9 – Capacidades de Tiempo Real

1. Introducción

La noción del tiempo y su uso en los lenguajes de programación puede estudiarse desde el punto de vista de tres elementos independientes:

1. La interfaz con el tiempo:
   a. Acceso a relojes para medir el paso del tiempo.
   b. Retardo de procesos.
   c. Programación de tiempos límite de espera (*timeouts*).
2. La representación de requisitos temporales:
   a. Especificar tasas de ejecución.
   b. Especificar tiempos límite (*deadlines*).
3. La satisfacción de los requisitos temporales.

Desde un punto de vista matemático, el tiempo se puede considerar como una recta real que tiene las siguientes propiedades topológicas:

- **Lineal**: dados dos instantes de tiempo cualesquiera, uno es anterior a otro, posterior o representan el mismo tiempo.
- **Transitivo**: si tres instantes de tiempo son anteriores dos a dos, lo son el primero y el tercero.
- **Irreflexivo**: un instante de tiempo no puede ser anterior ni posterior a sí mismo.
- **Denso**: si un instante de tiempo es anterior a otro, siempre hay un tercero entre ambos.

- Lineal: $\forall x, y : x < y \vee y < x \vee x = y$
- Transitivo: $\forall x, y, z : (x < y \wedge y < z) \Rightarrow x < z$
- Irreflexivo: $\forall x : not(x < x)$
- Denso: $\forall x, y : x < y \Rightarrow \exists z : (x < z < y)$

2. El concepto de tiempo

Si se dispone de eventos que se comportan de forma regular, se pueden definir en base a ellos distintos **estándares o medidas de tiempo**:

Name	Description	Note
True Solar Day	Time between two successive culminations (highest point of the sun)	Varies through the year by 15 minutes (approx)
Temporal Hour	One-twelfth part of the time between sunrise and sunset	Varies considerably through the year
Universal Time (UT0)	Mean solar time at Greenwich meridian	Defined in 1884
Second (1)	1/86,400 of a mean solar day	
Second(2)	1/31,566,925.9747 of the tropical year for 1900	Ephemris Time defined in 1955

UT1	correction to UTO because of polar motion
UT2	Correction of UT1 because of variation in the speed of rotation of the earth
Seconds(3)	Duration of 9_192_631_770 periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the Caesium - 133 atom
	Accuracy of current Caesium atomic clocks deemed to be one part of 10^13 (that is, one clock error per 300,000 years)
International Atomic Time (IAT)	Based upon Caesium atomic clock
Coordinated Universal Time (UTC)	An IAT clock synchronized to UT2 by the addition of occasional leap ticks
	Maximum difference between UT2 (which is based on astrological measurement) and IAT (which is based upon atomic measurements) is kept to below 0.5 seconds

El **tiempo universal** (Universal Time, UT) se definió como el tiempo solar medio en el meridiano 0 (Greenwich), siendo 1 s = 1/86.400 de un día solar medio. Como era impreciso, se definió el **tiempo de efemérides**, basado en el movimiento

orbital de objetos en el sistema solar, definiéndose de esta manera el año trópico, el período de tiempo para que la longitud de la eclíptica del sol aumente 360 grados (1s = 1/31.566.925,9747 de dicho año trópico).

Se define también el **tiempo atómico universal** (TAI) como un estándar atómico de alta precisión para medir el tiempo propio de un cuerpo geoide con reloj atómico, pero el TAI va lentamente diferenciándose del UT, ya que el día solar medio va aumentando.

A partir de estas dos medidas se define el **UTC (Universal Time Coordinated**), obtenido a partir del TAI, pero añadiendo o quitando un segundo cuando es necesario. UTC es adecuado para la comunicación entre personas, pero presenta algunos saltos que hace que sea inadecuado para medidas de tiempo precisas. Por el contrario, el TAI sigue siendo una medida más adecuada para controlar la ejecución de tareas en tiempo real.

Referencia de tiempo para un STR

En un entorno de un STR, las referencias de tiempo deben cumplir las siguientes características:

- **Estable**: sin presentar grandes variaciones a lo largo del tiempo.
- **Exacta**: con diferencias con IAT acotadas.
- **Precisa**: la diferencia entre dos lectoras debe estar acotada.
- **Monótona no decreciente**: sin presentar saltos hacia atrás.

