Tema 28- Programación en tiempo real. Interrupciones. Sincronización y comunicación entre tareas. Lenguajes.

Índice

1. Introducción.

Los paradigmas de programación han ido evolucionando a lo largo del tiempo para satisfacer las necesidades de los usuarios relativas a simular procesos en tiempo real. Para afrontar este nuevo paradigma, normalmente se parte de los conocimientos previos de programación secuencial en paradigmas estructurados u orientados a objetos.

El objetivo principal en la programación en tiempo real es lograr que los sistemas limitados o condicionados por problemas de tiempo funcionen de forma correcta ante determinadas circunstancias e imprevistos. Por ejemplo, un exceso de conexiones o peticiones a un sistema puede provocar que éste no las pueda atender en un tiempo razonable, o que incluso colapse y se bloquee. Por otra parte, algunas acciones o eventos que ocurren sobre un programa exigen de él una respuesta inmediata.

Existen distintos mecanismos que se pueden utilizar para la programación en tiempo real, y mediante los que se construyen *sistemas de tiempo real*, es decir, sistemas informáticos o software que responde a este paradigma, y que permite una comunicación o un funcionamiento en tiempo real, cuyas características vamos a ver a continuación. También veremos distintas técnicas de programación en tiempo real que podemos aplicar, desde algunas ya tradicionales basadas en interrupciones, uso de semáforos, etc, como otras más recientes basadas en gestión de hilos, llamadas a métodos asíncronos o comunicaciones cliente-servidor en tiempo real.

Este tema forma parte del bloque del temario referido a la programación, que abarca desde el tema 23 hasta el tema 33. En concreto, forma un bloque sobre los diferentes paradigmas de programación, el cual lo componen los temas 25 (programación estructurada), tema 26 (programación modular), tema 27 (programación orientada a objetos) y tema 28 (programación en tiempo real). No obstante, también se relaciona directa o indirectamente con el resto de temas del bloque, e incluso con algunos temas de otros bloques, como el tema 16, dedicado a la gestión de procesos en el sistema operativo.

2. Sistemas de tiempo real. Características. Lenguajes.

Un sistema en tiempo real (STR) es un sistema informático en el que, para que las operaciones que se realizan sean correctas, no sólo es necesario que la lógica e implementación de los programas sean correctas, sino también el tiempo en el que dichas operaciones entregan su resultado. Si las restricciones de tiempo no son respetadas, se dice que el sistema ha fallado. Un STR tiene las siguientes características principales:

- **Concurrencia:** Con la concurrencia se descompone un programa en procesos o tareas que se ejecutan de forma simultánea. Se pueden

utilizar procesos pesados, o hilos para desacoplar actividades con velocidades de procesamiento muy diferentes.

- **Dependencia temporal:** Debe especificarse el tiempo en que deben completarse las acciones. La concurrencia puede ayudar a satisfacer las necesidades temporales.

- **Fiabilidad:** Tanto el software como el hardware deben tener mecanismos para recuperarse de fallos.

- **Complejidad:** La variedad de funciones que puede realizar un STR aumenta considerablemente la complejidad del mismo. Algunos STR pueden llegar a tener millones de líneas de código.

En función de lo crítico o no que sea el factor tiempo, los STR pueden ser:

- **Críticos:** es imprescindible que el sistema realice sus funciones en el tiempo especificado. En caso de que no se complete la tarea en dicho plazo, se dice que el sistema ha fallado. Unos ejemplos típicos serían el sistema de un marcapasos cardíaco, el sistema que controla el frenado (ABS) de un coche y el sistema que controla un reactor nuclear. Cuando, además de ser un STR crítico, los plazos de tiempo son muy cortos, se dice que el STR es **estricto**.

- **Acríticos:** el sistema puede dar respuesta en un plazo de tiempo superior al especificado, lo que implica una pérdida de calidad del mismo. Como ejemplos tenemos, entre otros, un sistema de reservas de vuelos en una aerolínea o un sistema de predicción del valor de las acciones en la Bolsa de Valores.

2.1. STR y programación orientada a eventos.

Un STR responde a una serie de eventos que pueden ser producidos en su interior o en su entorno, llamándose eventos internos o externos, respectivamente. Pueden ser eventos periódicos o aperiódicos, y en este último caso normalmente se desconoce cuándo se van a producir. Por lo tanto, los STR están muy asociados al paradigma de la **programación orientada a eventos**, donde la estructura de un programa y su ejecución están definidos por los sucesos que ocurren en el sistema o que ellos mismos provocan.

Mientras que en la programación secuencial (estructurada) el programador determina cuál será el flujo del programa y cuándo habrá una intervención externa, en la programación orientada a eventos el programador deberá definir los sucesos que dirigirán el programa y las acciones que se desencadenarán como consecuencia de dichos sucesos. Al ejecutarse el programa se llevan a cabo primero las inicializaciones, y el programa quedará en espera hasta que se produzca algún evento.

La programación orientada a eventos es ampliamente utilizada en programas con interfaz gráfica de usuario. Para soportar este tipo de desarrollo, generalmente interactúan dos tipos de herramientas: una que permite realizar diseños gráficos, y un lenguaje de alto nivel que permite codificar los eventos. Con dichas herramientas es posible desarrollar cualquier tipo de aplicación basada en el entorno.

Sin embargo, la programación en tiempo real no sólo se vale de lenguajes de alto nivel y de interfaces gráficas, puesto que en muchos otros ámbitos, como por ejemplo eventos desencadenados por el sistema operativo, o eventos temporales (como *backups* periódicos, o e-mails de respuesta automática), la programación en tiempo real tiene un importante campo de acción.

2.2. Lenguajes y herramientas para desarrollo de STR

A la hora de desarrollar los sistemas de tiempo real, podemos emplear multitud de lenguajes de programación, y herramientas asociadas a dichos lenguajes, que nos facilitan la implementación de ciertos aspectos.

