



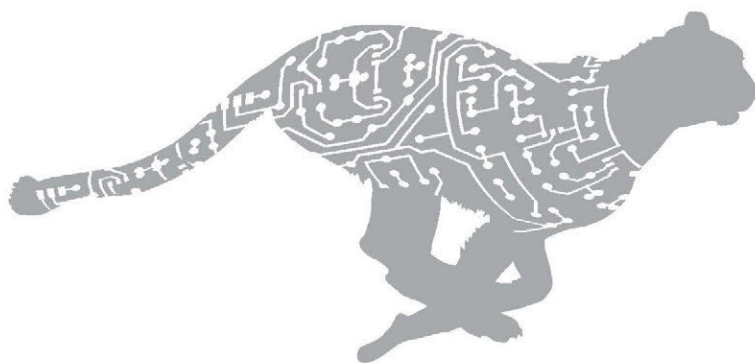
UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH



# Programación en tiempo real y bases de datos: → un enfoque práctico

Joséfinha López Herrera





UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH



iniciativa  
digital politècnica  
Publicacions Acadèmiques UPQ

→ **UPCGRAU**

Programación en tiempo real y bases de datos: →  
un enfoque práctico



Josefina López Herrera

Amb el suport de la Factoria de Recursos Docents de la UPC



Primera edició: setembre de 2011

Foto de cuberta: T.A. Rector/University of Alaska Anchorage, H. Schweiker/WIYN and NOAO/AURA/NSF.

Diseño y dibujo de la cubierta: Jordi Soldevila

Diseño maqueta interior: Jordi Soldevila

© Josefina López Herrera, 2011

© Iniciativa Digital Politècnica, 2011  
Oficina de Publicacions Acadèmiques Digitals de la UPC  
Jordi Girona Salgado 31,  
Edifici Torre Girona, D-203, 08034 Barcelona  
Tel.: 934 015 885 Fax: 934 054 101  
[www.upc.edu/idp](http://www.upc.edu/idp)  
E-mail: [info.idp@upc.edu](mailto:info.idp@upc.edu)

Depósito legal: B-33171-2011

ISBN: 978-84-7653-686-5

Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra sólo puede realizarse con la autorización de sus titulares, salvo excepción prevista en la ley.

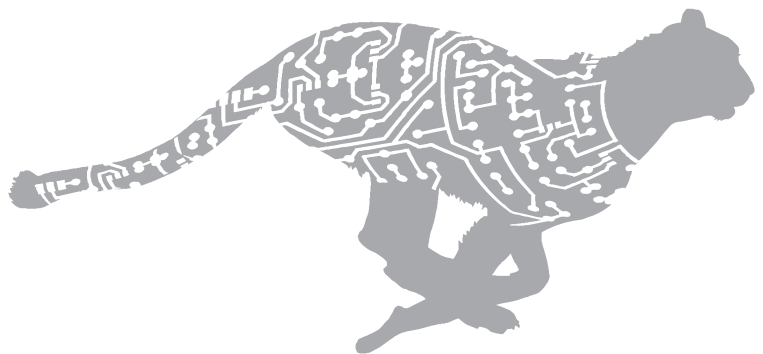


# Índice

|          |                                                                                    |           |
|----------|------------------------------------------------------------------------------------|-----------|
| <b>0</b> | <b>Índice .....</b>                                                                | <b>5</b>  |
| <b>1</b> | <b>Introducción .....</b>                                                          | <b>9</b>  |
| <b>2</b> | <b>Sistemas en tiempo real y sistemas embebidos.....</b>                           | <b>13</b> |
|          | 2.1. Introducción .....                                                            | 13        |
|          | 2.2. Programación orientada a eventos.....                                         | 16        |
| <b>3</b> | <b>Diseño de sistemas de tiempo real .....</b>                                     | <b>19</b> |
|          | 3.1. Introducción .....                                                            | 19        |
|          | 3.2. Detalle de las fases .....                                                    | 20        |
|          | 3.3. UML – Lenguaje de modelado unificado .....                                    | 22        |
|          | 3.4. Caso de estudio: captura y proceso de señales en un vuelo espacial<br>29      |           |
| <b>4</b> | <b>Proyectos: programación de eventos .....</b>                                    | <b>35</b> |
|          | 4.1. Introducción a la programación Windows con eventos .....                      | 36        |
|          | 4.2. Caso de estudio 1: Simulación del aterrizaje de un cohete .....               | 42        |
|          | 4.3. Caso de estudio 2: Definición del proyecto de control de un<br>ascensor ..... | 53        |
|          | 4.4. Caso de estudio 3: El juego del frontón.....                                  | 56        |
|          | 4.5. Caso de estudio 4: Movimiento de un cuerpo en caída libre.....                | 58        |
|          | 4.6. Problemas propuestos.....                                                     | 61        |
| <b>5</b> | <b>Las excepciones y su manejo .....</b>                                           | <b>65</b> |
|          | 5.1. Introducción .....                                                            | 65        |
|          | 5.2. Manejo de excepciones en C + + .....                                          | 67        |
|          | 5.3. Ejemplos resueltos.....                                                       | 67        |



|           |                                                                                     |            |
|-----------|-------------------------------------------------------------------------------------|------------|
| <b>6</b>  | <b>Programación concurrente .....</b>                                               | <b>71</b>  |
| 6.1.      | Introducción .....                                                                  | 71         |
| 6.2.      | Gestión de hebras ( <i>threads</i> ).....                                           | 74         |
| 6.3.      | Problemas propuestos .....                                                          | 78         |
| <b>7</b>  | <b>Sincronización .....</b>                                                         | <b>85</b>  |
| 7.1.      | Introducción .....                                                                  | 85         |
| 7.2.      | Gestión de la exclusión mutua (mutex) .....                                         | 87         |
| 7.3.      | Creación y destrucción de la exclusión mutua.....                                   | 88         |
| 7.4.      | Bloqueo y desbloqueo de la exclusión mutua .....                                    | 89         |
| 7.5.      | Ejemplo: Uso de la exclusión mutua .....                                            | 90         |
| 7.6.      | Problemas propuestos .....                                                          | 91         |
| <b>8</b>  | <b>Monitores .....</b>                                                              | <b>97</b>  |
| 8.1.      | Introducción .....                                                                  | 97         |
| 8.2.      | Exclusión mutua y variables de condición POSIX ( <i>Condition Variables</i> ) ..... | 98         |
| 8.3.      | Creación y destrucción de variables de condición.....                               | 99         |
| 8.4.      | Espera y señalización de variables de condición .....                               | 99         |
| 8.5.      | Programa de ejemplo .....                                                           | 100        |
| 8.6.      | Problemas propuestos .....                                                          | 100        |
| <b>9</b>  | <b>Semáforos .....</b>                                                              | <b>105</b> |
| 9.1.      | Introducción .....                                                                  | 105        |
| 9.2.      | Ejemplo de utilización de semáforos .....                                           | 106        |
| 9.3.      | Inicialización/destrucción de semáforos.....                                        | 106        |
| 9.4.      | Espera y señalización de semáforos.....                                             | 107        |
| 9.5.      | Programa de ejemplo de semáforos .....                                              | 108        |
| 9.6.      | Problemas presupuestos .....                                                        | 110        |
| <b>10</b> | <b>Bases de datos.....</b>                                                          | <b>119</b> |
| 10.1.     | Conceptos básicos .....                                                             | 119        |
| 10.2.     | Diseño de bases de datos .....                                                      | 123        |
| 10.3.     | MySQL: un gestor de base de datos.....                                              | 125        |
| 10.4.     | Instalación de MySQL.....                                                           | 126        |
| 10.5.     | Ejemplos sentencias MySQL.....                                                      | 127        |
| 10.6.     | MySQL en C + + .....                                                                | 131        |
| 10.7.     | Problemas propuestos .....                                                          | 134        |
| <b>11</b> | <b>Bibliografía.....</b>                                                            | <b>139</b> |



→ 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 [1]. Este libro es una guía para aprender los fundamentos de la programación en tiempo real partiendo de conocimientos previos de programación secuencial.

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 e interaccionen con su mundo físico, del cual reciben estímulos [17].

Un ordenador está formado por software y hardware. El software de los ordenadores puede dividirse de forma muy general en dos tipos: programas de sistema que controlan la operación del ordenador y programas de aplicación que realizan las tareas reales que el usuario desea. El programa de sistema base es el sistema operativo, que controla todos los recursos del ordenador y establece la base sobre la que pueden escribirse los programas de aplicación. El hardware engloba las partes materiales que constituyen un ordenador: componentes eléctricos, electrónicos, mecánicos y periféricos de todo tipo. [22]

Un ordenador moderno consta de uno o más procesadores, memoria principal (RAM- memoria de acceso aleatorio), discos, impresoras, interfaces de red y otros dispositivos de entrada/salida. La forma en que estos se construyen y sus principios de funcionamiento son objeto de estudio en el campo de la ingeniería electrónica. En algunas máquinas existen microprogramas (software primitivo), que controlan directamente estos dispositivos. Suelen estar almacenados en memoria de sólo lectura. El conjunto de instrucciones que el microprograma interpreta define el



lenguaje de máquina. Algunos ordenadores que utilizan tecnología RISC (ordenadores con un conjunto de instrucciones reducido) no tienen un nivel de microprogramación. El lenguaje de máquina tiene instrucciones que sirven para trasladar datos dentro de la máquina, realizar operaciones aritméticas y comparar valores. En este ámbito los dispositivos de entrada/salida se controlan cargando valores en registros de dispositivos especiales.