3. Relojes

Para medir el paso del tiempo en los programas hay dos posibilidades:

- Accediendo al marco temporal del entorno. Implica hacer uso de algún recurso externo a la máquina, como la hora local geográfica.
- Mediante un reloj hardware interno que dé una aproximación adecuada del paso del tiempo en el entorno. En este caso, es necesario hacer sincronizaciones periódicas, ya que se pueden producir desfases entre la hora interna que se está usando a partir del hardware y la hora real (*clock drift*). Aunque se hable de reloj hardware interno y se tenga una electrónica que genera eventos periódicos, se necesita un software que proporciona un valor de tiempo real cuando se lea, convirtiéndolo en unidades de tiempo

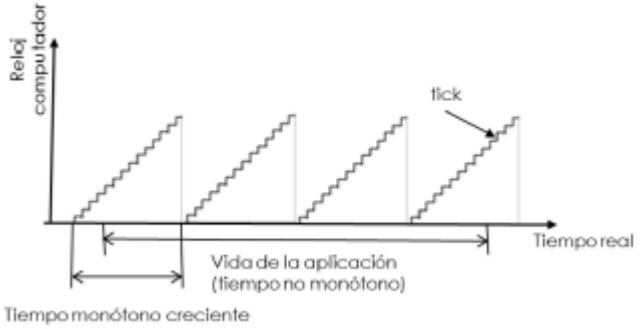
Características de un reloj

Las características más importantes de un reloj son:

- **Resolución**: corresponde a la unidad mínima representable por el reloj, es decir, las unidades que se representan en el contador de pulsos (por ejemplo, 1 ns). Es una característica estática.
- **Intervalo de valores**: es el límite inferior y superior del tiempo medible, que corresponde a la capacidad máxima del contador (Tr_{max}). También es una característica estática.
- **Precisión o exactitud**: permite saber la diferencia con respecto a un reloj de referencia, por lo que se pueden tener errores o derivas con respecto a la referencia de tiempo externo, causados por variaciones de temperaturas u otros efectos. Es una característica dinámica.
- **Granularidad**: es el período del oscilador (*tick*), que corresponde a la distancia en tiempo real entre dos tiempos de reloj consecutivos (Ejemplo: µs/ms). Es una característica dinámica.
- **Estabilidad**: permite saber las derivas en la frecuencia de progreso del reloj respecto a una referencia externa, es decir, la variación con el tiempo.

Los relojes internos tienen un contador que va acumulando la cuenta de los eventos periódicos y a partir del cual se pueden obtener las unidades de tiempo. Como este contador es finito, podría desbordarse y en ese caso habría que reiniciarlo. El **reinicio del contador** se lleva a cabo en función de su intervalo de valores y granularidad.

Mientras el contador no se desborda, el tiempo medido es monótono creciente, pero al desbordarse se producen saltos hacia atrás, perdiéndose la monotonía (no sería posible mantener la fecha, hora…). Como se requiere que el



tiempo sea monótono creciente, es necesario diseñar el contador con la suficiente capacidad para la aplicación a usar.

En sistemas distribuidos hay que tener en cuenta que cualquiera de los relojes que interactúan en el sistema pueden tener alguna diferencia, por lo que será necesario realizar algún tipo de sincronización entre ellos.

4. Relojes en Java

En Java hay diferentes opciones para trabajar con relojes.

- En el paquete estándar de Java (`java.lang`) existen las opciones:
  - `System.currenTimeMillis()`: es un método estático de la clase `System` que devuelve los milisegundos transcurridos desde el 01/01/1970 GTM.
  - Clases `Date` y `Calendar` (paquete `java.time`): hacen uso de `currentTimeMillis` para crear objetos. Son preferibles para trabajar con fechas. También se incluyen otras clases como `Clock`, `LocalDate`, LocalTime y otros tipos de datos.
- Real-Time Java: añade relojes de tiempo real con alta resolución, mayor precisión que los anteriores

Necesidades de los relojes para RTJ

Las necesidades básicas que deben cumplir los relojes para trabajar en un entorno de tiempo real son:

- **Reloj monótono**: progresa con un ritmo constante y no está sujeto a la introducción de "ticks" extras para reflejar los segundos intercalares (como sí hacen los relojes UTC). El ritmo constante es necesario para los algoritmos de control, que deben ejecutarse con una base regular.
  Muchos relojes monótonos son relativos al arranque del sistema y, por tanto, solo pueden usarse para medir el paso del tiempo, no el tiempo del calendario. Pero pueden ser usados para proporcionar tiempos límites en los que no se ha producido un evento, como pueda ser una interrupción.
- **Reloj de cuenta atrás:** deben poderse pausar, continuar y reiniciar.
- **Reloj del tiempo de ejecución de la CPU:** debe poder medir la cantidad de tiempo de procesador que ha consumido un cierto hilo u objeto.
- **Deben permitir tener una resolución menor a los milisegundos**.

