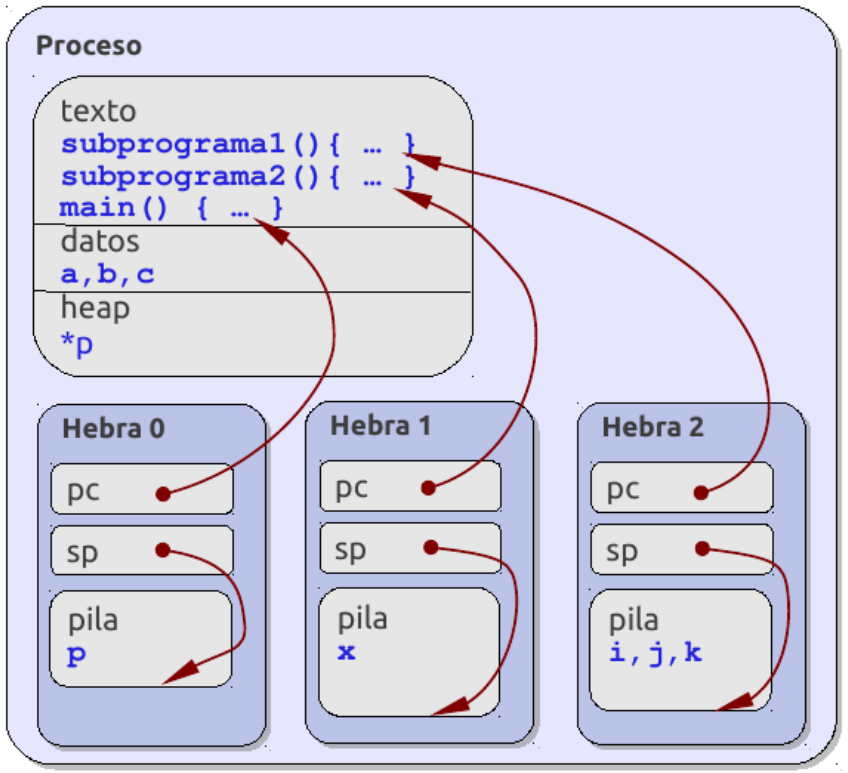
Una **hebra A en ejecución puede crear otra hebra B** en el mismo proceso de A:
- A **designa una función del texto** (y opcionalmente sus parámetros), y continúa su ejecución.

- **Hebra B**: ejecuta f concurrentemente con resto de hebras.
- **Termina** normalmente cuando finaliza ejecución de f.
- Una hebra podría **esperar** a que finalice **otra** hebra.

```
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)
 // ...
}
```



2. Hebras en C++11

2. Hebras en C++11
Introducción

Seguiremos el estándar C++11:

- C++11: versión del lenguaje de programación C++ publicada por ISO en Septiembre de 2011. https://www.iso.org/standard/50372.html
- Incluye funcionalidad para gestión de hebras: tipos de datos, clases y funciones.
- Los fuentes C++ que usan este estándard son portables a Linux,macOS y Windows.

Funcionalidad que veremos:

- Crear nueva hebra concurrente, y esperar su finalización.
- Declaración de variables de tipos atómicos.
- Sincronización de hebras para exclusión mutua o sobre condiciones.
- Bloqueo de una hebra durante intervalo de tiempo, o hasta instante de tiempo.
- Generación de números aleatorios.
- Medición tiempos reales y de proceso, con alta precisión.

2. Hebras en C++11
Creación de hebras

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

- En **declaración de la variable**, se indica el nombre de la función a ejecutar y se comienza ejecución concurrente de dicha función por una nueva hebra.

- En declaración se pueden especificar **parámetros**.

- La variable thread permite referenciar la hebra posteriormente.

ejemplo01.cpp

```
#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 ....
}
```

2. Hebras en C++11
Declaración e inicio separados

Ejemplo anterior: Hebras **se lanzan al declarar** las variables.

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

Posible **declarar las variables y después lanzar** las hebras → En la declaración no incluimos la función:

```
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
```

2. Hebras en C++11
Compilación con línea de órdenes

Compilación y enlace

```
g++ -std=c++11 -o ejecutable -lpthread fuente1.cpp fuente2.cpp .. fuenteN.cpp
```

- En algunos entornos, puede que no sea necesario indicar -lpthread, o incluso que sea un error.

Compilación separada y enlace:

```
g++ -std=c++11 -c fuente1.cpp
g++ -std=c++11 -c fuente2.cpp
       ......
g++ -std=c++11 -c fuenteN.cpp
g++ -o ejecutable fuente1.o fuente2.o .... fuenteN.o -lpthread
```

2. Hebras en C++11
Finalización de Hebras

Una hebra A, ejecutando función f, finaliza cuando:

- Llega al final f.
- Ejecuta return en f.
- Se lanza excepción que no se captura en f ni en ninguna función llamada desde f.
- Se destruye variable asociada (a evitar).

