



UNIVERSIDAD  
DE GRANADA

# Sistemas Concurrentes y Distribuidos:

## Práctica 4. Implementación de Sistemas de Tiempo Real.

---

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

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

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

## Práctica 4. Implementación de Sistemas de Tiempo Real. Índice.

1. Relojes, instantes y duraciones.
2. Implementación de un ejecutivo cíclico.

# Objetivos

Los objetivos de esta práctica son:

- ▶ Conocer las características de C++11 que ayudan a la programación de sistemas de tiempo real.
- ▶ Aprender a implementar en C++11 un sistema de restricciones temporales usando planificación estática *offline* (ejecutivo cíclico)

En este documento, primero se hace una breve introducción a los relojes en C++11, las duraciones y los instantes de tiempo. Después veremos como se puede implementar un ejecutivo cíclico sencillo, y finalmente se propone implementar uno al alumno.

## Sección 1. Relojes, instantes y duraciones..

- 1.1. Relojes
- 1.2. Instantes y duraciones
- 1.3. Esperas bloqueadas

Sistemas Concurrentes y Distribuidos, curso 2024-25.  
Práctica 4. Implementación de Sistemas de Tiempo Real.  
Sección 1. Relojes, instantes y duraciones.

Subsección 1.1.

Relojes.

Usaremos las características de C++11 para poder hacer implementaciones sencillas y portables a diferentes SS.OO. Las características son:

- ▶ Varias clases que representan relojes, incluyendo el reloj del sistema (no monotónico) y relojes monotónicos y/o de alta precisión.
- ▶ Tipos de datos para instantes de tiempo absolutos y para duraciones de tiempo. Una duración es la diferencia entre dos instantes de tiempo absolutos.
- ▶ Posibilidad de dejar una hebra bloqueada durante un intervalo de tiempo o bien hasta un instante de tiempo.

# Atributos de los relojes

Un reloj es una abstracción que usamos para **medir el tiempo real** transcurrido desde un determinado instante de inicio. Los relojes en C++11 son **clases portables**, que constituyen abstracciones sobre los relojes de bajo nivel proporcionados por el S.O. subyacente, que funcionan produciendo *ticks* a intervalos regulares de tiempo. Los relojes C++11 se caracterizan por estos atributos:

- ▶ **Período** (precisión): **mínimo intervalo de tiempo que es capaz de medir**, o equivalentemente, intervalo de tiempo entre dos *ticks* consecutivos del reloj.
- ▶ **Época**: instante de inicio absoluto del reloj, es decir, el **instante en el que el reloj empezó a contar ticks**. Los tiempos proporcionados por el reloj están referidos al instante de inicio.
- ▶ **Monotonidad**: un reloj es **monotónico** (o **sostenido**) cuando podemos asegurar que una lectura posterior a otra siempre proporciona un tiempo mayor al de la primera.

# Relojes concretos ofrecidos en C++11

Los tres relojes existentes son estas tres clases C++ (en el *namespace* **std::chrono**)

- ▶ **system\_clock**: reloj del sistema, no es monotónico (ya que puede ser atrasado), su época es conocida para el SS.OO, y normalmente estándar (suele ser el 1 de enero de 1970 a las 0 horas UTC)). Por tanto, sus medidas se pueden interpretar como fechas y horas concretas.
- ▶ **steady\_clock**: reloj monotónico (se garantiza que nunca atrasa). Su época puede ser cualquier instante previo al inicio del proceso.
- ▶ **high\_precision\_clock**: es el reloj con la máxima precisión del sistema, puede ser equivalente a cualquiera de los dos anteriores o a otro. Su época puede ser cualquiera.

El programa C++ **relojes.cpp** muestra los tres atributos de cada uno de los tres relojes.



# Características de los relojes C++11 (mac OS)

Probamos en mac OS High Sierra (con clang++), con este resultado:

```
-----  
Características del reloj: system_clock
```

```
-----  
Período (precisión)                = 1000 nanosegundos.  
Tiempo desde el inicio de la época = 420091 horas.  
Es un reloj monotónico             = no  
-----
```

```
Características del reloj: steady_clock
```

```
-----  
Período (precisión)                = 1 nanosegundos.  
Tiempo desde el inicio de la época = 70.7731 horas.  
Es un reloj monotónico             = sí  
-----
```

```
Características del reloj: high_resolution_clock
```

```
-----  
Período (precisión)                = 1 nanosegundos.  
Tiempo desde el inicio de la época = 70.7731 horas.  
Es un reloj monotónico             = sí  
-----
```

Vemos que el reloj de alta precisión coincide con el monotónico.