En RTSJ se define así una jerarquía de clases de tiempo en la clase abstracta `HighResolutionTime`.

```
public abstract class HighResolutionTime implements Comparable, Cloneable{
 public abstract AbsoluteTime absolute(Clock clock);
 public abstract RelativeTime relative(Clock clock);
 ...
 public boolean equals(HighResolutionTime time);
 public int compareTo(HighResolutionTime time);

 public Clock getClock();

 public final long getMilliseconds();
 public final int getNanoseconds();

 public void set(HighResolutionTime time);
 public void set(long millis);
 public void set(long millis, int nanos);

 public static void waitForObject(Object target, HighResolutionTime time)
        throws InterruptedException;
}
```

Estos métodos son heredados por los subtipos de esta clase:

- **`AbsoluteTime`**: subclase para trabajar con tiempos absolutos, medidos sobre una referencia (la que tenga asociada el reloj).

```
public class AbsoluteTime extends HighResolutionTime{
  // various constructor methods including
  public AbsoluteTime();
  public AbsoluteTime(AbsoluteTime T);
  public AbsoluteTime(long millis, int nanos);
  public AbsoluteTime(Date date);

  public AbsoluteTime absolute(Clock clock);
  public RelativeTime relative(Clock clock);

  public Date getDate();
  public void set(Date date);

  public AbsoluteTime add(RelativeTime time);
  public AbsoluteTime add(long millis, int nanos);
  public RelativeTime subtract(AbsoluteTime time);
  public AbsoluteTime subtract(RelativeTime time);
  ...
}
```

- **`RelativeTime`**: permite trabajar con tiempos relativos, con una duración temporal medida por un determinado reloj.

```
public class RelativeTime extends HighResolutionTime{

  // various constructor methods including
  public RelativeTime();
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  public AbsoluteTime absolute(Clock clock);
  public RelativeTime relative(Clock clock);

  public RelativeTime add(long millis, int nanos);
  public RelativeTime add(RelativeTime time);
  public RelativeTime subtract(RelativeTime time);

  ...
}
```

- **`RationalTime`**: es una subclase de tiempo relativo que tiene asociada una frecuencia. Se utiliza para medir la frecuencia con la que ocurren ciertos eventos.

Dos métodos principales de `HighResolutionTime` son los métodos `absolute` y `relative`, que permiten conversiones de tiempos absoluto a relativo y viceversa.

`Clock` en RT-Java

```
public abstract class Clock{
  public Clock();
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolution);
}
```

Ejemplo de uso:

```
…
AbsoluteTime oldTime, newTime;
RelativeTime interval;
Clock clock = Clock.getRealtimeClock();
oldTime = clock.getTime();
// other computations
newTime = clock.getTime();
interval = newTime.subtract(oldTime);
…
```

5. Retardos, tiempos límites y temporizadores

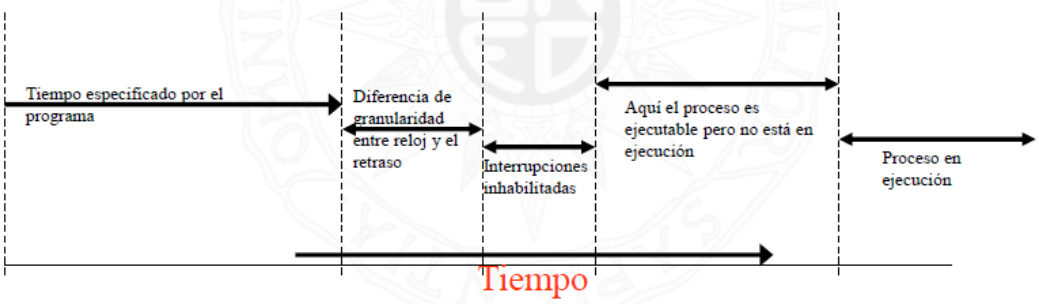
Los procesos deben, además de poder acceder al reloj, tener capacidad para retrasar su ejecución o suspenderla durante un intervalo de tiempo, es decir, retrasarse. Los retardos pueden ser:

- **Retardo relativo**: se especifica un intervalo de tiempo relativo. En Java, el método `sleep` permite asociar un retardo a un hilo con una precisión de milisegundos (`thread.currentThread().sleep(10000)` -> 10 segundos). En RT-Java, el método `sleep` proporciona una resolución mayor.
- **Retardo absoluto**: se especifica el instante absoluto en el futuro en el cual el proceso comenzará su ejecución o reanudará. Este tipo de retardos se puede implementar a partir de un retardo relativo y la hora actual (retardo relativo = hora absoluta - hora actual).

Tiempo de reanudación

En un sistema multiprocesos se pueden producir desplazamientos en los retardos ya que existen varios factores que pueden modificar el tiempo de reanudación del mismo:

- Cuando se inhiben las interrupciones en el sistema: si hay una interrupción para despertar el proceso, esto se retrasa hasta volver a habilitar las interrupciones.
- La diferente granularidad entre la expresión del tiempo y la granularidad real del reloj: esto se produce, por ejemplo, si se implementa el reloj mediante una interrupción periódica con una granularidad mayor que la del reloj.
- Durante el paso de listo a ejecutable, si hay procesos de mayor prioridad también listos, el proceso se retrasaría hasta terminar los de mayor prioridad.



Este tipo de desplazamientos adicionales forman la **deriva local**, esto es, el tiempo adicional que dura el retardo y que no se puede eliminar. Por otro lado, la **deriva acumulada** se corresponde al desplazamiento que se obtiene en sucesivas invocaciones, causada por la suma de las derivas locales superpuestas y sí puede ser eliminada.

```
T = 50;
siguiente = time();
while (true){
        sleep(siguiente-time());
        tarea();
        siguiente = siguiente + T;
}
```

Tiempos límites (*timeouts*)

Los **tiempos límites** (***timeouts***) permiten establecer una restricción sobre el tiempo que una tarea está dispuesta a esperar para una comunicación, esto es, limitar el tiempo en el que se quiere que ocurra un proceso. Para ello se establecen

distintas restricciones a la ocurrencia de un suceso en determinadas tareas. También se usan *timeouts* para para programar tiempos límite de ejecución de las tareas en función de sus características y las restricciones que se tengan.

Se puede hacer uso de *timeouts* en cualquiera de las posibilidades de sincronización:

- Semáforos.
- Regiones críticas condicionales (CCR).
- Variables de condición en monitores.
- Entradas en objetos protegidos.

Un *timeout* puede considerarse una notificación asíncrona, por lo que se pueden utilizar los modelos de terminación (ATC) para acciones, pueden ser útiles para capturar errores de código (como bucles infinitos) o para definir tareas con partes obligatorias y opcionales (la parte opcional solamente se ejecutará si hay tiempo disponible para ella).

Para trabajar con tiempos límite en Java, se utilizan los métodos:

- `wait`: con granularidad de milisegundos (clase `Object`).
- `waitForObject`: con granularidad de nanosegundos (clase abstracta `HighResolutionTime`).

Para RT-Java, en las acciones los tiempos límite son proporcionados por una subclase de `AsynchronouslyInterruptedException` denominada `Timed`.

```
public class Timed extends AsynchronouslyInterruptedException
                           implements java.io.Serializable{

  public Timed(HighResolutionTime time) throws IllegalArgumentException;

  public boolean doInterruptible(Interruptible logic);

  public void resetTime(HighResolutionTime time);
}
```