Todas las hebras de un programa finalizan cuando:

- Cualquiera de ellas llama a la función exit() (o abort, o terminate) provocando también la terminación del proceso.
- La hebra principal termina de ejecutar main (Error a evitar, visto en ejemplo01.cpp).

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

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

  - Es necesario que la hebra ppal. espera la terminación de hebra1 y hebra2.

2. Hebras en C++11
Operación de unión

C++11 provee diversos **mecanismos de sincronización: unión,** mutex y variables condición.

Operación de unión (join):

- Permite que una **hebra A espere a que otra hebra B termine**.
- **Hebra A invoca** la unión, y B es la hebra objetivo.
- **Al finalizar** llamada, hebra **B ha terminado** con seguridad.
- Si B ya ha terminado, no pasa nada pero si la espera es necesaria, la **hebra A queda suspendida** sin consumir CPU hasta que B termina.
- La hebra A debe **invocar el método join** sobre la variable thread asociada a B.
- **Hebra B debe ser una hebra activa**:
  - está en ejecución o
  - Finalizada pero no se ha invocado join sobre ella.

2. Hebras en C++11
Operación de unión. Ejemplo

ejemplo02.cpp

```
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
}
```

2. Hebras en C++11
Parámetros y resultados de una hebra

Hasta ahora solo hemos visto hebras que ejecutan funciones sin parámetros.

- Se pueden usar **funciones con parámetros**: Se deben de **especificar los valores de los parámetros al iniciar** la hebra.

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

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

  - Con declaración e inicio separados:

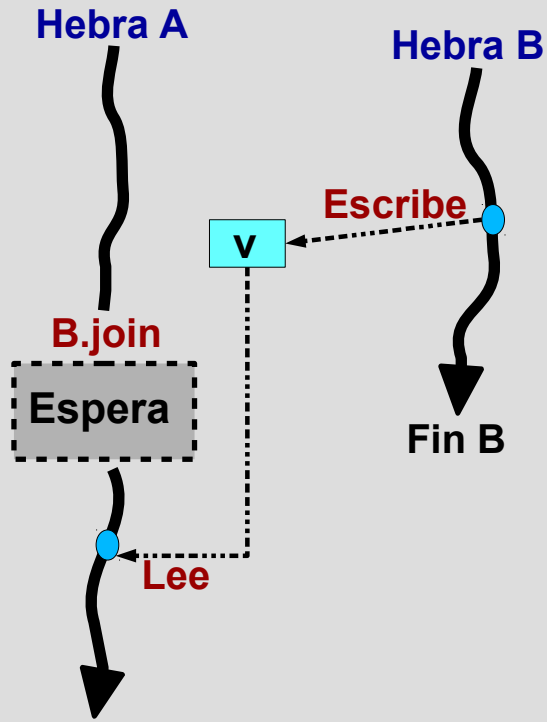
```
thread hebra1, hebra2 ;
...
hebra1 = thread( funcion_hebra_1, 3+2, 45.678 );   // a = 5, x = 45.678
hebra2 = thread( funcion_hebra_2, "hola!", true ); // p = "hola", b = true
```

- Si la función devuelve un valor de un tipo distinto de void, dicho valor es ignorado cuando se hace join.

2. Hebras en C++11
Obtención de valores resultados

Opciones para leer resultado de hebra B que ejecuta función f desde una hebra A:

- **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)**. f escribe en ese parámetro. Hebra A lee resultado tras hacer join.
  - También produce un efecto lateral.
- **Mediante futuros**: la función f devuelve el valor resultado mediante **return**. La hebra A inicia B mediante la **función async** que devuelve un **objeto de tipo future.** El objeto "future" permitirá recuperar el valor resultado cuando la hebra haya terminado.
  - Opción más simple y legible, sin efectos laterales.



2. Hebras en C++11
Uso de variables globales

Queremos que 2 hebras **calculen concurrentemente el factorial de 2 números**, usando **función factorial**:

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

ejemplo03.cpp

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

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

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

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

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

2. Hebras en C++11
Uso de un parámetro de salida

ejemplo04.cpp

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

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

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

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

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

2. Hebras en C++11
Obtención de valores resultado mediante futuros

La función que ejecuta la hebra devuelve el resultado usando return.