Por ejemplo, podemos basarnos en lenguajes ya muy tradicionales, como C o C++, para realizar fundamentalmente sistemas de tiempo real con programación a bajo nivel. Estos lenguajes son especialmente apropiados para tareas como la gestión de interrupciones que veremos a continuación, o la programación y sincronización de procesos en el sistema, a través de mecanismos de exclusión mutua o semáforos, que son conceptos que también explicaremos en breve.

Pero, además, la evolución de los lenguajes de programación ha hecho que se facilite tremendamente la tarea de programar aplicaciones que respondan en un determinado tiempo, o que lancen procesos o tareas en paralelo para agilizar la resolución de problemas. Así, lenguajes como Java o C# permiten lanzar hilos, coordinarlos y sincronizarlos para obtener un resultado final. También lenguajes como Java, o JavaScript, permiten ejecutar tareas asíncronas, y decirles qué tienen que hacer al finalizar, para no quedarnos esperando a recoger el resultado. Veremos algunos ejemplos de gestión de *callbacks* de respuesta y promesas, relacionado con este aspecto.

Finalmente, la evolución que ha tenido el desarrollo de aplicaciones web, y cliente-servidor en general, en los últimos años, ha favorecido la aparición de nuevos lenguajes, frameworks y librerías que podemos emplear para desarrollar sistemas de comunicación en tiempo real cliente-servidor.

- En la parte del servidor se dispone de distintos lenguajes como PHP, Java (JSP), ASP.NET, Python o incluso el mismo JavaScript. También existen numerosos frameworks de desarrollo que facilitan el desarrollo de aplicaciones servidor, como por ejemplo Laravel o Symfony, en el caso de PHP, Spring en el caso de Java, o Django en el caso de Python, además de Node.js para JavaScript.

- En la parte del cliente, el lenguaje JavaScript, apoyado también en su reciente hermano pequeño TypeScript, proporciona un amplio abanico de opciones de comunicación con el servidor. Además, disponemos de varios frameworks de desarrollo y librerías para ayudarnos en esta comunicación, tales como Angular, React, Vue, o algunas algo anteriores como jQuery.

En los siguientes apartados veremos distintos tipos de sistemas de tiempo real, junto con sus características principales y las alternativas que se tienen para poderlos desarrollar. También pondremos varios ejemplos utilizando algunos d estos lenguajes y librerías, para ilustrar gran parte de las alternativas de programación en tiempo real que veremos.

3. Sistemas basados en interrupciones.

Los sistemas basados en interrupciones son unos de los STR más tradicionales o "veteranos" que existen. Giran en torno al concepto de **interrupción**, que podría definirse como la suspensión temporal de la ejecución de un programa para ejecutar una tarea que normalmente es ajena al propio programa. Por ejemplo, una comunicación con un dispositivo periférico del sistema. Son operaciones que se realizan a muy bajo nivel (a nivel de CPU y dispositivos periféricos), por lo que su programación suele ser costosa.

Según la fuente que produce la interrupción, podemos hablar de:

- Interrupciones **hardware**: pueden ser internas (provocadas por la CPU) o externas (provocadas por algún dispositivo periférico de entrada/salida). En el primer caso, pueden deberse a errores en alguna operación (división por cero, desbordamiento de pila, etc.), y en el segundo caso, también pueden deberse a errores, o a simples accesos a bajo nivel para leer o enviar algo al dispositivo.

- Interrupciones **software**: producidas por la ejecución de alguna instrucción en la CPU.

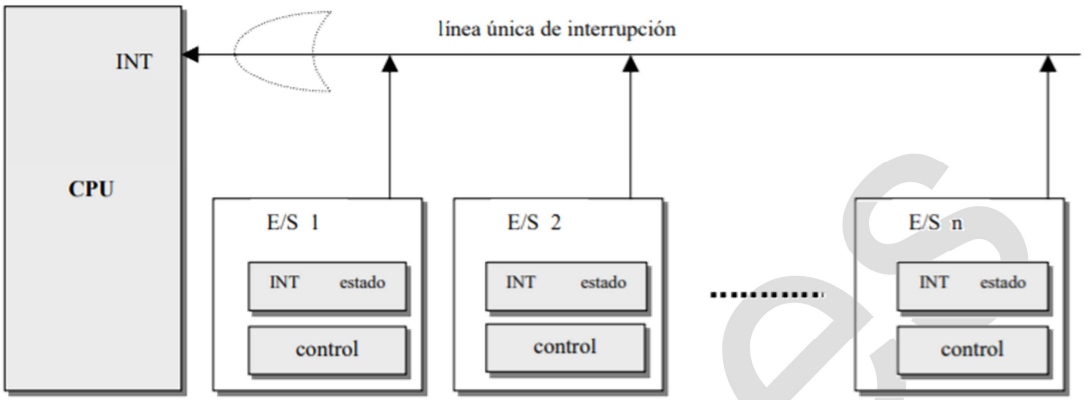
En ambos casos, cuando se recibe la interrupción debe tratarse, y esto se realiza a través de la tabla de vectores de interrupción, tabla que contiene las direcciones de las rutinas de tratamiento de cada interrupción. Estas direcciones pueden ser fijas (*autovectorizadas*) o tener una parte variable que suministra el propio dispositivo de E/S cuando se produce la interrupción.

El proceso de interrupción se desarrolla en los siguientes pasos:

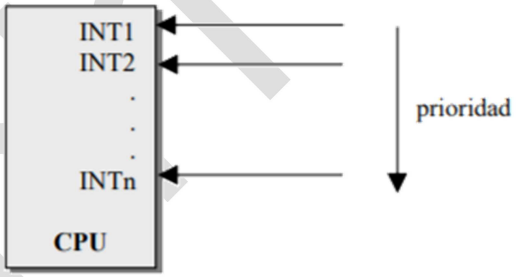
1. El dispositivo de E/S genera la petición de interrupción activando INT.
2. La CPU la reconoce activando RINT.
3. Los dispositivos de E/S que reciben RINT y no han interrumpido, la transmiten al siguiente elemento de la cadena.
4. El dispositivo de E/S que recibe RINT y ha realizado la petición coloca el vector de interrupción n en el bus de datos.