# Características de los relojes (Ubuntu 17 64 bits)

En Ubuntu 17 de 64 bits (con clang++ o g++), obtenemos:

```
-----  
Características del reloj: system_clock
```

```
-----  
Período (precisión)                = 1 nanosegundos.  
Tiempo desde el inicio de la época = 420090 horas.  
Es un reloj monotónico             = no
```

```
-----  
Características del reloj: steady_clock
```

```
-----  
Período (precisión)                = 1 nanosegundos.  
Tiempo desde el inicio de la época = 272.649 horas.  
Es un reloj monotónico             = sí
```

```
-----  
Características del reloj: high_resolution_clock
```

```
-----  
Período (precisión)                = 1 nanosegundos.  
Tiempo desde el inicio de la época = 420090 horas.  
Es un reloj monotónico             = no
```

Ahora, el reloj de alta precisión coincide con el del sistema.

# Selección de un reloj

Para implementar el ejecutivo cíclico, usaremos el reloj monotónico (**steady\_clock**), ya que:

- ▶ Su resolución es más que suficiente para nuestro propósito en los SS.OO. en los que lo hemos probado.
- ▶ No podemos usar el reloj del sistema, al no ser monotónico: habría errores al cambiar la hora del sistema.
- ▶ En algunos SS.OO., el reloj de alta precisión puede no ser monotónico (es el caso de mac OS), si lo usamos, el programa podría no ser portable.

En C++11 los valores de duraciones e instantes tienen siempre asociado un reloj, ya que solo podemos operar dos de esos valores si su época y precisión coinciden. Por tanto, usaremos instantes y duraciones relativos a **steady\_clock** en todos los casos.

Sistemas Concurrentes y Distribuidos, curso 2024-25.  
Práctica 4. Implementación de Sistemas de Tiempo Real.  
Sección 1. Relojes, instantes y duraciones.

## Subsección 1.2. Instantes y duraciones.

# Tipos para instantes de tiempo

En C++11, un instante de tiempo absoluto se representa como la duración del intervalo de tiempo transcurrido desde la época de un reloj (es un punto en la línea de tiempo del reloj)

- ▶ El tipo de datos C++11 se llama `time_point<reloj>` (usaremos concretamente el tipo `time_point<steady_clock>`).
- ▶ Para medir el tiempo actual de un reloj podemos usar el método `now` de la clase del reloj, devuelve el instante de tiempo del momento de la llamada, por ejemplo:

```
// declara la variable ahora y la inicializa con el instante actual  
// (requiere using namespace std::chrono)  
time_point<steady_clock> ahora = steady_clock::now();
```

- ▶ Los instantes de tiempo relativos a un mismo reloj son comparables entre sí (podemos saber cual es anterior y cual es posterior con los operadores de comparación de C++).

# Tipos para duraciones

Una **duración** es una medida del tiempo que transcurre entre dos instantes.

- ▶ Una variable de un tipo duración contiene un valor numérico, entero o real, interpretable en una unidad de tiempo determinada.
- ▶ El nombre de un tipo de datos para las duraciones es de la forma **duration**<Rep, Per>, donde:
  - ▶ Rep es el tipo de datos usado para representar la duración, puede ser un tipo flotante (**float** o **double**), o bien un tipo entero (**int**, **unsigned**, **long**, etc...)
  - ▶ Per es la duración de una unidad de tiempo, expresada como una fracción de la duración de un segundo (un valor racional  $p/q$  escrito de la forma **ratio**< $p, qp$  y  $q$  son enteros no nulos, tales que  $p/q$  puede ser mayor, menor o igual que la unidad).

## Tipos para duraciones (2)

C++11 incluye varios tipos ya predefinidos para duraciones, y podemos definir otros.