- **Lanzamiento de hebra:** mediante una **llamada a la función async**, especificando:
  - **Modo:** constante que especifica que se lanzará una hebra específica para ejecutar la función.
  - **Nombre de la función y sus parámetros**.
- **Obtención del resultado**: async devuelve **un objeto de tipo future** con un **método get** que permite, tras la terminación de la hebra, leer el resultado generado.

ejemplo05.cpp

```
#include <future>      // declaracion de std::thread, std::async, std::future

// iniciar las hebras y obtener los objetos future (conteniendo un long)
// (el valor 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 ;
```

2. Hebras en C++11
Vectores de Hebras/Futuros

Frecuente: Diferentes hebras ejecutan la misma función pero sobre distintos datos

- Cada hebra debe recibir **parámetros distintos**.
- Cada hebra suele recibir un **número de orden** (un entero) que lo identifica.
- Se puede usar un **vector de variables de tipo thread (o future)**.

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() ;
```

2. Hebras en C++11
Ejemplo de Vector de Futuros

- Usamos un **vector de futuros para resolver el problema anterior**
- Hebra principal imprime secuencialmente los resultados.

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 ;
}
```

2. Hebras en C++11
Medición de tiempos en C++11

Objetivo: **Medir duración** de un intervalo de tiempo en una fase de ejecución del programa.

Las mediciones se basan en **2 tipos de datos** (en std::chrono):

- **Instantes de tiempo (time_point):** representa tiempo desde un instante de inicio con un reloj concreto.
- **Duraciones de intervalos de tiempo (duration)**:duración es la diferencia entre dos instantes de tiempo.
  - Puede representarse con enteros o flotantes, y en cualquier unidad de tiempo (nanosegundos, milisegundos, segundos, minutos, etc...).

En C++11 se definen **tres clases para tres relojes distintos**:

- **Reloj del sistema**: (tipo **system_clock**). Tiempo indicado por la hora/fecha del sistema.
- **Reloj monotónico**: (tipo **steady_clock**). Mide tiempo real desde un instante en el pasado. No sufre saltos, por lo que será el que usemos.
- **Reloj de alta precisión**: (tipo **high_precision_clock**). reloj de máxima precisión en el sistema. Podría ser cualquiera de los dos anteriores.

2. Hebras en C++11
Medición de tiempos con steady_clock

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 ;
}
```

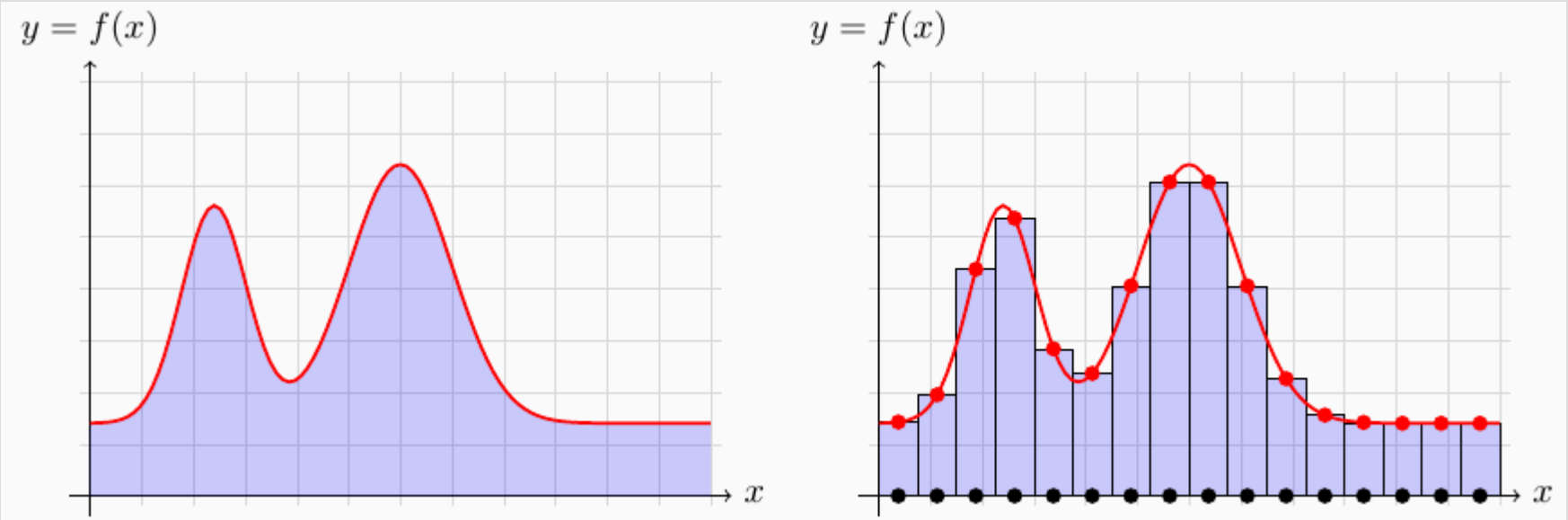
2. Hebras en C++11
Ejemplo. Cálculo numérico de integrales

La programación concurrente puede ser usada para **acelerar la resolución** de multitud de problemas, entre ellos algoritmos numéricos.

Ejemplo típico: Cálculo de la integral I de una función f de variable real.

- **Cuadratura numérica**: Evaluar la función f en un conjunto de m puntos uniformemente espaciados en el intervalo [0, 1], y aproximar I como la media de todos esos valores:

$$I = \int_0^1 f(x)\,dx \qquad I \approx \frac{1}{m}\sum_{i=0}^{m-1} f(x_i) \quad \text{donde:} \quad x_i = \frac{i+1/2}{m}$$



2. Hebras en C++11

Ejemplo. Aproximación del número π

Usaremos una integral de valor conocido:

$$I = \pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

Implementación Secuencial

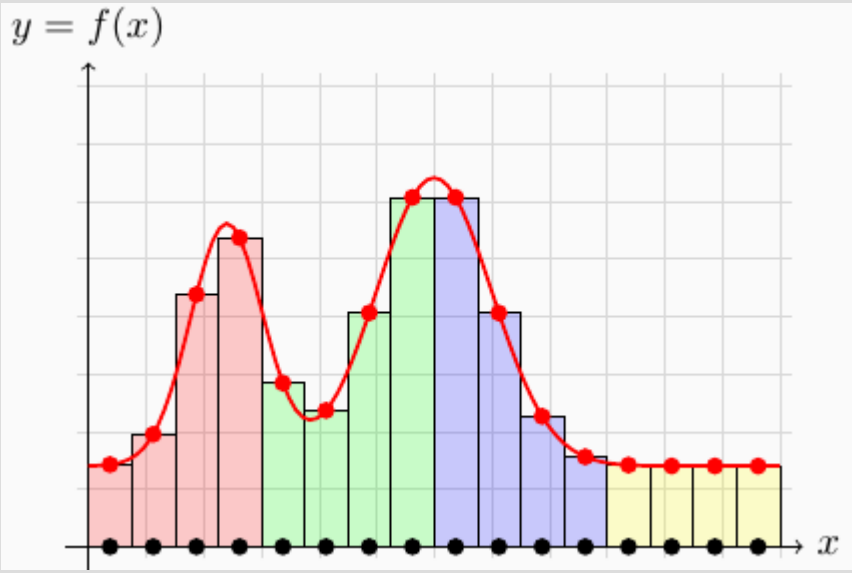
```
const long m = ..., n = ...; // el valor m es alto (del orden de millones)
// implementa función f
double f( double x )
{ return 4.0/(1+x*x) ;      //    f(x) = 4/(1+x²)
}
// calcula la integral de forma secuencial, devuelve resultado:
double calcular_integral_secuencial( long m ) // m == núm. muestras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