5. A partir del vector n la CPU bifurca a la rutina de tratamiento correspondiente al dispositivo.

Cuando uno o más dispositivos genera una señal de interrupción, la CPU debe identificar por software el dispositivo causante de la interrupción, y en caso de ser varios, establecer el orden de prioridad entre ellos. La identificación se realiza consultando los registros de estado locales de cada módulo de E/S.



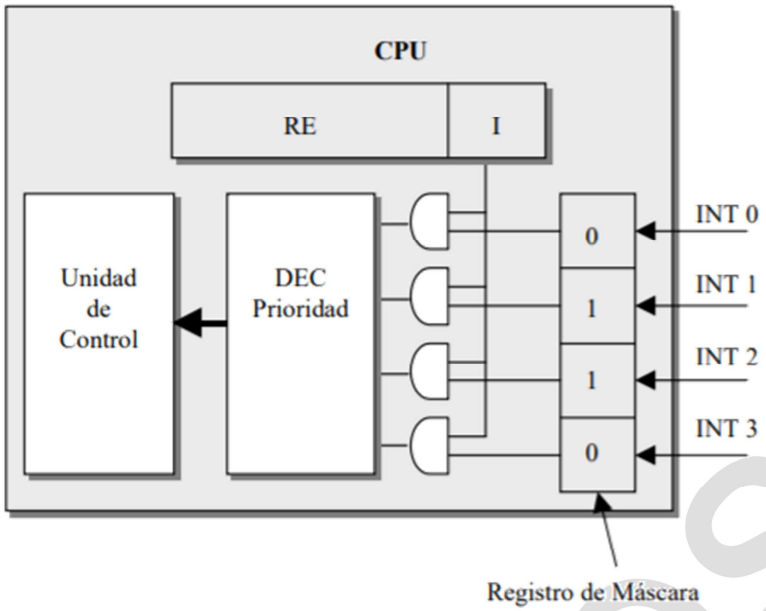
En el caso de tener que tratar más de una interrupción a la vez, la prioridad se puede establecer por hardware o por software. La priorización por hardware se da en las interrupciones autovectorizadas, y se establece el orden según la proximidad a la CPU. También si el procesador admite varias líneas de interrupción, éste tiene programado por hardware la prioridad de cada línea.



En las interrupciones no vectorizadas, la prioridad se suele establecer por software, atendiendo al orden de recorrido de los dispositivos de E/S.

3.1. Enmascaramiento de interrupciones.

El sistema de interrupciones de un procesador dispone en general de la posibilidad de ser inhibido, es decir, impedir que las interrupciones sean atendidas por la CPU. Esta posibilidad hay que utilizarla en determinadas ocasiones en las que por ningún concepto se puede interrumpir el programa en ejecución. Además de la inhibición total del sistema, existe la posibilidad de enmascarar individualmente algunas de las líneas de interrupción utilizando un registro de máscara, que mediante operaciones AND, logra anular el efecto de las interrupciones en las líneas afectadas.

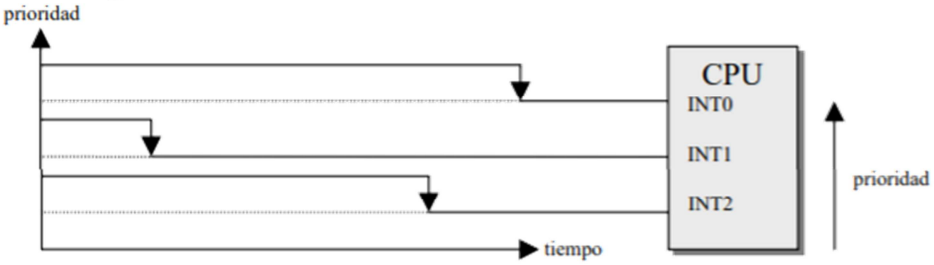


A nivel software también se pueden inhibir las interrupciones producidas sobre una misma línea por diferentes periféricos.

3.2. Anidamiento de interrupciones.

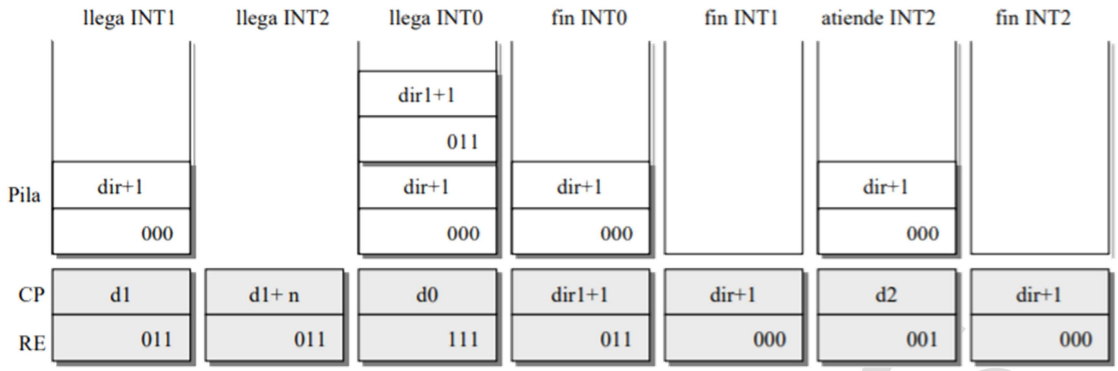
Un tema importante en un sistema de interrupciones el de la capacidad de anidar interrupciones, es decir, la posibilidad de interrumpir la rutina de tratamiento de una interrupción por la generación de otra interrupción. Tal posibilidad viene determinada por el sistema de prioridades: un dispositivo sólo puede interrumpir a otro cuando su nivel de prioridad es mayor que el que se está atendiendo.