Tareas con partes opcionales en Java

```
public class PreciseResult{
  public resultType value; // the result
  public boolean preciseResult; // indicates if it is imprecise
}

public class ImpreciseComputation {
  private HighResolutionTime completionTime;
  private PreciseResult result = new PreciseResult();

  public ImpreciseComputation(HighResolutionTimer T){
    completionTime = T; //can be absolute or relative
  }
  private resultType compulsoryPart(){
    // function which computes the compulsory part
  }
```

```
public PreciseResult Service(){  // public service
   Interruptible I = new Interruptible(){ //clase anónima
     public void run(AsynchronouslyInterruptedException exception)
                    throws AsynchronouslyInterruptedException{
       // this is the optional function which improves on the
       // compulsory part
       boolean canBeImproved = true;
       while(canBeImproved){
         // improve result
         synchronized(this) {
           // write result
           // synchronized ensures atomicity of the write operation
         }
       }
       result.preciseResult = true;
     }
     public void interruptAction(AsynchronouslyInterruptedException e){
       result.preciseResult = false;
     }
   };//fin de la clase anónima
```

```
    Timed t = new Timed(CompletionTime);

    result.value = compulsoryPart(); // compute the compulsory part
    if(t.doInterruptible(I)) {// execute the optional part with the timer
        return result;
    } else { ... };
  }//fin de método que define el servicio
}//fin de la clase ImpreciseComputation
```

Temporizadores

Una **temporización** es una restricción sobre el tiempo que una tarea permanece a la espera de un evento. Por eso, generalmente se incluye en las primitivas de comunicación y sincronización. Así, un **temporizador** es un mecanismo que permite avisar de que ha transcurrido un cierto tiempo. Puede ser de dos tipos: de un **solo disparo** o **periódico**. Estos temporizadores pueden tener asociados tres eventos:

- Activar: fijar el valor y el tipo.
- Cancelar: cuando el evento llega a tiempo.
- Expirar.

Temporizadores en RT-Java

Los temporizadores pueden implementarse con la clase abstracta `Timer`.

```
public abstract class Timer extends AsyncEvent{
  protected Timer(HighResolutionTimer time, Clock clock, AsyncEventHandler
                                                          handler);
  public ReleaseParameters createReleaseParameters();
  public AbsoluteTime getFireTime(); //Tiempo en el que se produce el evento
  public void reschedule(HighResolutionTimer time); //Modifica el tiempo
  public Clock getClock();
  public void disable(); //Inhabilita el temporizador
  public void enable(); //Habilita el temporizador
  public void start(); //Inicial el temporizador
  …
}
```

A partir de la clase `Timer` se pueden implementar los temporizadores de un solo disparo con la clase `OneShotTimer` o periódicos con `PeriodicTimer`:

```
public class OneShotTimer extends Timer{
  public OneShotTimer(HighResolutionTimer time, AsyncEventHandler handler);
}
```

```
public class PeriodicTimer extends Timer{
  public PeriodicTimer(HighResolutionTimer start, RelativeTime interval,
                                          AsyncEventHandler handler);
  public ReleaseParameters createReleaseParameters();
  public void setInterval(RelativeTime interval);
  public RelativeTime getInterval();
  …
}
```

6. Requisitos temporales

Para especificar los requisitos temporales, existen dos formas diferentes:

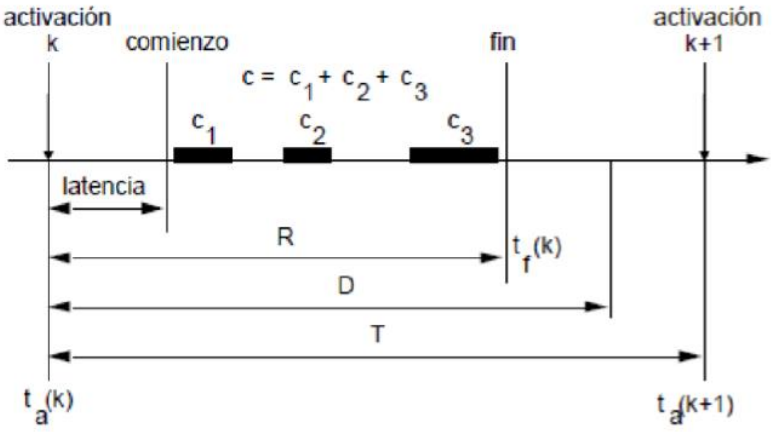
- **Métodos formales**: hacer uso de lenguajes de semántica formalmente definida y requisitos temporales. Se incluye el uso de notaciones que posibiliten su representación y análisis. Estos métodos no están completamente asentados para trabajar con ellos en sistemas de tiempo real
- **Métodos analíticos**: están centrados en las prestaciones de tiempo real en cuanto a la factibilidad de planificación de la carga de trabajo en función de los requisitos disponibles (como los procesadores).