2. Hebras en C++11
Ejemplo. Versión concurrente. Estrategia de Reparto

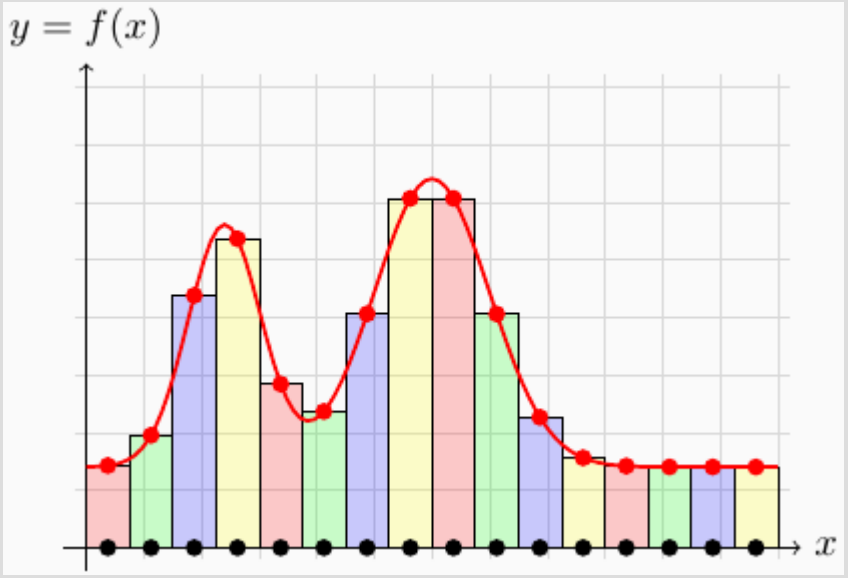
El cálculo se puede **repartir entre n hebras idénticas** (asumimos que m es múltiplo de n):

- Cada hebra evalúa f en m/n puntos del dominio y calcula la suma parcial de los valores de f → Cálculos independientes.
- La **hebra principal recoge** las sumas parciales y calcula la suma total.
- **Con k procesadores** → hasta k veces más rápido si m es mucho mayor que n.