- ▶ Los tipos predefinidos se denominan **nanoseconds**, **microseconds**, **milliseconds**, **seconds**, **minutes** y **hours** (todos ellos representados con enteros). Usaremos **milliseconds** en esta práctica.
- ▶ Es posible definir tipos nuevos, por ejemplo, es útil definir un tipo llamado **milliseconds\_f** para duraciones en milisegundos, pero expresadas en coma flotante (con parte fraccionaria), lo hacemos con:

```
typedef duration<float, ratio<1,1000>> milliseconds_f ;
```

- ▶ El método **count** devuelve un valor de tipo *Rep* (**int**, **float**, etc....) con la medida de una duración **d** (se debe usar para imprimir una duración, o por ejemplo para compararla con un valor).

# Operaciones con duraciones e instantes

Un instante menos otro instante produce una duración (el primero debe ser posterior al segundo, y ambos deben de corresponder al mismo reloj). La duración resultante es de tipo `reloj::duration`, donde `reloj` es el reloj los instantes. En este ejemplo se mide lo que tarda una secuencia de instrucciones cualquiera:

```
time_point<steady_clock> instante1 = steady_clock::now() ;  
//..... sentencias  
time_point<steady_clock> instante2 = steady_clock::now() ;  
// obtenemos el tiempo transcurrido entre instante1 e instante2  
// (es un float, con unidades de milisegundos y con parte fraccionaria)  
float tiempo_ms_f = milliseconds_f( instante2 - instante1 ).count();  
cout << "Duración == " << tiempo_ms_f << " milisegundos." << endl;
```

Un instante más una duración produce un instante posterior al primero (relativo al mismo reloj). En este ejemplo se calcula un instante en el futuro (`instf`), 200 milisegundos después:

```
time_point<steady_clock> instf = steady_clock::now() + milliseconds(200);
```



Sistemas Concurrentes y Distribuidos, curso 2024-25.  
Práctica 4. Implementación de Sistemas de Tiempo Real.  
Sección 1. Relojes, instantes y duraciones.

## Subsección 1.3. Esperas bloqueadas.

# Las funciones `sleep_for` y `sleep_until`

La función `sleep_for` (que ya hemos usado) deja a la hebra que la llama bloqueada un intervalo de tiempo cuya duración mínima la especificamos como parámetro. Por ejemplo:

```
sleep_for( milliseconds(200) ); // duerme durante al menos 200 milisegundos
```

La función `sleep_until` deja a la hebra que la llama bloqueada hasta un instante no anterior al instante final que pasamos como parámetro:

```
time_point<steady_clock> instante_futuro = ..... ; // calcular un instante  
sleep_until( instante_futuro ) ; // duerme hasta después de instante_futuro
```

Ambas funciones están en el namespace `std::this_thread`, es decir, debemos de poner el prefijo `std::this_thread::` o bien hacer `using namespace std::this_thread;`

# Retraso de `sleep_for` y `sleep_until`

Usaremos estas funciones para la implementación de la simulación de un ejecutivo cíclico, pero hay que tener en cuenta que:

- ▶ Nuestras pruebas se hacen en sistemas operativos de propósito general (Linux, macOS, Windows), en los cuales el proceso que llama a estas funciones convive con otros muchos procesos, compartiendo el tiempo de las CPUs disponibles.
- ▶ Como consecuencia, cuando acaba el tiempo especificado para `sleep_for` o `sleep_until`, la hebra no tiene garantizado que pueda reanudar su ejecución de forma inmediata en una CPU.
- ▶ Así que el tiempo que realmente tardan estas funciones podrá ser ligeramente superior al valor exacto especificado.

En el archivo `tiempos.cpp` usamos `now` para medir los tiempos de `sleep_for` y `sleep_until`, en todos los casos se observa un retraso de entre 0 y 5 milisegundos aproximadamente.

## Sección 2. Implementación de un ejecutivo cíclico..

- 2.1. Ejemplo de implementación
- 2.2. Actividades

Sistemas Concurrentes y Distribuidos, curso 2024-25.  
Práctica 4. Implementación de Sistemas de Tiempo Real.  
Sección 2. Implementación de un ejecutivo cíclico.