Encima del sistema operativo está el resto del software del ordenador [21]: el intérprete de comandos (shell), los sistemas de ventanas, los compiladores, los editores y otros programas similares independientes de la aplicación. El sistema operativo es la porción del software que se ejecuta en modo kernel (núcleo) o modo supervisor y está protegido por el hardware contra la intervención del usuario.

Un sistema operativo puede estudiarse desde dos puntos de vista: como administrador de recursos y como máquina extendida. Visto como administrador de recursos, su tarea consiste en administrar con eficiencia las diferentes partes del sistema, mientras que como máquina extendida deberá proporcionar a los usuarios una máquina virtual que sea más cómoda de usar que la máquina real. La parte principal de un sistema operativo es el conjunto de llamadas al sistema que puede manejar. Según la estructura del núcleo, los sistemas operativos pueden clasificarse como monolíticos (Linux, Unix, MS-DOS), microkernel (AIX, QNX), híbridos (Windows XP, XNU) y exokernel.

La misión del sistema operativo es asegurar un reparto ordenado y controlado de los procesadores, memorias y dispositivos de E/S entre los diferentes programas que compiten por ellos. Su tarea primordial consiste en seguir la pista de quién está usando qué recurso, atender solicitudes de recursos, contabilizar la utilización y mediar entre solicitudes en conflicto provenientes de diferentes programas y usuarios.

La interfaz entre el sistema operativo y los programas de usuario está definida por el conjunto de operaciones extendidas que un sistema operativo ofrece. Tradicionalmente se han denominado "llamadas al sistema". Como veremos más adelante, un concepto clave es el de "proceso". Un proceso llama a las funciones que proporciona el núcleo mediante dichas llamadas al sistema [18].

En el capítulo 2 se describen los conceptos básicos de los sistemas en tiempo real y embebidos. En el capítulo 3 se describe cómo diseñar sistemas en tiempo real. Se especifican las fases de un proyecto, los diferentes tipos de diagramas UML y se define un caso de estudio. En el capítulo 4 se describen varios proyectos que se estudian detalladamente y un conjunto de problemas propuestos. En el capítulo 5 se definen los conceptos básicos sobre manejo de excepciones. En los capítulos 6, 7, 8 y 9 se definen los elementos teóricos de programación concurrente, así como

ejemplos y problemas propuestos. En el capítulo 10 se hace una introducción sobre bases de datos y se presentan ejemplos prácticos de programación utilizando el gestor de bases de datos MySQL. En todos los ejemplos resueltos del libro se ha utilizado el entorno de programación Dev-C++ [20].

→2



# Sistemas en tiempo real y sistemas embebidos

## 2.1. Introducción

En este capítulo presentamos un breve resumen de los conceptos y características principales de los sistemas en tiempo real.

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.

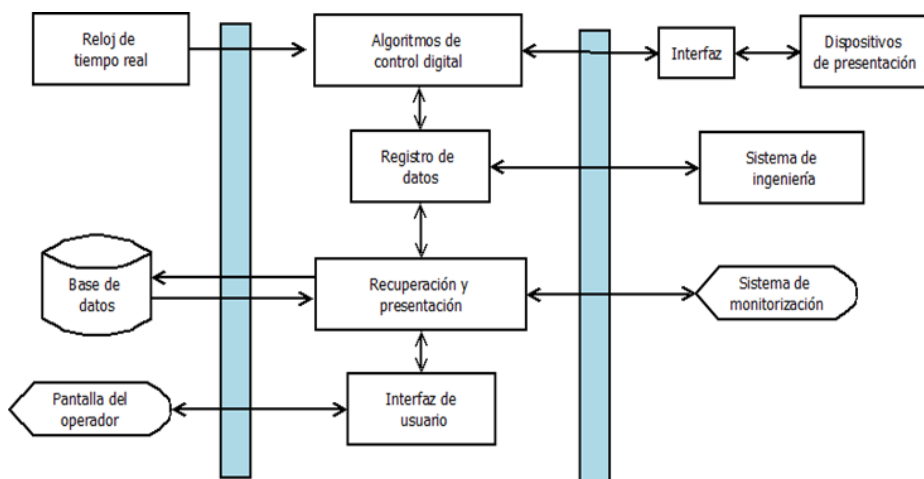


Fig. 2.1.1.  
Esquema típico de  
un sistema en  
tiempo real (STR).



Un sistema embebido o empotrado es un sistema de computación diseñado para realizar un conjunto limitado de funciones específicas, frecuentemente en un sistema informático de tiempo real [6]. Los sistemas embebidos se utilizan para usos concretos, muy diferentes a los usos generales a los que se suelen dedicar los ordenadores. En un sistema embebido la mayoría de los componentes se encuentran integrados en un soporte físico que suele incluir dispositivos usados para controlar equipos, operaciones de maquinarias o plantas industriales completas. El término “embebido” nos indica que esos circuitos son una parte integral del hardware del sistema en que se encuentran.

Un STR tiene las siguientes características principales:

- **Concurrencia:** Característica desarrollada en el capítulo 4. Con la concurrencia se descompone un programa en procesos o tareas que se ejecutan de forma simultánea. Se pueden utilizar hilos o hebras 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. En el capítulo 3 se desarrolla la técnica de manejo de excepciones para la recuperación y tratamiento de fallos de software.
- **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 la criticidad temporal (fiabilidad) de los STR, los podemos clasificar en:

- **STR crítico:** 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.
- **STR acrítico:** 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.



- **STR estricto:** El sistema es crítico y además los plazos son muy cortos.

En función de la evolución temporal de las actividades que se realizan en un STR, los podemos clasificar en:

- **STR de actividad periódica:** Las actividades se ejecutan a intervalos regulares de tiempo y deben completarse en un plazo de tiempo determinado.
- **STR de actividad aperiódica:** Estas actividades se ejecutan como respuesta a un determinado evento externo. Generalmente se especifica el tiempo mínimo entre 2 eventos, así como el plazo máximo de respuesta al mismo.

Para aclarar esto es conveniente saber que un sistema de tiempo real responde a una serie de eventos que pueden ser producidos en su interior o en su entorno, llamándose eventos internos o externos, respectivamente. En función de la capacidad del sistema para predecir su llegada podemos dividirlos también en eventos periódicos o síncronos y eventos aperiódicos o asíncronos. Los primeros son aquellos que ocurren en tiempos predecibles en el flujo de control. Al conocerse cuándo se van a producir, entonces es posible hacer una planificación anticipada de las acciones que el sistema deberealizar cuando ocurren; por ejemplo, si se sabe que cada segundo el sistema va a recibir una determinada señal. Los eventos aperiódicos o asíncronos son los que llegan de forma impredecible al sistema y dificultan por tanto el cumplimiento del plan de acción. Usualmente son causados por fuentes externas y suelen tener asociada una frecuencia promedio de llegada. Son casos en los que no existe o no se conoce una relación temporal entre dos eventos consecutivos del sistema. Por ejemplo, cuando el sistema recibe una señal de alarma que no estaba esperando.

Resumiendo:

Los sistemas de tiempo real interaccionan con su entorno y deben ejecutar sus acciones dentro de períodos de tiempo determinados.

Tienen requisitos muy exigentes en cuanto a:

- concurrencia;
- fiabilidad y seguridad;
- comportamiento temporal.

Existen varias clases de sistemas de tiempo real, con distintos requisitos temporales y de fiabilidad [7].

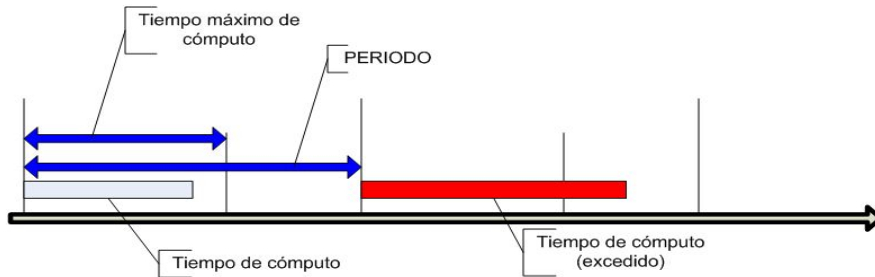


Fig. 2.1.2a:  
Esquema de un  
STR de actividad  
periódica.