Asignación por bloques contiguos



Asignación cíclica



2. Hebras en C++11
Ejemplo. Implementación concurrente

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

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

- Una función debe lanzar n hebras (con async), y crea un vector de future. La hebra principal espera que vayan acabando, obtiene sumas parciales y devuelve la suma total:

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

- La función main tiene la forma mostrada en archivo ejemplo09-plantilla.cpp:

```
...
const double pi_sec  = calcular_integral_secuencial( );
...


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

ACTIVIDAD PROPUESTA

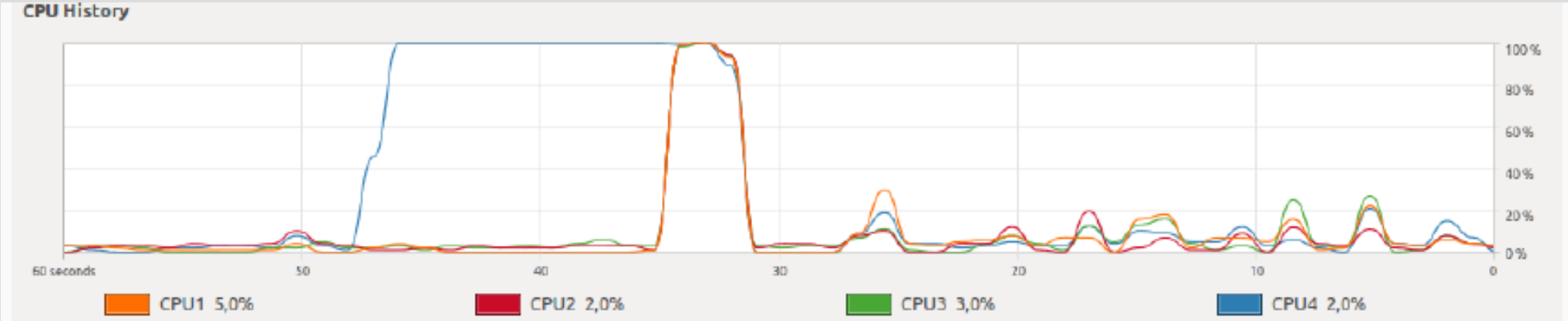
Implementación concurrente del cálculo y medición de tiempos

- **Completar la plantilla en ejemplo09.cpp** con la implementación del cálculo concurrente del número π, tal y como se ha descrito.
- **Salida:** se presentará:
  - El valor exacto de π y el calculado de las dos formas.
  - La **duración del cálculo concurrente**, del secuencial y el porcentaje de tiempo concurrente respecto del secuencial:

```
Número de muestras (m)   : 1073741824
Número de hebras (n)     : 4
Valor de PI              : 3.1415926535897312
Resultado secuencial     : 3.1415926535898185
Resultado concurrente    : 3.1415926535978601
Tiempo secuencial        : 11576 milisegundos.
Tiempo concurrente       : 2990.6 milisegundos.
Porcentaje t.conc/t.sec. : 25.83%
```

ACTIVIDAD PROPUESTA
Resultados esperados

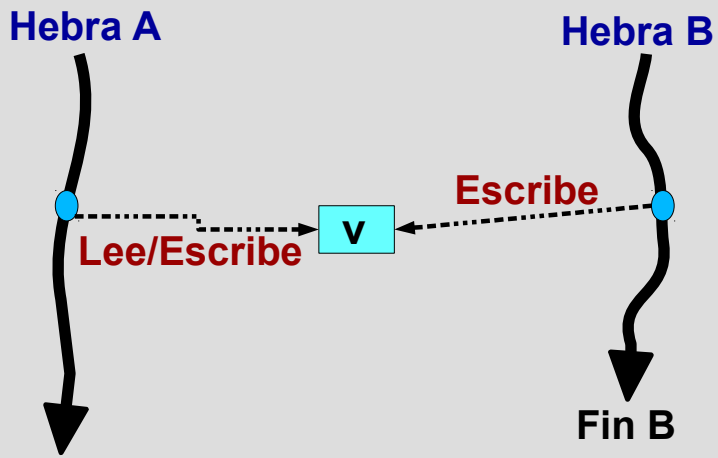
- **Figura**: Captura de **system monitor** mostrando evolución de porcentajes de uso de cada CPU al ejecutar programa con 4 hebras:
- **Parte secuencial**: la hebra principal ejecuta la versión secuencial, y ocupa al 100 % de 1 CPU (CPU4, **línea azul**).
- **Parte concurrente**: 4 hebras creadas por la ppal ocupan cada una CPU al 100 %, la principal espera.
- Cálculo concurrente tardaría algo más de 1/4 del secuencial.



3. Sincronización básica en C++11
Introducción

Veremos algunos **mecanismos básicos** que ofrece C++11 para sincronización de hebras que acceden a variables compartidas asegurando **Exclusion Mutua**:

- **Tipos atómicos**: tipos de datos (típicamente enteros) cuyas variables se pueden actualizar de forma atómica.
  - Ejemplo: al decrementar/incrementar una variable entera por parte de varias hebras.
- **Objetos mutex**: incluyen operaciones que permiten garantizar la EM en la ejecución de trozos de código (secciones críticas)
  - Ejemplo: al realizar dos inserciones concurrentes de nodos en una lista.



Existen **otros tipos de mecanismos** de sincronización en C++11 que veremos más adelante.

3. Sincronización básica en C++11
Tipos atómicos C++11

Para cada tipo entero posible T (char, int, long, unsigned, etc.) existe un correspondiente tipo atómico: **atomic<T>** o **atomic_T**.

- Las operaciones se suelen implementar con **instrucciones hardware atómica**s específicas.
- Para una variable atómica k, las siguientes operaciones se hacen de forma atómica: **k=expresion**; **k++**; **k--**; **k+=expresion**; **k-=expresion**;

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

ejemplo10.cpp