Para describir con más detalle la forma en que opera la combinación de las prioridades y el anidamiento en un sistema de interrupciones, consideremos una CPU con tres líneas de interrupción: INT0, INT1, INT2, siendo la primera la más prioritaria, y la última la menos prioritaria. Supongamos que llegan tres peticiones de interrupción en el orden temporal del siguiente esquema: primero se produce la interrupción INT1, luego INT2 y finalmente INT0.



Las interrupciones con menor prioridad no podrán interrumpir la rutina de tratamiento de una de mayor prioridad. La evolución consiguiente de la pila y el estado de la CPU (CP y RE) será el siguiente: primero se atenderá INT1, y

antes de que finalice, llegan INT2 (que se pospone) e INT0 (que se añade a la pila por delante de INT1). Cuando INT0 finaliza, se sigue atendiendo INT1, y cuando ésta finaliza, se trata finalmente la interrupción INT2.



4. Programación concurrente. Procesos e hilos.

La programación concurrente es una técnica de programación que permite generar distintos procesos o hilos simultáneos, de forma que todos ellos colaboran mediante mecanismos de coordinación y sincronización para resolver un problema común.

Los tres servicios básicos que debe proporcionar la programación concurrente son: definición de procesos, sincronización de procesos y comunicación entre procesos.

4.1. Procesos vs hilos.

A la hora de hablar de programación concurrente, podemos optar indistintamente por desarrollar procesos y/o hilos, también llamados *hebras*. La diferencia entre ambos conceptos radica en un par de aspectos fundamentalmente.

- Por un lado, un proceso es un objeto “pesado” en memoria, que cuenta con su propio contexto (variables locales propias del proceso), además de las instrucciones a ejecutar. El hilo, por el contrario, no tiene contexto propio (todos los hilos de un programa comparten contexto), por lo que, una vez iniciados, su “puesta en escena” en memoria es más rápida y consume menos recursos.

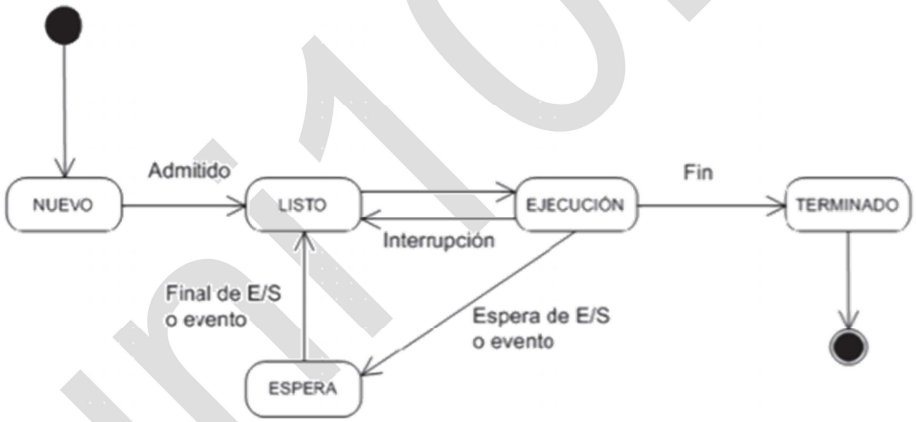
- Por otra parte, los procesos suelen representar programas independientes, aunque sean varias instancias de un mismo programa. Si, por ejemplo, abrimos dos ventanas de Google Chrome en el sistema operativo, esto desencadenará dos procesos distintos (sobre el mismo programa) en el administrador de tareas del sistema. En cambio, los hilos se asientan sobre un único proceso o programa, por lo que, a

efectos de administración del sistema operativo, son más “invisibles”, ya que no cuelgan directamente de él, sino del programa que los inició.

Por otra parte, tanto hilos como procesos también tienen una serie de características comunes. Por ejemplo, ambos tienen una serie de estados por los que pasan durante su ciclo de vida:

- **Nuevo**: el proceso o hilo se acaba de crear, y aún no ha comenzado a ejecutarse.
- **Listo**: el proceso o hilo no se está ejecutando, pero está preparado para ello.
- **En ejecución**: el proceso o hilo se está ejecutando ahora mismo en algún procesador
- **En espera**: el proceso o hilo está esperando a que suceda algún evento. Puede que sea una entrada de usuario, o el desbloqueo de algún fichero… Cuando esto suceda, el proceso/hilo se desbloqueará
- **Terminado**: el proceso o hilo ha terminado su tarea, o el sistema le ha forzado a finalizar mediante una interrupción

Estos estados pueden relacionarse entre sí atendiendo al siguiente esquema, que representa el ciclo de vida de los procesos o hilos:



4.2. Definición de procesos.

Un proceso puede considerarse como un programa en ejecución, y por eso necesita recursos: tiempo de CPU, memoria, archivos y dispositivos de E/S para realizar su tarea. Un proceso es la unidad de trabajo en la mayoría de sistemas. La interacción entre procesos puede tener un comportamiento independiente, cooperativo o competitivo.

Los sistemas operativos proporcionan mecanismos para crear procesos concurrentes, y ciertos lenguajes permiten definir estos procesos sobre dicho sistema operativo. Es el caso de C, que permite crear nuevos procesos mediante su llamada al sistema *fork*.