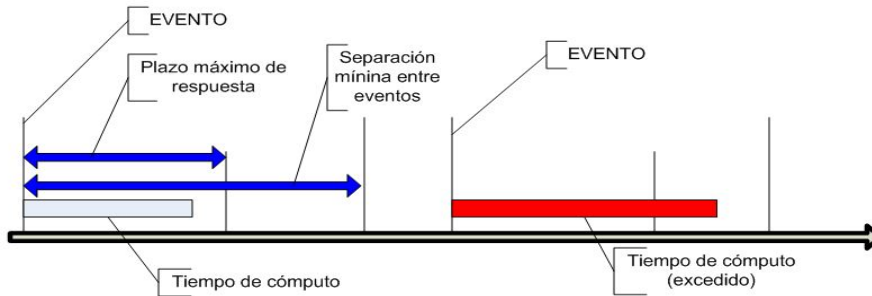


Fig.2.1.2b:  
Esquema de un  
STR de actividad  
aperiódica.

## 2.2. Programación orientada a eventos

La programación dirigida por eventos es un paradigma de programación en el que la estructura de un programa y su ejecución están definidos por los sucesos que ocurren en el sistema o que ellos mismos provocan [12].

Los sistemas en tiempo real deben responder a eventos externos o internos. De ahí que tanto la estructura como la ejecución de los programas están determinados por los sucesos que ocurran en el sistema o que ellos mismos provoquen [13].

La programación orientada a eventos permite interactuar con el usuario en cualquier momento de la ejecución [14]. Los programas creados bajo esta arquitectura están compuestos por un bucle principal en ejecución permanente, dividido claramente en dos secciones: una sección encargada de seleccionar el evento y otra sección que será la encargada del tratamiento de cada uno de los eventos detectados en la selección. Sólo se tratan aquellos eventos para los cuales se haya definido una rutina específica de tratamiento. En caso de sistemas embebidos, se puede conseguir la misma funcionalidad utilizando interrupciones en vez del bucle principal [22].



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. Cuando se produzca el evento el programa ejecutará el código del correspondiente administrador de evento.

En principio se puede utilizar cualquier lenguaje de programación para construir una arquitectura basada en eventos, aunque esa construcción es mucho más fácil cuando se utilizan lenguajes de alto nivel de abstracción.

La programación orientada a eventos es ampliamente utilizada en programas con interfaz gráfica de usuario, pues se ha adoptado como un estándar comercial para la interacción del usuario con el sistema [8]. Los lenguajes visuales orientados al evento y con manejo de componentes dan al usuario poco experto en desarrollo la posibilidad de construir sus propias aplicaciones utilizando interfaces gráficas sobre la base de ocurrencia de eventos.

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. En los ejemplos de manejo de eventos que desarrollaremos más adelante sólo utilizaremos la parte de codificación, sin ayudarnos de una herramienta específica para realizar los diseños gráficos.

→ 3



# Diseño de sistemas de tiempo real

## 3.1. Introducción

Lo más importante del desarrollo de un sistema de tiempo real [9] es la generación de un diseño consistente que satisfaga una especificación acreditada de los requisitos. La construcción de un sistema de tiempo real está formada por una serie de actividades que interactúan entre sí y con el entorno [6].

Cada una de estas actividades modela una parte del sistema que habitualmente se llama "tarea". Las tareas son una abstracción que permite simplificar la descomposición de los problemas en subproblemas. Distinguimos entre tareas de alto nivel definidas en la fase de análisis y tareas de bajo nivel o hilos de ejecución definidos en la implementación. A continuación se presentan las fases del desarrollo [15]:

- Especificación de requisitos
- Diseño arquitectónico
- Diseño detallado
- Implementación
- Prueba

Adicionalmente pueden incluirse otras actividades:

- Prototipado previo a la implementación final
- Diseño de la interfaz hombre-máquina
- Criterios para la evaluación de los lenguajes de implementación



## 3.2. Detalle de las fases

### 3.2.1. Especificación de requisitos

La fase de la especificación tiene como objetivo determinar cómo debe comportarse el sistema para cumplir con su propósito. En caso de un STR, además de las especificaciones funcionales deben indicarse los requisitos temporales que debe cumplir el sistema, esto es, las “especificaciones temporales”.

Para detallar dichas especificaciones temporales disponemos de los siguientes parámetros:

- **Tiempo de respuesta (*deadline*):** Es el tiempo máximo del que dispone una tarea para completar su ejecución en cada una de sus activaciones.
- **Retardo máximo/mínimo:** Es el tiempo máximo o mínimo que debe transcurrir antes de comenzar la ejecución de una tarea.
- **Tiempo mínimo de separación:** Es el mínimo tiempo que debe transcurrir entre dos activaciones sucesivas de una tarea. En tareas periódicas coincide con el período. .

La clave en los sistemas de tiempo real es la predictibilidad en el tiempo [7], que ha de reflejarse en cada una de las tareas especificadas. No basta con conocer el tiempo máximo que puede tardar una tarea en completarse, sino que también es necesario que éste sea muy parecido al tiempo medio de ejecución, ya que en caso contrario es difícil hacer cálculos ajustados a la realidad. La política de planificación se convierte así en un elemento crucial, ya que el cumplimiento de los límites de tiempo de todas las tareas dependerá en gran medida del orden en que éstas se ejecuten.

### 3.2.2. Diseño arquitectónico

La arquitectura de un STR debe proporcionar el soporte adecuado para la tolerancia a fallos, la posible planificación, la concurrencia y la administración del tiempo [6, 11, 15]. El sistema operativo tiene un papel muy importante en la arquitectura del sistema, ya que debe ofrecer el soporte mínimo para conseguir planificar y ejecutar las tareas o procesos de tiempo real de forma que se garantice que van a satisfacer los requisitos temporales.

### 3.2.3. Diseño detallado

De entre las diferentes metodologías o lenguajes de diseño que existen en la actualidad para diseñar sistemas de tiempo real se analizará el UML (lenguaje de modelado unificado)



UML (*Unified Modeling Language*). Es un lenguaje que sirve para especificar, visualizar, construir y documentar sistemas de software [16]. Representa una colección de las mejores prácticas de ingeniería que han sido probadas exitosamente en el modelado de grandes y complejos sistemas. El UML es un lenguaje de modelado gráfico que usa una variedad de elementos visuales para mostrar los elementos semánticos de manera que sean fácilmente aprovechables y manipulables por un modelador para especificar o describir métodos o procesos.

En UML se distinguen tres tipos diferentes de modelos [5]:

- **De requerimientos:** Ponen el énfasis en la identificación de las características del sistema más que en los objetos que lo conforman o su implementación. Para esto utilizan los “casos de uso” que presentan la funcionalidad externa del sistema.
- **Estructurales:** Permiten describir tanto la estructura lógica del sistema como la física. Para ello se utilizan los “diagramas de implementación” del UML.
- **De comportamiento:** El UML propone la utilización, entre otros, de los “diagramas de estado” y los “diagramas de actividad” para los modelos de comportamiento que definen la actuación de los procesos. Para representar la sucesión temporal se utilizan los “diagramas de secuencia”. Los diagramas de estado son útiles para representar sistemas que reaccionan a eventos. Los diagramas de secuencia permiten mostrar los mensajes y relaciones que se establecen entre los diferentes objetos del sistema.

### 3.2.4. Implementación

La implementación del sistema se refiere a la forma en que se construyen y organizan los programas para que se cumplan los objetivos y las especificaciones lógicas y temporales definidas. El lenguaje de programación debe proporcionar el soporte apropiado para desarrollar las características de un STR, ofreciendo instrucciones especiales para declarar las diferentes restricciones temporales. El lenguaje debe contar con procedimientos para el manejo y recuperación ante fallos (*exception handling*), así como facilitar el proceso concurrente, la sincronización, permitir el acceso a los dispositivos de hardware y manejar las interrupciones o eventos que se requieran. En nuestro caso utilizaremos C/C++ + POSIX [6,15] como lenguaje de programación para los ejemplos y casos de estudio expuestos en los capítulos siguientes.

### 3.2.5. Pruebas

La fase de pruebas permite verificar la calidad del software desarrollado y que se cumplen todos los requisitos especificados. Para abordar el proceso



de pruebas existen muchos planteamientos, pero para realizar pruebas en sistemas complejos es preciso analizar en detalle cada sistema, más que seguir al pie de la letra un procedimiento concreto. Existen diversas herramientas de software que permiten crear aplicaciones de pruebas en entornos de tiempo real. Nunca debe ponerse en producción un sistema que no ha sido convenientemente probado, especialmente cuando las consecuencias de un error puedan ser graves. Debemos recordar, no obstante, que las pruebas pueden demostrar la presencia de errores pero no la ausencia de ellos, ya que es imposible reproducir exactamente todas las situaciones (internas y externas) en las que se puede encontrar el sistema.