```
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
}
```

3. Sincronización básica en C++11
Tipos atómicos C++11. Ejemplo

```
// 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();
```

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

3. Sincronización básica en C++11
Objetos Mutex

Operaciones complejas sobre estructuras de datos compartidas → **Exclusion Mutua** en trozos de código llamados **Secciones críticas**

Se pueden usar un **objeto mutex o cerrojo** (Locks) para cada sección crítica **(tipo std::mutex)** diferente. Un objeto mutex soporta dos operaciones:

- **lock**: al inicio de SC. La hebra espera si ya hay otra ejecutando dicha SC. Si hay espera, la hebra queda bloqueada sin gastar CPU.
- **Unlock**: al final de la SC para indicar que se ha terminado de ejecutarla.

Entre lock y unlock, la hebra ha adquirido el mutex. El método **lock permite adquirir el mutex**, y **unlock permite liberarlo**. Un mutex está libre o adquirido por una única hebra.

ejemplo12.cpp

```
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
}
```

3. Sincronización básica en C++11
Objetos Mutex. Comparación con tipos atómicos

En **ejemplo11.cpp** se comparan los tiempos de cálculo del ejemplo del contador, pero ahora usando también objetos mutex. Se obtienen estos resultados:

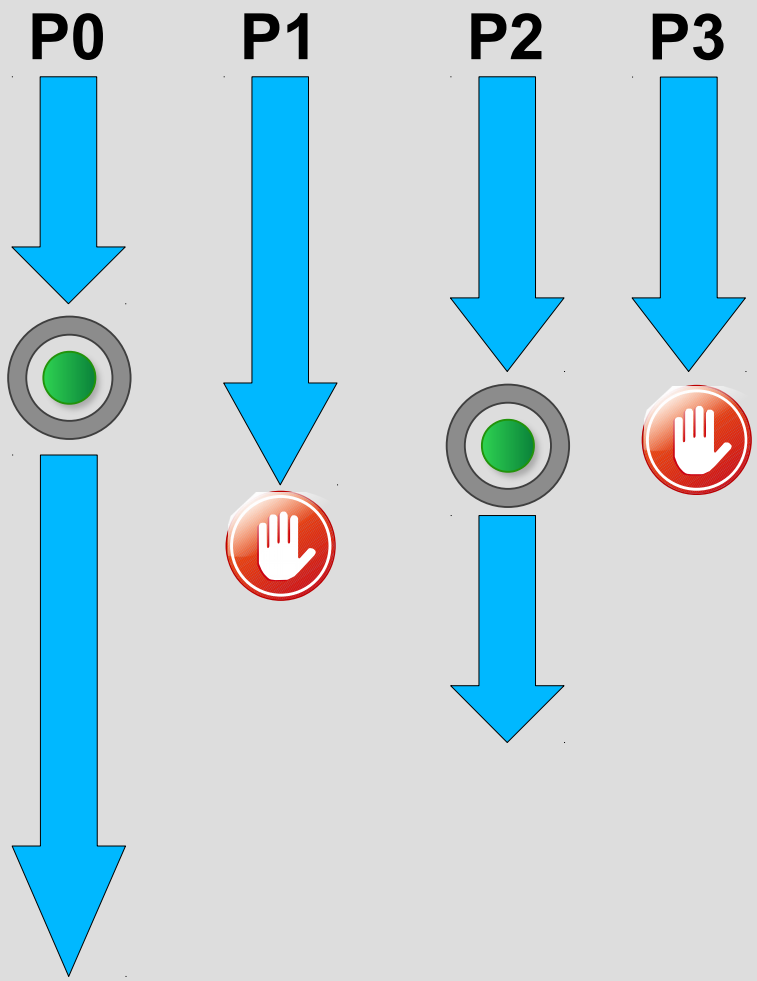
```
valor esperado        : 2000000
resultado (mutex)     : 2000000
resultado (atom.)     : 2000000
resultado (no atom.) : 1222377
tiempo mutex          : 7001.01 milisegundos
tiempo atom.          : 39.5807 milisegundos.
tiempo no atom.       : 7.67227 millisegundos.
```

El **tiempo para objetos mutex es mucho mayor** que con instrucciones atómicas.

4. Introducción a los Semáforos
Concepto

Semáforos: mecanismo que aminora problemas de soluciones de bajo nivel, y tiene un ámbito de uso más amplio.

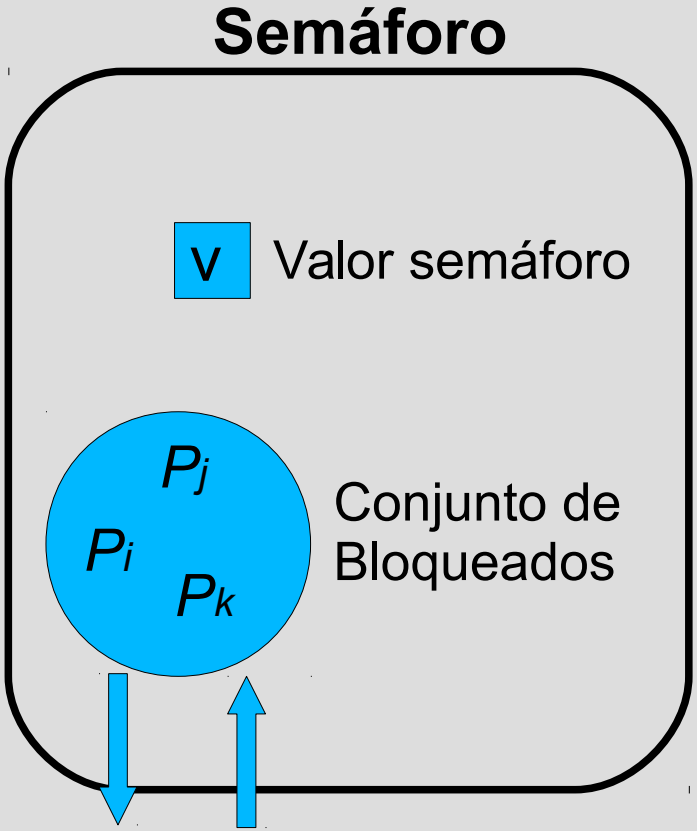
- No se usa espera ocupada, sino **bloqueo de procesos** (Más eficiente).
- Resuelven fácilmente la **Exclusión Mutua**.
- Permiten resolver cualquier **problema de sincronización** (aunque los esquemas de uso pueden ser complejos).
- Se implementa mediante instancias de una **estructura de datos** accesible únicamente mediante **subprogramas específicos** ⇒ aumenta la seguridad y simplicidad.