El proceso de verificación de requisitos temporales se realiza en dos etapas:

1. **Verificar los requisitos/diseños**: puede requerir un razonamiento formal para verificar el orden temporal y causal. Si esta etapa no se pasa, no es necesario realizar la segunda. Se comprueba si los requisitos temporales son coherentes y consistentes y si son factibles.
2. **Verificar la implementación**: consiste en chequear el conjunto de recursos hardware del que se dispone y, con todas sus posibilidades, comprobar si se pueden satisfacer los requisitos temporales que ya han sido verificados en la etapa anterior en función de su diseño.

En este contexto y para facilitar las especificaciones de restricciones temporales en los sistemas de tiempo real, se definen los **ámbitos o marcos temporales**, que corresponden a un conjunto de sentencias asociadas a una restricción temporal. Estos marcos temporales se definen mediante los **atributos temporales** asociados a una secuencia de instrucciones. Entre estos atributos estarían:

- **Latencia mínima** (Jmin): tiempo mínimo que debe transcurrir desde que se produce el evento de activación hasta que se ejecuta completamente el marco temporal.
- **Latencia máxima** (Jmax): tiempo máximo que debe transcurrir desde que se produce el evento de activación hasta que se ejecuta el marco temporal.
- **Plazo de respuesta** (D): representa el máximo tiempo en que la ejecución del marco temporal debe haber finalizado, medido desde el evento de activación del mismo.
- **Tiempo de respuesta** (R): tiempo transcurrido realmente desde que se produce el evento de activación hasta que acaba la ejecución del marco temporal. Por tanto, existe una restricción con respecto al plazo de respuesta y este tiempo debe ser inferior o como mucho igual al plazo de respuesta.
- **Tiempo de cómputo** (C): tiempo de utilización del procesador dentro del marco temporal.
- **Tiempo límite** (L): máximo tiempo en que la ejecución del marco temporal debe haber finalizado, medido desde el comienzo de su ejecución.



Los **marcos temporales** se dan como combinación de estos atributos. En general, los marcos temporales pueden ser:

- **Periódicos**: se repiten periódicamente y pueden corresponder al muestreo de una señal o la ejecución de un lazo de control, teniendo un plazo de respuesta asignado. Se caracterizan por tener un periodo de ejecución T entre dos acciones sucesivas.
- **Aperiódicos o esporádicos**: corresponden a eventos asíncronos que ocurren fuera del computador. A menudo se define un periodo mínimo entre dos eventos aperiódicos (S).

Los diferentes incumplimientos que pueden darse en un marco temporal se denominan **fluctuaciones o jitter**. No todas las aplicaciones requieren de un cumplimiento estricto, por lo que pueden caracterizarse como:

- **Estrictos** (hard): se necesita que se cumplan los requisitos temporales. Si no se cumplen, fallan.
- **Flexibles** (soft): se pueden tolerar ciertos incumplimientos.
- **Interactivos** (interactive): no hay especificados tiempos de respuesta, sino que se esperan cumplimientos adecuados.

7. Tolerancia a fallos

Aunque se haya realizado una verificación del sistema, hay que tener en cuenta que se pueden producir incumplimientos temporales de los plazos, que pueden deberse a distintos motivos:

- Errores en el cálculo del tiempo de cómputo.
- Análisis de tiempos de respuesta no realistas.
- Errores propios en las herramientas de análisis
- No se cumplen las hipótesis de diseño.

En estas situaciones, lo ideal sería poder realizar una detección de los fallos y una posterior recuperación. Para poder considerar un sistema como tolerante a **fallos de temporización**, el sistema tiene que poder detectar:

- El desbordamiento de un tiempo límite o de ejecución.
- Eventos esporádicos con alta concurrencia, que ocurran más a menudo de lo previsto.
- Tiempos límites de espera en comunicaciones con otros procesos.

Las consecuencias que se dan ante errores de temporización son que otros procesos deben alterar sus límites temporales, que finalicen, se suspendan o arranquen procesos. En muchos casos, los más perjudicados son los procesos de menor prioridad que pueden tener que suspenderse.

En caso de que se pueda producir una interrupción o suspensión de algún proceso, estos procesos han de llevar a cabo alguna acción posterior a la interrupción como devolver el mejor resultado que se tenga hasta el momento, tener la posibilidad de cambiar de algoritmo o pasar a estar disponible para recibir nuevas instrucciones.