### 3.3. UML – Lenguaje de modelado unificado

Este lenguaje ofrece grandes ventajas respecto a la mayoría de lenguajes de modelado, y una de ellas es el planteamiento en la parte del diseño de una arquitectura del sistema, tanto física como lógica, que permite modelar tareas concurrentes, hilos de ejecución, plantear patrones, mostrar reutilización, así como permitir una visión dinámica del sistema. En el presente apartado presentamos con un poco más de detalle las características principales de esta metodología [23].

Como ya hemos comentado, mediante la utilización de diagramas de “casos de uso” el UML permite modelar la funcionalidad externa del sistema o partes del mismo. Mediante “diagramas de acción” permite modelar la comunicación entre las diferentes partes del sistema; mediante los “diagramas de estado” permite modelar el comportamiento de los objetos, y mediante los “diagramas de secuencia” permite modelar las restricciones temporales. Gracias al diseño de la parte dinámica del sistema podemos darnos cuenta en la fase de diseño, por ejemplo, de partes que necesitan ser sincronizadas, así como del estado de cada una de las instancias en cada momento.

Veamos a continuación algunos de los diagramas que incorpora la metodología.

#### 3.3.1. Diagrama de casos de uso

Se emplea para visualizar el comportamiento estático del sistema o una parte del mismo, de forma que se pueda conocer cómo responde dicha parte. No indica cómo están implementadas las partes que define, por lo que es un buen mecanismo para que los expertos de negocio o dominio se comuniquen con los informáticos sin llegar a niveles de complejidad. Un caso de uso especifica un requerimiento funcional, es decir indica *qué* debe hacer la parte del sistema pero no el *cómo*.



El diagrama de caso de uso de nivel superior se puede considerar como un “diagrama de contexto” y especifica las relaciones del sistema global con el “mundo exterior” y los actores que intervienen en ellas.

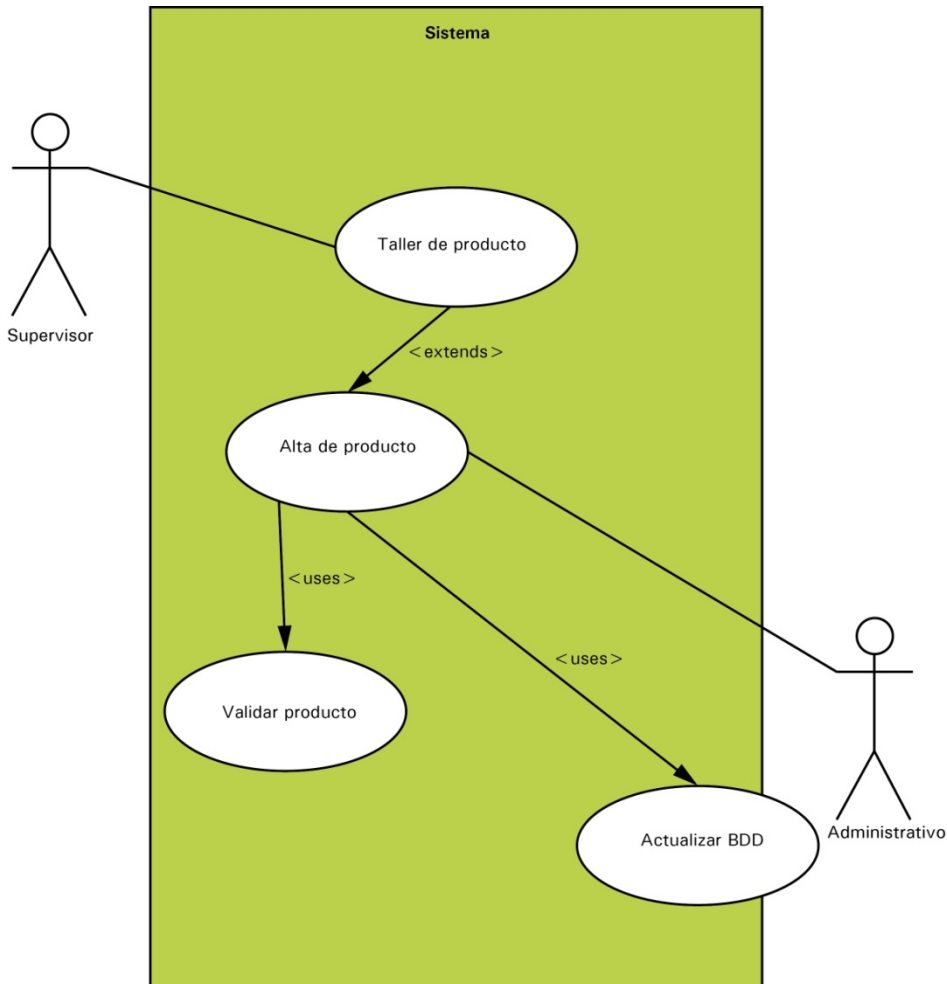


Fig 3.3.1:  
Ejemplo de  
diagrama de casos  
de uso.

En este diagrama nos encontramos con diferentes elementos:

- **Casos de uso:** Representados por una elipse con un nombre que indica su funcionalidad. Cada funcionalidad puede estar relacionada con otras mediante los siguientes símbolos:
- **Uses:** Representado por una flecha. En el ejemplo de la figura 3.3.1 podemos ver cómo la funcionalidad “Alta producto” incluye o utiliza las funcionalidades “Validar producto” y “Actualizar BDD”.



- **Extend:** Una relación de una funcionalidad A hacia un funcionalidad B indica que la funcionalidad B implementa la funcionalidad A.
- **Actores:** Se representan mediante un muñeco. Se pueden relacionar con una funcionalidad o con otros actores.
- **Límites del sistema:** Representados por un cuadrado. Representan una parte o la totalidad del sistema e incluyen las funcionalidades que lo componen.

Como complemento a los diagramas de casos de uso se utilizan documentos que describen dichos casos. En estos documentos se explica la forma de interactuar entre el sistema y el usuario. A continuación se muestra un ejemplo del documento que describe el caso de uso "Alta producto":

|                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Nombre:</b>            | <b>Alta producto</b>                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Autor:</b>             | Josefina López                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Fecha:</b>             | 24/11/2010                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Descripción:</b>       | Permite dar de alta un nuevo producto.                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Actores:</b>           | Usuario Administrativo (Dept. técnico).                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Precondiciones:</b>    | El usuario debe haberse identificado en el sistema y tener acceso a la opción de alta de producto.                                                                                                                                                                                                                                                                                                                                              |
| <b>Flujo normal:</b>      | <ul style="list-style-type: none"><li>- El usuario pulsa sobre el botón "Alta" para crear un nuevo producto.</li><li>- El sistema muestra una pantalla para seleccionar el "producto tipo" existente en el "taller de productos".</li><li>- El usuario selecciona un producto tipo de la lista y establece sus parámetros de acuerdo a las nuevas necesidades.</li><li>- El sistema comprueba la validez de los datos y los almacena.</li></ul> |
| <b>Flujo alternativo:</b> | <ul style="list-style-type: none"><li>- El sistema comprueba la validez de los datos y, si los datos no son correctos, se avisa al usuario de ello permitiéndole que los corrija</li></ul>                                                                                                                                                                                                                                                      |
| <b>Poscondiciones:</b>    | El producto ha sido almacenado en el sistema.                                                                                                                                                                                                                                                                                                                                                                                                   |



### 3.3.2. Diagrama de secuencia

El diagrama de secuencia forma parte del modelado dinámico del sistema. Muestra la secuencia de mensajes entre objetos en un escenario concreto. Los diagramas de secuencia no están pensados para mostrar lógicas de procedimientos complejos, por lo que deben representarse tantos diagramas de secuencia como llamadas diferentes haya recibido el objeto que inicia la acción.