## Subsección 2.1. Ejemplo de implementación.

# Sistema y planificación de ejemplo

En primer lugar consideramos una implementación de referencia de un ejecutivo cíclico, que ilustra el esquema a seguir. Consideramos el siguiente conjunto de tareas y restricciones temporales (tiempos en unidades de milisegundos)

Tarea	$T$	$C$
$A$	250	100
$B$	250	80
$C$	500	50
$D$	500	40
$E$	1000	20

- ▶ Se asume que  $D_i = T_i$
- ▶ El ciclo principal dura  $T_M = 1000$  ms, ya que

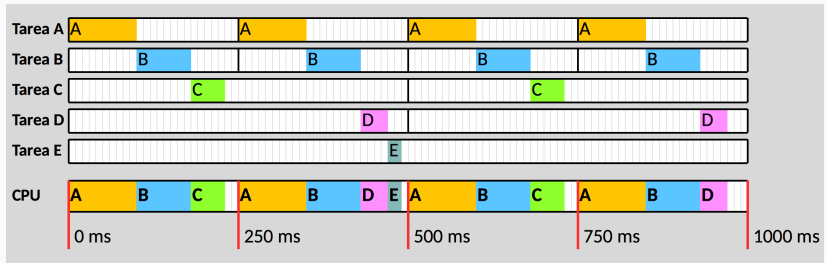
$$T_M = \text{mcm}(250, 250, 500, 500, 1000) = 1000$$

# Planificación del sistema

Diseñamos una planificación manualmente, de forma que

- ▶ Cada tarea se ejecuta una única vez completamente dentro de cada repetición de su período.
- ▶ Si una tarea se inicia dentro de una iteración del ciclo secundario, acaba antes del final de dicha iteración.

Una posible planificación con  $T_S = 250$  ms para este sistema es esta:



# Implementación del ejemplo

Esquema de una posible implementación (en `ejecutivo1.cpp`)

```
int main( int argc, char *argv[] )
{
    // Ts es la duración del ciclo secundario (la damos en milisegundos)
    const milliseconds Ts( 250 );
    // ini_sec es el instante inicial de cada iteración del ciclo secundario
    time_point<steady_clock> ini_sec = steady_clock::now() ;

    while( true ) // ciclo principal
    { for( int i = 1 ; i <= 4 ; i++ ) // ciclo secundario
        { // ejecutamos las tareas correspondientes a esta iteración del c.s.
            switch( i )
            { case 1 : TareaA(); TareaB(); TareaC();           break ;
              case 2 : TareaA(); TareaB(); TareaD(); TareaE(); break ;
              case 3 : TareaA(); TareaB(); TareaC();           break ;
              case 4 : TareaA(); TareaB(); TareaD();           break ;
            }
            ini_sec += Ts ;      // calcular instante inicial para siguiente iteración
            sleep_until( ini_sec ); // esperamos hasta el instante inicial
        }
    }
}
```



Sistemas Concurrentes y Distribuidos, curso 2024-25.  
Práctica 4. Implementación de Sistemas de Tiempo Real.  
Sección 2. Implementación de un ejecutivo cíclico.

Subsección 2.2.

Actividades.

# Actividad 1: nueva funcionalidad

En la simulación (en `ejecutivo1.cpp`) cada tarea es una simple espera bloqueada de duración igual a su tiempo de cómputo. También hay una espera al final del ciclo secundario.

- ▶ Sabemos que, en la práctica, en una ejecución el tiempo de duración actual de cada una de esas esperas puede ser algo mayor que el argumento de `sleep_for`.
- ▶ Copia el código en `ejecutivo1-compr.cpp` y ahí extiéndelo de forma que, cada vez que acaba un ciclo secundario, se informe del retraso del instante final actual respecto al instante final esperado.
- ▶ La comprobación se hará al final del bucle, inmediatamente después de `sleep_until`.

## Actividad 2: nuevo ejemplo

Diseña una planificación para las tareas y restricciones que se indican en esta tabla (tiempos en milisegundos). Copia `ejecutivo1-compr.cpp` en `ejecutivo2.cpp` y en este último implementa la planificación.

Tarea	$T$	$C$
$A$	500	100
$B$	500	150
$C$	1000	200
$D$	2000	240

Responde en tu portafolios a estas cuestiones:

- ▶ ¿cual es el mínimo tiempo de espera que queda al final de las iteraciones del ciclo secundario con tu solución ?
- ▶ ¿ sería planificable si la tarea  $D$  tuviese un tiempo cómputo de 250 ms ?

Fin de la presentación.