El siguiente ejemplo crea un proceso hijo a partir del programa principal. La llamada a la instrucción *fork* devuelve 0 para el caso del proceso hijo, por lo que de ese modo podemos identificarlo:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void ejemploFork()
{
    int x = 1;

    if (fork() == 0)
        printf("El hijo tiene x = %d\n", ++x);
    else
        printf("El padre tiene x = %d\n", --x);
}

int main()
{
    ejemploFork();
    return 0;
}
```

La salida del programa mostrará los mensajes “El padre tiene x = 0” y “El hijo tiene x = 2”, en un orden arbitrario (dependiendo de quién finalice o acceda antes a la CPU). Pero, en cualquiera de los dos casos, queda evidenciado que ambos procesos tienen un contexto propio, y una copia propia de la variable x.

4.3. Definición de hilos.

Como hemos comentado, los hilos son una especie de procesos “ligeros”, que comparten un contexto común (variables locales compartidas). Normalmente los hilos se crean a partir de un programa, y dependen de éste (cuando el programa finaliza, lo hacen los hilos que ha generado).

Existen varios lenguajes que permiten definir hilos. Algunos ya muy tradicionales como C++, a través de librerías POSIX, y otros más recientes como C# o Java, que ofrecen unos mecanismos de gestión de hilos más avanzados.

El siguiente ejemplo muestra cómo crear dos hilos en Java, empleando expresiones lambda (programación funcional). Lo que se hace es implementar la interfaz *Runnable* que ofrece Java para definir hilos, y ponerlos en marcha. Cada hilo ejecuta su método implementado (que equivale al método *run* que ofrece la interfaz *Runnable*), por lo que podemos deducir que uno de los hilos va a contar del 1 al 10 (esperando 1 segundo entre cada número) y el otro va a contar del 10 al 1 (esperando medio segundo entre cada número):

```
public class Prueba
{
```

```
    public static void main(String[] args)
    {
        Runnable hilo1 = () -> {
            for (int i = 1; i <= 10; i++)
            {
                System.out.println(i);
                try
                {
                    Thread.sleep(1000);
                } catch (Exception e) {}
            }
        };

        Runnable hilo2 = () -> {
            for (int i = 10; i >= 1; i--)
            {
                System.out.println(i);
                try
                {
                    Thread.sleep(500);
                } catch (Exception e) {}
            }
        };

        Thread t1 = new Thread(hilo1);
        Thread t2 = new Thread(hilo2);

        t1.start();
        t2.start();
    }
}
```

La clase *Thread* es la que se encarga de definir los hilos (a partir de los objetos *Runnable* creados antes), y el método *start* los pone en ejecución. A partir de ese punto, se alternarán para ocupar la CPU (en el caso de sistemas con un solo núcleo) hasta finalizar la tarea.

Además, en el caso de Java, por ejemplo, podemos definir distintos tipos de hilos. Por un lado se tienen los hilos *Runnables*, como los del ejemplo anterior, que simplemente realizan una tarea y finalizan. Pero también están los hilos *Callables*, que antes de finalizar la tarea devuelven un resultado, que puede ser recogido por el programa principal, o por otro hilo coordinador.

4.4. Mecanismos de sincronización y coordinación.

Cuando definimos múltiples procesos e hilos que deben acceder de forma simultánea a un determinado recurso (por ejemplo, un fichero en el que varios tengan que escribir información, o una variable que varios tengan que manipular), es necesario establecer algún mecanismo que permita que dicha

manipulación no corrompa el recurso compartido. Dicho de otro modo, que el resultado final del proceso no difiera del que se habría producido si los procesos o hilos se hubiesen iniciado cada uno tras la finalización del anterior, sin concurrencia.

Para conseguir este objetivo, existen distintas estrategias y herramientas que podemos utilizar. Veamos algunas de las más habituales.

4.4.1. Exclusión mutua.

Un método para controlar el acceso a un recurso compartido consiste en declarar una sección de código como crítica para luego controlar la entrada de acceso a dicha sección. Si tenemos, por ejemplo, un sistema con varios hilos y cada hilo tiene un segmento de código que permite que ese hilo pueda actualizar variables comunes como una tabla o un archivo compartido, denominaremos a ese segmento de código que lo actualiza **sección crítica**.

Los mecanismos de exclusión mutua se pueden utilizar para evitar situaciones inconsistentes. Por ejemplo, cuando se accede simultáneamente a una variable que acumula el saldo de una cuenta, podríamos encontrarnos con lo siguiente:

Tiempo	Hebra 1	Hebra 2	Saldo
T1	Saldo leído: 8000		8000
T2		Saldo leído: 8000	8000
T3	Ingresa: 200		8000
T4		Ingresa: 200	8000
T5	Actualiza saldo: 8000 + 200		8200
T6		Actualiza saldo: 8000 + 200	8200!!

En este ejemplo, la hebra o hilo que actualiza el saldo debe bloquear al resto antes de la lectura del saldo y desbloquearlo una vez actualizado su valor. Ésta es la manera de garantizar que el resultado de la actualización será el mismo que se hubiera producido si solamente tuviéramos un hilo en ejecución.

El sistema debe tener la característica de que cuando un hilo está ejecutando su sección crítica no debe permitir que otros hilos ejecuten esa misma sección. La ejecución de secciones críticas por los hilos es mutuamente exclusiva en el tiempo, y por eso a esta estrategia se le denomina exclusión mutua, aunque también se le conoce por su nombre abreviado en inglés, *mutex* (*mutual exclusion*).

El algoritmo de la exclusión mutua es relativamente sencillo:

- Cuando ningún hilo está ejecutando su sección crítica y algunos hilos desean hacerlo, el primero que consiga entrar bloqueará al resto.

- Mientras un hilo esté ejecutando su sección crítica, ningún otro hilo puede estar en la misma ejecución. Sí que pueden estar ejecutando otras cosas que no sean sección crítica.

- Cuando el hilo que ocupa la sección crítica finaliza su tarea, desbloquea dicha sección, y el siguiente hilo en espera pasará a ejecutarla.

Obviamente, si dos procesos no comparten variables o recursos, no se necesita exclusión mutua.

En Java, por ejemplo, la exclusión mutua se consigue definiendo funciones o bloques de código con el modificador *synchronized*. Esto hace que sólo un hilo a la vez puede estar ejecutando un bloque *synchronized* en todo el código. Podría quedar algo así:

```
class GestionCuentaBancaria
{
     int saldo;

     … // Otros métodos y atributos

     public void synchronized actualizarSaldo(int euros)
     {
          saldo += euros;
     }
}
```

Esta sección crítica puede ser accedida por distintos hilos para hacer distintas operaciones:

```
// Variable compartida
GestionCuentaBancaria cuenta = new GestionCuentaBancaria();