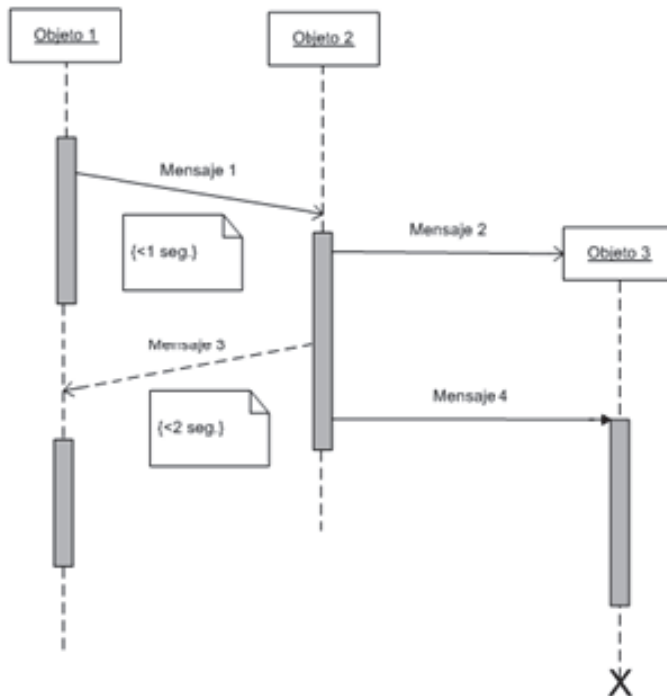
Destacamos los siguientes elementos que intervienen en este diagrama:

- **Objetos:** El diagrama se forma con los objetos que forman parte de la secuencia. Estos se sitúan en la parte superior de la pantalla, colocando a la izquierda el que inicia la acción. Un actor también puede ser un objeto de este diagrama.
- **Línea de vida:** De estos objetos sale una línea que indica su vida en el sistema y que llamaremos "línea de vida". Una línea de vida se puede crear o destruir durante la escala de tiempo representada por un diagrama de secuencia. En el último caso, la línea de vida se termina con un símbolo de detención, representado por una cruz. Cuando la línea de vida se crea con posterioridad a la llamada que inicia la acción, el objeto se representa en un nivel más bajo que los objetos iniciadores o ya existentes en el instante inicial.
- **Ocurrencia de ejecución:** La línea simple se convierte en una línea gruesa cuando representa que el objeto tiene el foco del sistema, es decir, cuando está activo.
- **Mensajes:** Los mensajes se muestran como flechas. Estos pueden ser síncronos o asíncronos (es decir, llamadas o señales). Un mensaje síncrono estará representado por una punta de flecha oscura; un mensaje asíncrono por una punta de flecha en línea. En caso de llamada la línea es continua y en caso de retorno, punteada. Cuando se modela un sistema en tiempo real, es importante considerar el tiempo que toma realizar las acciones. Al configurar una restricción de duración para un mensaje, el mensaje se mostrará como una línea inclinada.

En el ejemplo de diagrama de secuencia de la fig. 3.3.2 podemos observar los diferentes elementos gráficos comentados anteriormente.



Fig 3.3.2.  
Ejemplo de diagrama  
de secuencia.



### 3.3.3. Diagrama de estados

El diagrama de estados también forma parte del modelado dinámico del sistema. Modela el comportamiento de un solo objeto y se usa para describir el comportamiento dinámico de entidades, especificando sus respuestas a la recepción de instancias de eventos durante su tiempo de vida.

En un diagrama de estados, distinguimos los siguientes elementos:

- **Estado:** Un estado es una condición durante la vida de un objeto o de una interacción durante la cual se satisface alguna condición, se ejecuta alguna acción o se espera por algún evento. Un estado se indica mediante un rectángulo con las esquinas redondeadas, dentro del cual aparecerá nombrada esa condición o interacción. Los estados inicial y final se representan mediante puntos.
- **Transición:** Una transición significa un cambio desde un estado al siguiente y se representa con una flecha. Una transición puede tener una causa, como pueden ser una señal, un evento, una condición o el transcurso de un determinado período de tiempo, y un efecto, que es una acción que se ejecutará directamente en el objeto como resultado de la transición. Hay que tener en cuenta que no todos los eventos son válidos en todos los estados: por ejemplo, si un depósito



está vacío no se puede extraer líquido del mismo. Asimismo, una transición de estado puede tener asociada una condición de control adjunta, de manera que si no se satisface la misma no se producirá el cambio de estado.

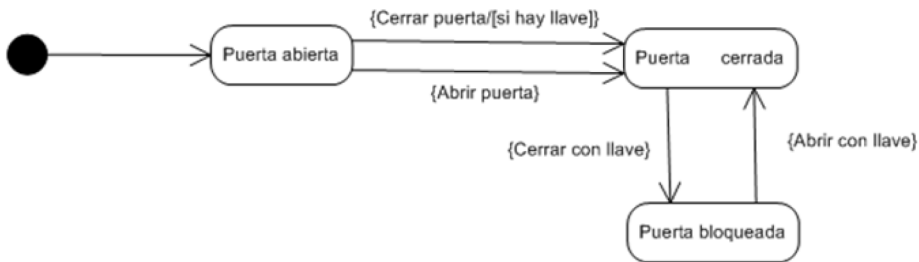


Fig. 3.3.3  
Ejemplo de diagrama  
de estados

En el ejemplo de la fig. 3.3.3 podemos observar una condición de control en la transición de puerta abierta a puerta cerrada representada por la frase “si hay llave” en el evento “Cerrar puerta”.

### 3.3.4. Diagrama de actividad

Los diagramas de actividad muestran el flujo de trabajo desde el punto de inicio hasta el punto final, detallando las rutas de decisiones que existen en el progreso de eventos contenidos en la actividad. Los diagramas de actividad son una variación del diagrama de estados cuyo propósito es enfocar la conducción de los flujos por el procesamiento interno, opuesto a los eventos externos del diagrama de estados. Estos diagramas permiten representar, entre otras, decisiones, actividades concurrentes y actividades en diferentes unidades.

En un diagrama de actividad, distinguimos los siguientes elementos:

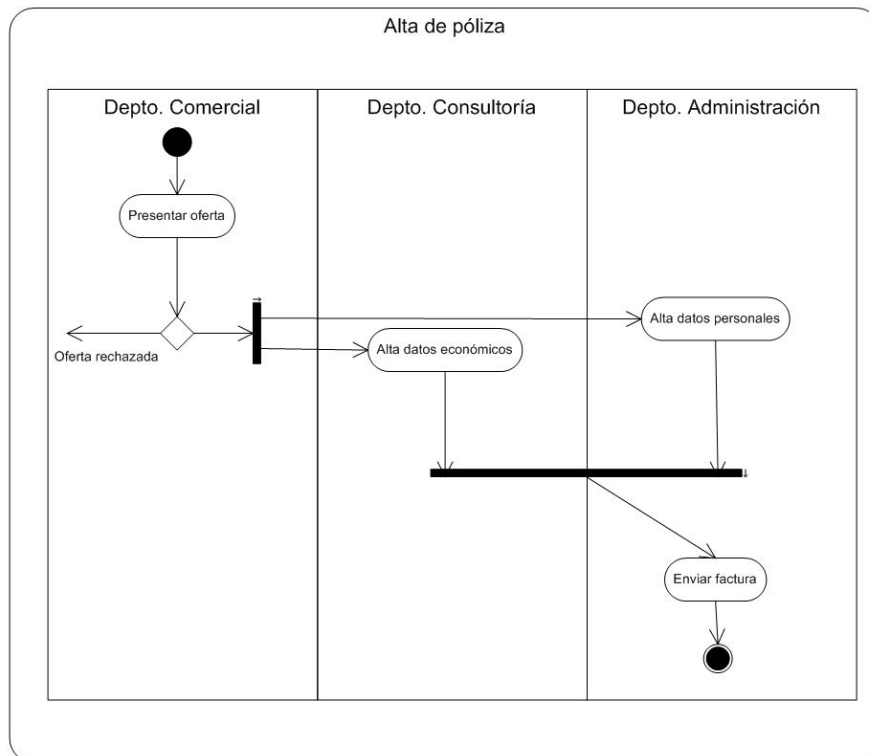
- **Actividad:** Una actividad es la especificación de una secuencia de comportamiento cuyos parámetros han sido previamente establecidos. Una actividad se muestra mediante un rectángulo con las puntas redondeadas y está compuesta por una serie de acciones, flujos de control y demás elementos que la constituyan.
- **Partición:** La partición de una actividad se muestra mediante recuadros verticales que se usan para separar acciones que son llevadas a cabo por diferentes actores (por ejemplo distintos departamentos).
- **Acción:** Una acción representa un solo paso dentro de una actividad. Las acciones se indican, igual que las actividades, mediante rectángulos con las puntas redondeadas.



- **Nodo inicial:** Significa el comienzo de la actividad. Se representa con un punto negro.
- **Nodo final:** Significa el final de todos los flujos de la actividad. Se representa con un círculo con un punto en el centro.
- **Flujo de control:** Un flujo de control muestra el paso de información de una acción a otra. Su notación es una línea con una punta de flecha.
- **Nodos de decisión y combinación:** Ambos se pueden nombrar y tienen la misma notación: una forma de diamante. Los flujos de control que provienen de un nodo de decisión tendrán condiciones que permitirán que el flujo continúe si la condición se satisface.
- **Nodos de bifurcación y unión:** Se representan mediante una barra horizontal o vertical (la orientación depende de si el flujo de control va de derecha a izquierda o de arriba abajo). Estos indican el comienzo y final de los hilos actuales de control.