4. Introducción a los Semáforos
Estructura de un semáforo

Un semáforo es un **instancia de una estructura de datos** que contiene los siguientes elementos en memoria compartida:

- **Conjunto de procesos bloqueados** (“esperando al semáforo”).
- **Valor del semáforo:** valor entero no negativo.



Al **inicio de un programa** que usa un semáforo, debe poder inicializarse:

- Su **conjunto** de procesos esperando estará **vacío**
- Se debe indicar un **valor inicial** del semáforo

4. Introducción a los Semáforos
Operaciones sobre un semáforo

Además de inicialización, solo hay dos operaciones:

sem_wait(s)

```
If (s.valor ==0)
        Bloquea proceso
s.valor = s.valor - 1;
```

sem_signal(s)

```
s.valor = s.valor + 1;
If (hay procesos esperando s)
        reanuda uno;
```

- **s.valor nunca es negativo**, ya que antes de decrementar se espera a que sea 1.
- Solo puede haber procesos esperando s cuando **s.valor** es 0.
- Estas operaciones se ejecutan **en exclusión mutua** sobre cada semáforo (excluyendo periodo bloqueo wait),

4. Introducción a los Semáforos
Invariante de un semáforo

- Dado un **semáforo *s***, que se **inicializó a v*0***, con valor *vt* en un instante de tiempo, se verifica:

$$v_t = v_0 + n_s - n_w \geq 0$$

donde:

  - n_s es el número de llamadas a sem_signal completadas y
  - n_w es el número de llamadas a sem_wait completadas.

- Los 4 valores son **enteros no negativos**.
- La **igualdad** se deriva de que sem_signal siempre incrementa el valor y sem_wait siempre lo decrementa (pero espera antes cuando es 0).
- Se mantiene cuando no se está ejecutando sem_wait o sem_signal.
- Solo **cuentan llamadas a sem_wait completadas totalmente** en t.

4. Introducción a los Semáforos
Patrones de uso sencillos

- **Espera única**: Un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (típicamente cuando un proceso debe leer una variable escrita por otro proceso).

- **Exclusión Mutua**: acceso a una sección crítica por parte de un número arbitrario de procesos.

- **Productor/Consumidor**: un proceso escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso.

4. Introducción a los Semáforos
Espera Única. problema