Runnable r1 = () -> {
     cuenta.actualizarSaldo(100);
};

Runnable r2 = () -> {
     cuenta.actualizarSaldo(-50);
};

Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);
t1.start();
t2.start();
```

4.4.2. Uso de semáforos.

Los semáforos son otro mecanismo de sincronización entre procesos o hilos, que se gestionan a través de una variable entera no negativa. Los procesos que compiten por un recurso actualizan dicha variable, y en función de su valor podrán acceder o no al recurso.

El semáforo admite básicamente dos operaciones sobre él, que típicamente se llaman *wait* y *signal*, aunque esto depende del lenguaje de programación utilizado. Ambas operaciones son atómicas, es decir, o se completan en su totalidad o no se realizan.

La operación **wait(S)** actúa sobre la variable semáforo, S. Si es mayor que cero, disminuye su valor en uno; en caso contrario, demora el proceso hasta que S sea mayor que cero (y entonces disminuye su valor). Si el proceso logra disminuir el valor de S, entonces puede gestionar el recurso compartido. De lo contrario, deberá esperar

La operación **signal(S)** incrementa el valor del semáforo S, en uno. Lo realiza el proceso o hilo que ha finalizado su trabajo en la zona compartida, para permitir que otro proceso o hilo pueda entrar.

En la gestión de semáforos debemos tener precaución de evitar el **interbloqueo,** una situación que se puede dar cuando el proceso o hilo que ha ocupado un semáforo se queda bloqueado a la espera de que otro se libere, y el proceso que está ocupando ese otro semáforo está esperando que el primero también quede liberado. Ninguno de los dos procesos podrá avanzar en su ejecución.

4.4.3. Uso de datos atómicos.

Algunos lenguajes de programación también permiten utilizar tipos de datos atómicos. Estos tipos de datos permiten que se hagan operaciones sobre ellos de forma atómica, es decir, que cualquier operación de incremento, decremento o modificación de valor debe completarse, o de lo contrario se descartarán los cambios parciales que se hayan hecho.

Así, por ejemplo, el lenguaje Java ofrece distintos tipos de datos atómicos, como por ejemplo *AtomicInteger* o *AtomicLong*, que sirven para tipos numéricos. También ofrece el tipo *AtomicReference*, que se emplea para convertir en atómico cualquier tipo de dato compuesto que se quiera, e incluso *AtomicReferenceArray*, para definir un array de objetos del tipo que se quiera, como atómico. El siguiente código, por ejemplo, define una variable de texto (*String*) de forma atómica.

```
AtomicReference<String> nombre = new AtomicReference<String>();
nombre.set("Nacho");
System.out.println("El nombre es " + nombre.get());
```

Así, cualquier modificación que se haga sobre ella se ejecutará de una sola vez, evitando que la cadena se pueda corromper por el acceso simultáneo de varios hilos. Si, por ejemplo, hay varios hilos intentando cambiar el valor, o añadiendo texto uno a continuación de otro, se garantiza que el resultado final sería el mismo que si los hilos se ejecutasen de forma secuencial.

5. Programación asíncrona.

La programación asíncrona es una técnica que permite lanzar tareas en paralelo a la tarea principal, igual que ocurre con la programación concurrente. Pero, a diferencia de ésta, en la programación asíncrona estas tareas no compiten por ningún recurso compartido, por lo que no es necesario ningún mecanismo de sincronización entre ellas.

Dependiendo del lenguaje utilizado, éste ofrece unas u otras alternativas o mecanismos de programación asíncrona. Veremos como ejemplo algunas características de uno de los lenguajes que más facilidades ofrece al respecto, que es JavaScript, junto con otras que ofrece el lenguaje Java, que hemos venido usando como hilo conductor de los ejemplos de este tema.

5.1. Los *callbacks*.

Un *callback* es una función que se le pasa como parámetro a otra para que la ejecute cuando finalice. Es el mecanismo que proporcionan algunos lenguajes, como JavaScript, para hacer llamadas asíncronas a otras funciones, y no esperar a recoger el resultado.

Un ejemplo lo tenemos con la función *setTimeout* de Javascript. A esta función le podemos indicar una función a la que llamar, y un tiempo (en milisegundos) que esperar antes de llamarla. Ejecutada la línea de la llamada a *setTimeout* , el programa sigue su curso y cuando el tiempo expira, se llama a la función callback indicada. El siguiente ejemplo llama a *setTimeout* para que muestre el mensaje “Finalizado callback” pasados 2 segundos (2000 milisegundos), y después continúa su ejecución normal, mostrando “Hola” por la consola. Este último mensaje se mostrará nada más ejecutar el programa, y el mensaje de “Finalizado callback” se mostrará pasados los 2 segundos.

```
setTimeout(function() {console.log("Finalizado callback");}, 2000);
console.log("Hola");
```

Existen muchos otros ejemplos de uso cotidiano en JavaScript que realizan llamadas a *callbacks*. Por ejemplo, este fragmento de código utilizando el framework Node.js realiza una consulta a una base de datos MySQL utilizando la librería *mysql*. La llamada a la función *query* recibe como segundo parámetro, después de la instrucción SQL, el *callback* de respuesta. Dicho *callback* recibe como parámetro un objeto donde almacenar el error, si se ha producido, en la consulta, o el resultado obtenido si todo ha ido bien.