El siguiente diagrama de la fig. 3.3.4 muestra un ejemplo del uso de los diferentes elementos que intervienen en un diagrama de actividad. En él se representan las diferentes acciones, flujos, particiones y nodos de la actividad “Alta de póliza”.

Fig. 3.3.4  
Ejemplo de  
diagrama de  
actividad.





En el UML existen más diagramas que los expuestos en este capítulo, pero los que hemos indicado pueden considerarse los más representativos del conjunto de diagramas que modelan el comportamiento de un sistema.

### **3.4. Caso de estudio: captura y proceso de señales en un vuelo espacial**

A continuación vamos a desarrollar con cierto detalle un proyecto de ejemplo que nos servirá para ilustrar de manera práctica la utilización de diversos elementos gráficos de modelado en UML.

El objetivo del proyecto es el diseño y programación de un software en tiempo real que permita la administración de señales a bordo de un cohete espacial. Las señales provienen de diferentes sensores, se almacenan directamente en tablas de base de datos y a continuación son tratadas por los algoritmos de control para corregir las posibles desviaciones de las trayectorias establecidas. El sistema que hay que gestionar comprende desde que un sensor envía una señal, hasta que un actuador recibe la respuesta pertinente.

#### **3.4.1. Requerimientos del proyecto**

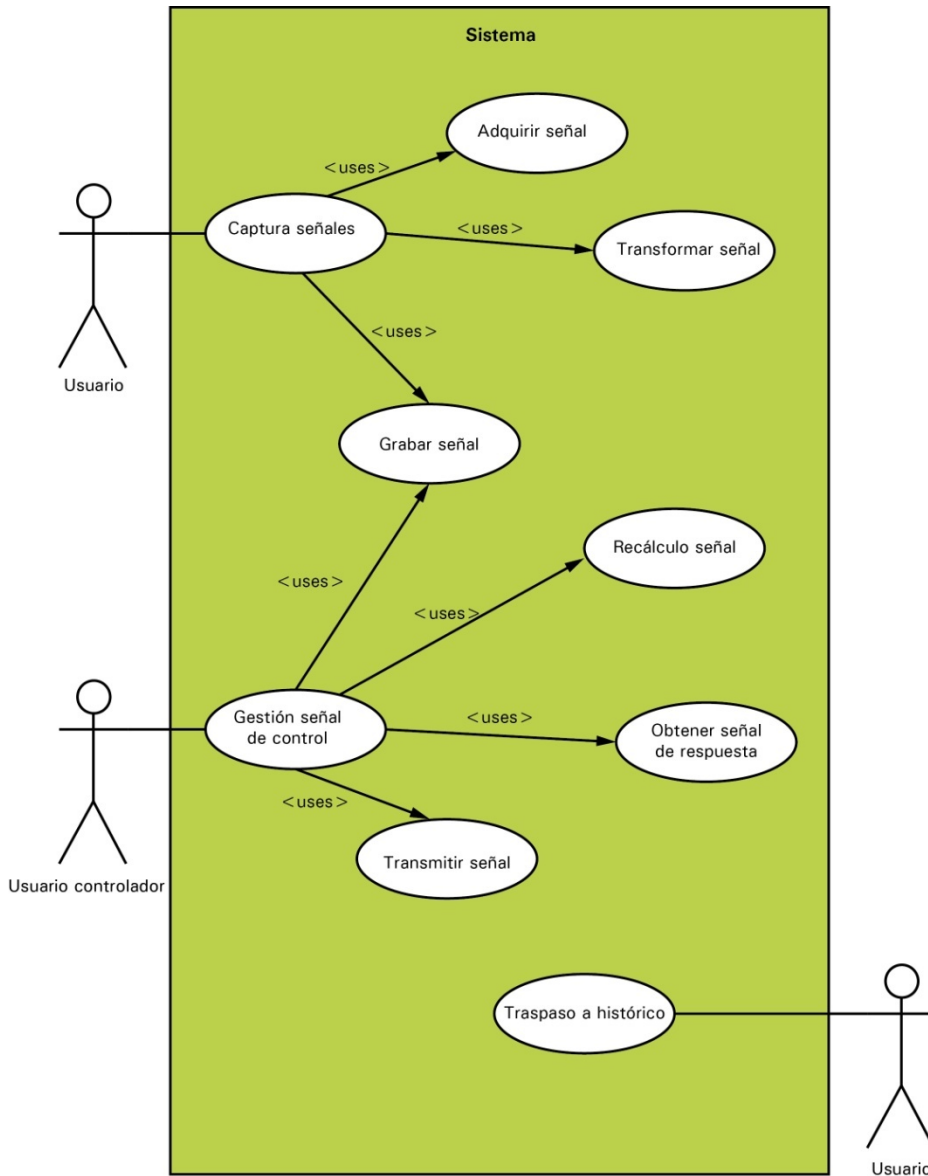
- El programa debe ejecutarse en tiempo real de manera que se cumplan en cada momento los requisitos de entrega de cada señal.
- Tanto las señales capturadas como las de control deben almacenarse en base de datos de manera que el usuario pueda saber cuándo se han almacenado y cuándo se han procesado.
- Las señales de control se calculan previamente, permitiéndose no obstante introducir correcciones una vez se ha iniciado el vuelo.
- Las correcciones introducidas por los algoritmos de control también se almacenan en base de datos.
- Las señales ya procesadas se almacenarán en unas tablas históricas para su posterior análisis.
- Deben implementarse métodos para la detección de errores para saber el lugar en el que se ha producido un error y las condiciones que lo han provocado.

#### **3.4.2. Modelado de diagramas del proyecto**

El diagrama de casos de uso del sistema sería el de la figura 3.3.5. Como se puede observar, los usuarios pueden interactuar con el subsistema de captura para iniciar el proceso y con el de señales de control para su eventual corrección.



Fig. 3.3.5  
Diagrama de casos  
de uso del proyecto  
de vuelo espacial.

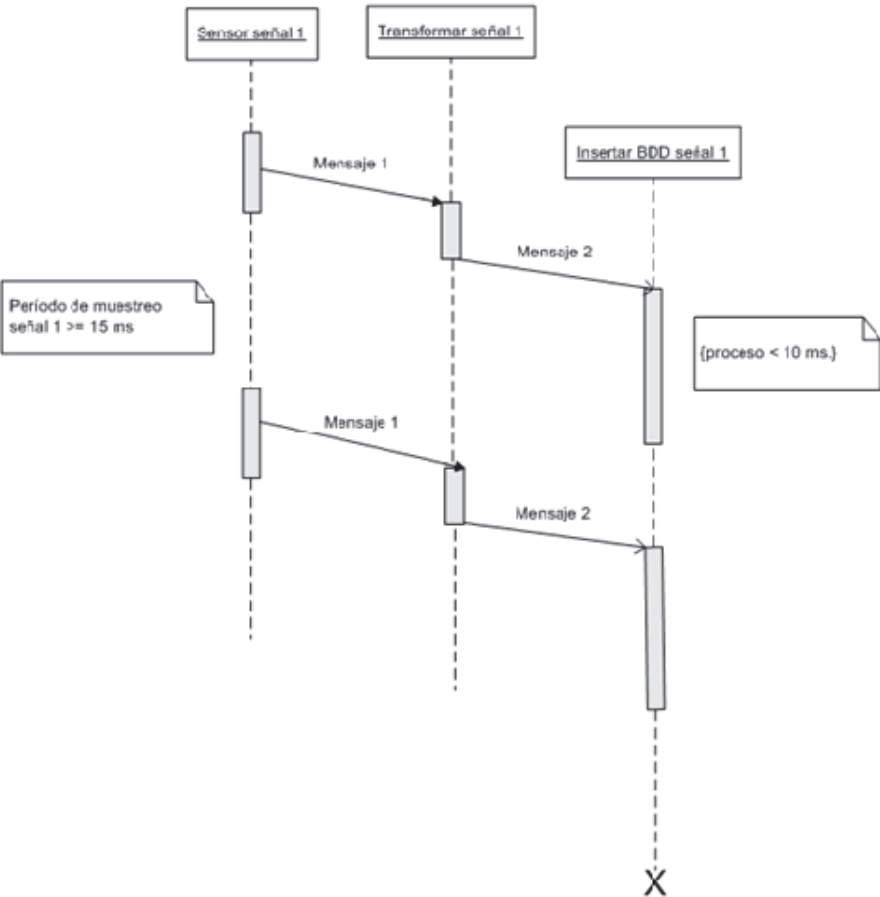


Podemos construir el diagrama de secuencia correspondiente al almacenamiento de la señal de captura en la base de datos. Para ello se procede a la lectura de la señal capturada del sensor, transformada en el formato adecuado para su inserción en las tablas de la base de datos. En el diagrama se incluyen restricciones temporales al ser un sistema de tiempo real. Este diagrama se muestra en la figura 3.3.6.

Pasemos a analizar con un poco más de detalle el caso de uso "Obtener señal de respuesta". Utilizaremos para ello el diagrama de descripción de los casos de uso.