```
{ variables compartidas y valores iniciales }
var x              : integer ;          { variable escrita por Productor }
    puede_leer     : semaphore := 0 ; { 1 si x ya escrita y aun no leída}
```

```
process Productor ; { escribe 'x' }
   var a : integer ;
begin
   a := ProducirValor() ;
   x := a ; { sentencia E }
end
```

```
process Consumidor  { lee 'x' }
   var b : integer ;
begin
   b := x ; { sentencia L }
   UsarValor(b) ;
end
```

- Sentencias E y L son atómicas.
- Correctas interfoliaciones en las que E antes que L.

4. Introducción a los Semáforos
Espera Única. Solución con semáforo

```
{ variables compartidas y valores iniciales }
var x           : integer ;       { variable escrita por Productor }
    puede_leer : semaphore := 0; { 1 si x ya escrita y aun no leída }
```

```
process Productor ; { escribe 'x' }
   var a : integer ;
begin
   a := ProducirValor() ;
   x := a ; { sentencia E }
   sem_signal( puede_leer ) ;
end
```

```
process Consumidor  { lee 'x' }
   var b : integer ;
begin
   sem_wait( puede_leer ) ;
   b := x ; { sentencia L }
   UsarValor(b) ;
end
```

4. Introducción a los Semáforos
Exclusión Mutua. Solución con semáforo

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { 1 si SC está libre, 0 si SC ocupada }
                                { (núm. de procs. que pueden entrar a SC) }

process P[ i : 0..n ];
begin
  while true do begin

   { esperar hasta que sc_libre sea 1, entonces ponerla a 0 }
   sem_wait( sc_libre ) ;

   { seccion critica: ..... }

   { poner sc_libre a 1, y desbloquear un proceso en espera si hay alguno}
   sem_signal( sc_libre ) ;

   { resto seccion: ..... }
 end
end
```

4. Introducción a los Semáforos
Productor-Consumidor. Problema

```
{ variables compartidas y valores iniciales }
var x : integer ; { contiene cada valor producido y pte. de leer }
```

```
process Productor ; { calcula x }
var a : integer ;
begin
   while true do begin
      a := ProducirValor() ;
      x := a ; { sentencia E }
   end
end
```

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

- Son correctas las interfoliaciones en las que E y L se alternan, comenzando en E.

4. Introducción a los Semáforos
Productor-Consumidor. Solución con semáforos

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

```
process Productor ; { escribe x }
var a : integer ;
begin
   while true do begin
      a := ProducirValor() ;
      sem_wait( puede_escribir  );
      x := a ; { sentencia 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 ; { sentencia L }
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

5. Semáforos en C++11
Tipos de datos y operaciones

Se ha diseñado un **tipo de datos en C++11** con la funcionalidad de los semáforos:

- Tipo **Semaphore** con 2 operaciones: **sem_wait** y **sem_signal**.
- **Inicialización**, obligatoriamente en la declaración
- Variables Semaphore se pueden pasar como parámetros pero no se pueden copiar mediante asignaciones. Se destruyen automáticamente.

Definición

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

Uso

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

- Generalmente se declaran como **variables globales** compartidas entre las hebras que los usan.

5. Semáforos en C++11
Programas con semáforos

- Necesario hacer **#include** y **using** en la cabecera:

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

Compilación y enlace

- Se debe de disponer de los archivos **Semaphore.h y Semaphore.cpp en el directorio de trabajo**.
- Se debe de **compilar y enlazar el archivo Semaphore.cpp**, junto con los fuentes que usan los semáforos:

```
g++ -std=c++11 -I. -o ejecutable fuente1.cpp Semaphore.cpp -lpthread
```

5. Semáforos en C++11
Productor-Consumidor simple. Cabecera

ejemplo13-s.cpp

```
#include <iostream>
#include <thread>
#include "Semaphore.hpp" // incluye tipo 'Semaphore'

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


.....
```

5. Semáforos en C++11
Funciones de Hebra productora y consumidora

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

```
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 ) ;
   }
}
```

```
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 );
   }
}
```

5. Semáforos en C++11
Hebra principal

```
.....
// 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();
}
```