```
let conexion = mysql.createConnection({
     host: "servidor",
     user: "usuario",
     password: "password",
     database: "nombre_bd"
});

conexion.query("SELECT * FROM contactos",
     function (error, resultado, campos)
{
     if (error)
          console.log("Error al procesar la consulta");
     else
          console.log("Resultado:", resultado);
});
```

5.2. Promesas y futuros.

Las **promesas** son otro mecanismo importante para dotar de asincronía a Javascript. Se emplean para definir la finalización (exitosa o no) de una operación asíncrona. El resultado que se obtiene en la finalización de dicha promesa se denomina **futuro.** En nuestro código, podemos definir promesas para realizar operaciones asíncronas, o bien (más habitual) utilizar las promesas definidas por otros en el uso de sus librerías, y recoger el resultado (futuro) de manera asíncrona.

Cuando se utiliza una promesa, se deben tener en cuenta las dos posibles opciones: que se finalice correctamente o que se haya producido un error. En JavaScript, por ejemplo, esto se controla a través de las cláusulas *then* y *catch*, de forma que se ejecutará la primera si todo ha ido bien, o la segunda si ha fallado algo.

El siguiente ejemplo trata de conectar con una base de datos MongoDB, en concreto con una colección llamada *Libros*, y buscar un libro a partir de un *id* almacenado en una variable. Si todo ha ido bien, se mostrará el libro por consola. De lo contrario, se mostrará el error producido.

```
Libros.findById('5ab2dfb06cf5de1d626d5c09')
.then(resultado => {
     console.log('Resultado encontrado:', resultado);
})
.catch(error => {
     console.log('ERROR:', error);
});
```

En las últimas versiones de JavaScript (a partir de EcmaScript 7), se ha definido una especificación adicional para gestionar llamadas asíncronas. Es la especificación **async/await**, que permite, por un lado, hacer llamadas síncronas a tareas asíncronas y esperar a que finalicen para pasar al siguiente

paso. Por otro lado, permite agrupar varias de estas llamadas síncronas en una función asíncrona, para que el resultado completo del proceso no haga esperar al programa principal, o a quien llame a la función. Es un mecanismo más elegante y abreviado de enlazar llamadas asíncronas que deben esperarse consecutivamente para seguir. Por ejemplo: buscar un libro por su *id*, asociarle un autor y guardar los cambios.

En Java, por ejemplo, existen mecanismos similares para lanzar tareas asíncronas y recoger los resultados. Un ejemplo de ello es la clase *CompletableFuture*, existente a partir de Java 8. Con dicha clase, podemos definir una tarea asíncrona que devuelva un resultado, y también el método que se ejecutará cuando dicha tarea finalice.

El siguiente ejemplo crea un objeto *CompletableFuture* que devuelve un entero. Internamente, el código simplemente genera un entero aleatorio del 0 al 99 y lo devuelve pasados 3 segundos.

```
CompletableFuture<Integer> cf =
    CompletableFuture.supplyAsync(
        () -> {
            try
            {
                TimeUnit.SECONDS.sleep(3);
                return (new Random()).nextInt(100);
            } catch (InterruptedException ex) {
                return -1;
            }
    });

// Método a ejecutar cuando termine el CompletableFuture.
// en el parámetro num tendremos el número generado
cf.thenAccept((num) ->
      System.out.println("Número generado: " + num));
```

El método *thenAccept* recibe como parámetro un *callback* que se ejecutará cuando el *CompletableFuture* finalice. En este caso se han empleado expresiones lambda para definir tanto este *callback* como el código interno del *CompletableFuture*. De ahí la notación de los paréntesis y la flecha (() → {…}), aunque se podría haber implementado de igual forma, aunque con más líneas de código, utilizando interfaces y métodos tradicionales.

6. Sistemas de tiempo real cliente-servidor.

Una arquitectura cliente-servidor permite desarrollar aplicaciones divididas en dos componentes principales:

- Por un lado, está el **servidor**, que típicamente es una aplicación que está a la espera de peticiones de la otra parte (los clientes) para enviarles una respuesta.

- Por otro lado, está el **cliente**, que normalmente se conecta a un servidor y le solicita determinados recursos, o le pide realizar distintas operaciones. Por ejemplo, le puede solicitar descargar un archivo, o insertar un determinado elemento en una base de datos, entre otras cosas.

6.1. Comunicación tradicional cliente-servidor.

La comunicación típica en las aplicaciones cliente-servidor la suele iniciar el cliente: él es quien inicia la conexión con el servidor, y quien inicia las peticiones para obtener la respuesta. Así es como funcionan normalmente las aplicaciones web, que son el tipo más representativo de aplicaciones cliente-servidor, pero también otro tipo de aplicaciones que siguen esta arquitectura, como por ejemplo los juegos online multijugador.

Aunque podríamos considerar este tipo de comunicación como de "tiempo real", ya que existen unas limitaciones de tiempo de espera, fiabilidad, complejidad, etc, que hemos comentado en el punto 2 de este tema, se tiene el sesgo de que son sistemas donde la comunicación siempre la inicia uno de los dos extremos.

En cualquier caso, cualquier aplicación web que podamos encontrar nos sirve de ejemplo para ilustrar este tipo de comunicaciones, utilizando muy diversas tecnologías, entre los lenguajes y frameworks que hemos comentado anteriormente:

- En el lado del servidor, podemos emplear distintos lenguajes como PHP, JSP, ASP.NET, Python, e incluso JavaScript con su framework Node.js.

- En el lado del cliente, normalmente se emplea la tecnología JavaScript o TypeScript, que también puede ir asociada a algún framework de desarrollo del lado del cliente, como Angular o React.

6.2. Comunicación en tiempo real cliente-servidor.

Lo que no es tan habitual es encontrar un sistema de comunicación en tiempo real entre cliente y servidor donde cualquiera de las dos partes pueda iniciar dicha comunicación.

Para ello, podemos emplear algunos mecanismos de bajo nivel, como el uso de **sockets** en distintos lenguajes de programación (Java, C#, C++…). El socket establece un canal de comunicación entre los dos extremos, de forma que cualquiera de las dos partes puede enviar información a la otra.

Existen sockets donde una parte es servidora y la otra cliente, y es el cliente quien debe comenzar la conexión. Es lo que se conoce normalmente como *sockets TCP*. Pero también existen otro tipo de sockets más flexibles, donde cualquiera de las dos partes puede iniciar la comunicación y enviar datos a la

otra parte (siempre que esta otra parte esté preparada para recibirlos). En el caso de Java, por ejemplo, esto se puede conseguir a través de *sockets UDP*.

Este es un ejemplo en Java de envío y recepción de texto por parte de uno de los extremos de la comunicación, empleando sockets UDP. Como puede verse, en ningún momento se establece que tenga que ser cliente o servidor, sólo es necesario acordar el protocolo de comunicación, para determinar quién envía y quién recibe en cada momento:

```
// Enviar mensaje
String texto = "Hola";
byte[] mensaje = texto.getBytes();
DatagramPacket enviado =
     // Mensaje a enviar, tamaño, dirección destino y
puerto
     new DatagramPacket(mensaje, mensaje.length,
          InetAddress.getLocalHost(), 2000);
socket.send(enviado);

// Recibir mensaje
byte[] buffer = new byte[1024];
DatagramPacket recibido =
     new DatagramPacket(buffer, buffer.length);
socket.receive(recibido);
```

Otros lenguajes, como por ejemplo JavaScript, disponen de librerías avanzadas para gestionar estas comunicaciones en tiempo real. Un ejemplo muy representativo es la librería **socket.io**. Esta librería se puede utilizar tanto en la parte cliente como en la parte servidor, y permite definir una serie de eventos en ambas partes, de forma que, cuando se produzca dicho evento, se emitirá una respuesta. De este modo, cualquiera de los dos extremos puede emitir eventos, que serán recogidos por la otra parte para producir una respuesta.

Así podría quedar un sistema de chat básico cliente-servidor. En la parte del cliente, cada vez que el usuario quiera enviar un mensaje, se puede activar o emitir el siguiente evento:

```
// Conectar con el servidor
var socket = io('http://localhost:8080');

// Función para enviar un mensaje
function enviar() {
     var nick = document.getElementById('nick').value;
     var texto = document.getElementById('texto').value;
     if (texto != "" && nick != "")
          socket.emit('enviar', {nick: nick, texto: texto});
}
```

En la parte del servidor, se recogerá este mismo evento, y se producirá otro para difundir el mensaje al resto de miembros del chat.

```
io.on('connection', (socket) => {
    socket.on('enviar', (datos) => {
         io.emit('difundir', datos);
    });
});
```

Finalmente, de nuevo en el cliente, se atenderá a este evento emitido por el servidor para actualizar el contenido de sus ventanas de chat con el nuevo mensaje recibido.

```
socket.on('difundir', function (datos) {
    var chat = document.getElementById('chat');
    chat.innerHTML += '<p><em>' + datos.nick + '</em>: ' +
        datos.texto + '</p>';
});
```

Como puede verse, únicamente a través de los métodos *on* y *emit* se pueden capturar y emitir eventos desde los dos lados de la comunicación, en cualquier momento, una vez se ha establecido la conexión inicial. Esto permite comunicaciones bidireccionales en tiempo real, iniciadas desde cualquier extremo.

7. Conclusión.

Como hemos visto, un sistema en tiempo real (STR) es un sistema informático en el que, para que las operaciones que se realizan sean correctas, no sólo es necesario que la lógica e implementación de los programas sean correctas, sino también el tiempo en el que dichas operaciones entregan su resultado. Si las restricciones de tiempo no son respetadas.

En muchos casos, los sistemas de tiempo real se basan en la programación dirigida por eventos, de modo que el sistema se programa para quedar a la espera de que se produzca algún evento, y que dicho evento desencadene una respuesta en el sistema. Dicho evento puede producirse de forma interna o externa al sistema.

Existen distintas tecnologías y estrategias para desarrollar sistemas de tiempo real. Algunas son más tradicionales, como el uso de interrupciones para programar a bajo nivel respuestas del sistema ante pausas provocadas por agentes externos (periféricos, sistema operativo, etc). También la programación concurrente puede considerarse una técnica tradicional de programación de sistemas en tiempo real. Utilizando esta técnica, podemos generar distintos hilos o procesos que colaboren para realizar una tarea compleja, de forma que se reduce el tiempo total de finalización de dicha tarea, aunque a cambio hay que gestionar mecanismos de sincronización entre procesos o hilos para evitar que los datos que comparten se modifiquen de forma inadecuada.

En los últimos años también se han desarrollado nuevas estrategias de programación de sistemas en tiempo real. Por un lado, y gracias a lenguajes como JavaScript fundamentalmente, aunque también otros como Java, C#, etc, se han desarrollado mecanismos de programación asíncrona, alternativos a la programación concurrente tradicional, mediante los cuales podemos lanzar tareas en paralelo al programa principal y no dejar a éste "colgado" esperando a que dichas tareas finalicen. Para ello, dependiendo del lenguaje, podemos utilizar mecanismos como los *callbacks,* las promesas y/o los futuros.

Finalmente, también hemos hablado de los sistemas de tiempo real aplicados a arquitecturas cliente servidor. Además de los mecanismos tradicionales de comunicación, en los que el cliente siempre toma la iniciativa de solicitar algo al servidor, existen también algunas librerías que permiten una comunicación bidireccional, donde cualquiera de las dos partes puede enviar un mensaje o producir una respuesta de la otra, en cualquier momento. Hemos visto el ejemplo de la librería *socket.io* aplicada al lenguaje JavaScript.

Como hemos podido ver a lo largo del tema, existen multitud de lenguajes y herramientas que podemos utilizar para desarrollar sistemas de tiempo real. Desde algunas más tradicionales como C o C++, hasta otras más actuales como JavaScript o Java, pasando por distintos frameworks y librerías de apoyo para la construcción de estas aplicaciones, tales como *socket.io*, Laravel, Symfony, Angular, etc.

8. Bibliografía.

- L.M. Jiménez, R. Puerto. Sistemas informáticos en tiempo real: teoría y aplicaciones. Amazon Media.
- N. Gehani, William D. Roome. The concurrent C programming language. Silicon Press
- N. Clark. Java: Advanced features and programming techniques. Amazon Media.
- David Heron. Node.js web development. Packt Publishing
- A. Osmani. Learning JavaScript design patterns. O'Reilly