Fig. 3.3.6  
Diagrama de secuencia  
del proyecto de vuelo  
espacial.



|         |                            |
|---------|----------------------------|
| Nombre: | Obtener señal de respuesta |
| Autor:  | Josefina López             |
| Fecha:  | 24/11/2010                 |

**Descripción:**

Este proceso se encarga de comparar las mediciones de los dispositivos de entrada con las señales de control. Si es necesario se debe rectificar la señal de salida del controlador. Las señales de control se grabarán en memoria en bloques de “r” registros consecutivos correspondientes a un determinado intervalo de tiempo. Las señales de “dispositivo” se deberán leer de la BDD mediante cursores a intervalos inferiores al período de muestreo, almacenando también la información en memoria. Cada registro tendrá una referencia para calcular el instante de tiempo que le corresponde. La zona de memoria de la señal de control será inspeccionada continuamente para calcular las correcciones que se deban aplicar.



**Actores:**

Usuario controlador.

---

**Precondiciones:**

Las señales de control deben haberse grabado previamente en la BDD.

Debe haberse iniciado la trayectoria del cohete y con ello la captura de las señales de los dispositivos.

---

**Flujo normal:**

- Al inicio del proceso o cada vez que el búfer está "lleno", la "hebra de gestión del búfer" lee "n" registros del siguiente cursor --> `select * from tabla_control_tipo_nn where tiempo_relativo >= ultr_nn` (`ultr_nn` es una variable global que indica el último valor almacenado en el campo "tiempo\_relativo" de la tabla de "captura" correspondiente al tipo de señal "nn", que inicialmente está a 0), y se almacena en el búfer.
- La "hebra de gestión del búfer" lee todos los nuevos registros insertados en la tabla de señal de captura (`select * from tabla_captura_tipo_nn where tiempo_relativo >= ultr_nn`) y los guarda en el búfer.
- Cuando el búfer está "lleno", elimina los elementos ya tratados y los reordena. Consideramos que está lleno cuando se ha grabado la señal de captura "n" en el búfer. Debe haber una variable global (por tipo de señal) que nos indique si el búfer está lleno. Se define un búfer por cada tipo de señal.
- Mientras tanto la "hebra de tratamiento" irá explorando los elementos no tratados del búfer para ir aplicando los algoritmos correspondientes (por tipo de señal). Ha de haber una variable global por tipo de señal que indique el "siguiente elemento" del búfer que debe tratarse. Esta variable la debe actualizar la rutina de "reordenación del búfer". La actualización de variables globales y del búfer debe protegerse mediante exclusión mutua.

---

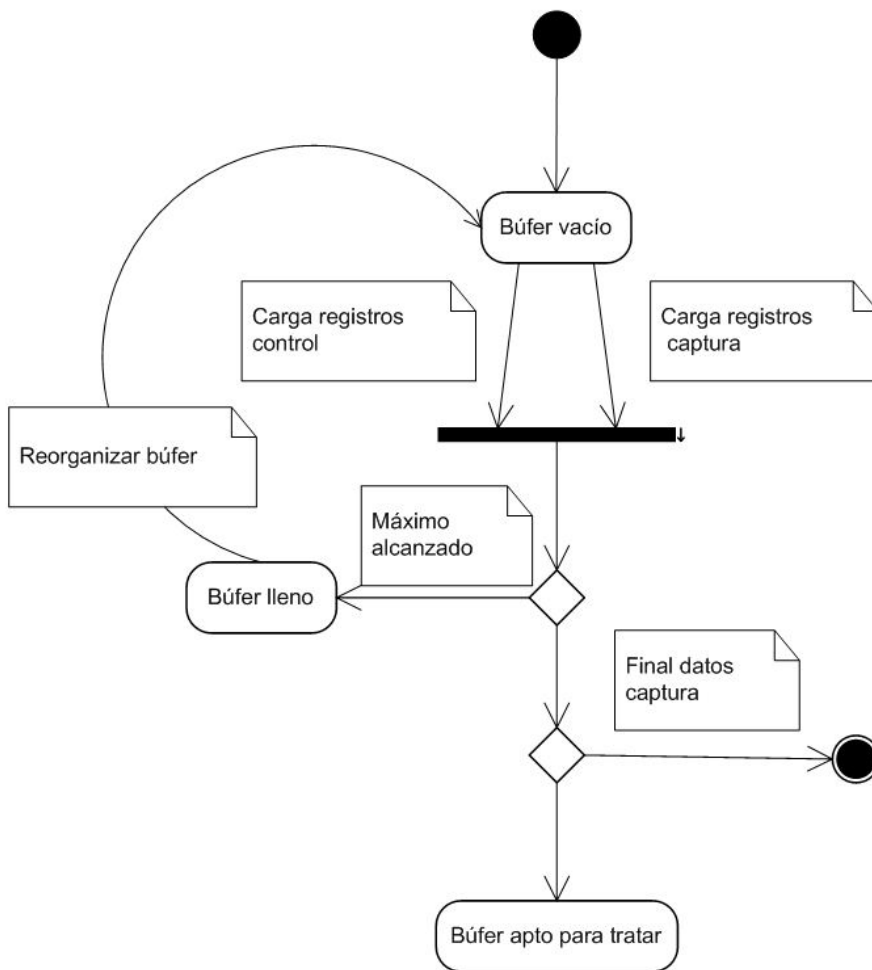
**Flujo alternativo:**

- El sistema comprueba la consistencia de los datos, y, si los datos no son consistentes, se graba un registro de excepción indicando los datos y el período de tiempo no tratados correctamente.
  - Si se detecta alguna excepción de acceso a datos se graba un registro de excepción indicando el tipo de acceso, la tabla afectada y el período de tiempo en que ha ocurrido.
- 

A partir de la descripción detallada del caso de uso anterior podemos representar, en la figura 3.3.7, el diagrama de estados correspondiente al búfer de memoria utilizado para el intercambio de información entre la "hebra de captura" y la "hebra de tratamiento".



Fig. 3.3.7  
Diagrama de estados  
del búfer del proyecto  
de vuelo espacial.



→ 4



## Proyectos:

# Programación de eventos

En este capítulo se describen proyectos para desarrollar utilizando la programación de eventos y se proponen problemas que implementar. Se proporciona una implementación de los componentes gráficos y del bucle principal de mensajes de los proyectos por desarrollar, y faltan únicamente determinadas funciones específicas de tratamiento de eventos.

La programación dirigida por eventos se basa en el patrón estructural llamado patrón *Handler* (manejador), el cual tiene su origen en los diagramas de análisis de transacciones definidos por De Marco, Yourdon y Constantine. Algunas metodologías para construir sistemas dirigidos por eventos utilizan métodos estructurados para coordinar los componentes del sistema, como las de Fiege, Mühl y Gärtner [14]. El manejo de excepciones e hilos son mecanismos que están presentes en la programación dirigida por eventos, aunque en este capítulo nos centraremos en el tratamiento no concurrente de los procesos asociados a cada evento.

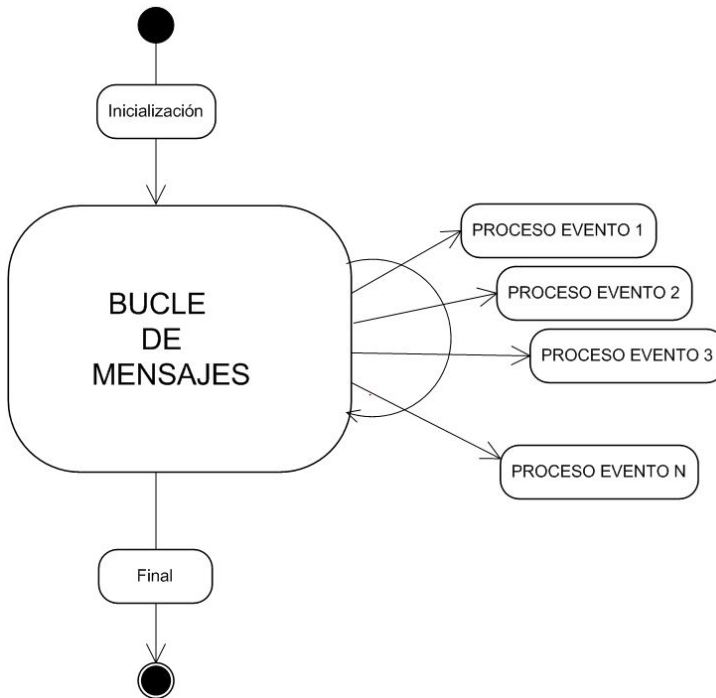
En el diseño de sistemas implementados con programación dirigida por eventos se utilizan metodologías estructuradas como las de Jackson y Yourdon [15]. Utilizando el diseño descendente se pueden definir las funciones/acciones que gestionarán los diferentes mensajes para solucionar las peticiones al sistema producidas por los eventos externos. En la Figura 4.1 mostramos un esquema con esta estructura.

Para mostrar cómo se implementa un sistema dirigido por eventos utilizaremos como plataforma de desarrollo la programación Windows. Los



programas en Windows están orientados a eventos, lo que significa que, normalmente, están esperando a que se produzca un acontecimiento que les incumba y, mientras tanto, permanecen aletargados o dormidos.

Fig. 4.1  
Esquema típico de  
un bucle de  
tratamiento de  
mensajes



Un evento puede ser, por ejemplo, el movimiento del ratón, la activación de un menú, la llegada de información desde el puerto serie, la pulsación de una tecla...

Esto es así porque Windows es un sistema operativo multitarea, y el tiempo del microprocesador ha de repartirse entre todos los programas que se estén ejecutando.

#### 4.1. Introducción a la programación Windows con eventos

El sistema operativo Windows permite diseñar sistemas dirigidos por eventos con las siguientes características:

- Los dispositivos de entrada y salida se consideran como recursos.
- Como interfaz entre usuario y aplicación se utiliza un área rectangular de la pantalla llamada ventana. Cada aplicación tiene una ventana principal. Sólo una ventana puede estar activa para recibir información del usuario.



- Los controles son la forma en que las aplicaciones Windows intercambian datos con el usuario. Normalmente se usan dentro de los cuadros de diálogo, pero en realidad pueden usarse en cualquier ventana.
- Los programas están organizados en un bucle principal de recepción de mensajes (eventos) y un conjunto de procesos que gestionan dichos mensajes, como se muestra en la Figura 4.1.

Para diseñar un programa C++ en Windows dividiremos en varios ficheros fuentes los programas y los integraremos en un proyecto.

Para crear un proyecto Windows usando el compilador DevC++ elegiremos el menú "File/New Project". Se abrirá un cuadro de diálogo donde podremos elegir el tipo de proyecto. Elegiremos "Windows Application" y "C++ Project". A continuación pulsamos "Aceptar".

Una ventana está formada por varios componentes. En la Fig 4.1.2 podemos ver un ejemplo:

1. Borde de la ventana.
2. Barra de título. Zona en la parte superior de la ventana que contiene el icono y el título de la ventana. Esta zona también se usa para mover la ventana a través de la pantalla y, mediante doble clic, para cambiar entre el modo maximizado y el tamaño normal.
3. Caja de Minimizar.
4. Caja de Maximizar.
5. Caja de Cerrar.
6. Caja de Control menú.
7. Es la zona donde el programador sitúa los controles y los datos para el usuario. En general abarca toda la superficie de la ventana que no está ocupada por las zonas anteriores.
8. Control (barra de desplazamiento vertical).
9. Control (botón de comando).

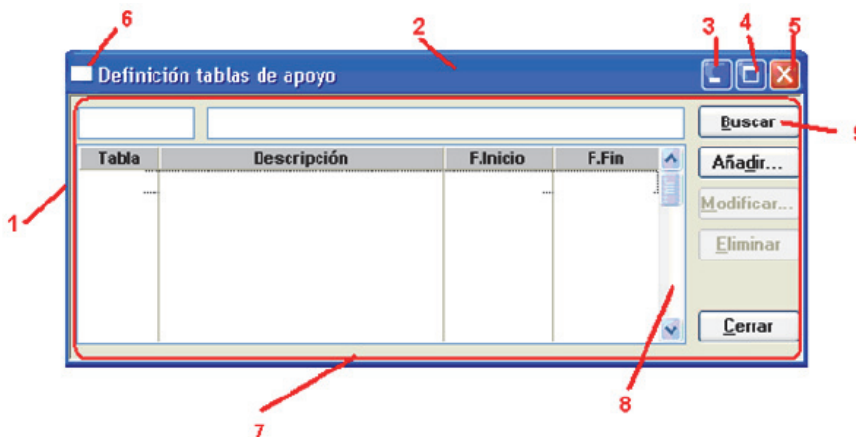


Fig 4.1.2  
Ejemplo de  
ventana.



```
#include <windows.h>
// Prototipos:
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM,
LPARAM);
// Función de entrada:
int WINAPI WinMain (HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR lpszCmdParam,
int nCmdShow)
{
// Declaración:
// Inicialización:
// Bucle de mensajes:
return Message.WParam;
}
// Definición de funciones:
```

La función de entrada de un programa Windows es “WinMain”, en lugar de la conocida “main”. La definición de esta función cambia muy poco entre las aplicaciones. Se divide en tres partes claramente diferenciadas:

- Declaración
- Inicialización
- Bucle de mensajes

La función WinMain tiene cuatro parámetros de entrada:

- hInstance: Es un manipulador para la instancia del programa que estamos ejecutando. Cada vez que se ejecuta una aplicación, Windows crea una instancia para ella, y le pasa un manipulador de dicha instancia a la aplicación.
- hPrevInstance: Es un manipulador a instancias previas de la misma aplicación. Como Windows es multitarea, pueden existir varias versiones de la misma aplicación ejecutándose, varias instancias.
- lpszCmdParam: Esta cadena contiene los argumentos de entrada del comando de línea.
- nCmdShow: Este parámetro especifica cómo se mostrará la ventana.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
LPSTR lpszCmdParam, int nCmdShow)
{
/* Declaración: */
HWND hwnd;
MSG mensaje;
```

```

WNDCLASSEX wincl;
/* Inicialización: */
/* Estructura de la ventana */
wincl.hInstance = hInstance;
wincl.lpszClassName = " CLASSNAME";
wincl.lpfnWndProc = WindowProcedure;
wincl.style = CS_DBLCLKS;
wincl.cbSize = sizeof (WNDCLASSEX);
/* Usar icono y puntero por defecto */
wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
wincl.lpszMenuName = NULL;
wincl.cbClsExtra = 0;
wincl.cbWndExtra = 0;
wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;
/* Registrar la clase de ventana, si falla, salir del
programa */
if(!RegisterClassEx(&wincl)) return 0;
hwnd = CreateWindowEx(
0, " CLASSNAME", "Ejemplo Ventana", WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT, 544, 375, HWND_DESKTOP,
NULL, hThisInstance, NULL
);
ShowWindow(hwnd, SW_SHOWDEFAULT);
/* Bucle de mensajes: */
while(TRUE == GetMessage(&mensaje, 0, 0, 0))
{
TranslateMessage(&mensaje);
DispatchMessage(&mensaje);
}
return mensaje.wParam;
}

```

#### 4.1.2. Declaración

En la primera zona declararemos las variables que necesitamos para nuestra función WinMain, que como mínimo serán tres:

- HWND hwnd: Un manipulador para la ventana principal de la aplicación. Ya sabemos que nuestra aplicación necesitará al menos una ventana.
- MSG Message: Una variable para manipular los mensajes que lleguen a nuestra aplicación.
- WNDCLASSEX wincl: Una estructura que se usará para registrar la clase particular de ventana que usaremos en nuestra aplicación. Existe otra estructura para registrar clases que se usaba antiguamente, pero que ha sido desplazada por esta nueva versión, se trata de WNDCLASS.



#### 4.1.3. Inicialización

Esta zona se encarga de registrar la clase o clases de ventana, crear la ventana y visualizarla en pantalla.

Para registrar la clase primero hay que rellenar adecuadamente la estructura `WNDCLASSEX`, que define algunas características que serán comunes a todas las ventanas de una misma clase, como el color de fondo, el icono, el menú por defecto, el procedimiento de ventana, etc. Después hay que llamar a la función `RegisterClassEx`.

En el caso de usar una estructura `WNDCLASS` se debe registrar la clase usando la función `RegisterClass`.

A continuación se crea la ventana usando la función `CreateWindowEx`. Para que la ventana sea visible hay que llamar a la función `ShowWindow`.

#### 4.1.4. Bucle de mensajes

Éste es el núcleo de la aplicación. Como se ve en el ejemplo, el programa permanece en este bucle mientras la función `GetMessage` retorne con un valor `TRUE`.

```
while(TRUE == GetMessage(&mensaje, 0, 0, 0)) {  
    TranslateMessage(&mensaje);  
    DispatchMessage(&mensaje);  
}
```

La función `GetMessage` puede retornar con tres valores: `TRUE`, `FALSE` ó `-1`. El valor `-1` indica un error, así que en este caso se debería abandonar el bucle.

El bucle de mensajes que encontraremos habitualmente es el siguiente:

```
while(GetMessage(&mensaje, 0, 0, 0)) {  
    TranslateMessage(&mensaje);  
    DispatchMessage(&mensaje);  
}
```

La función `TranslateMessage` se usa para traducir los mensajes de teclas virtuales a mensajes de carácter.

Los mensajes traducidos se reenvían a la lista de mensajes del proceso y se recuperarán con las siguientes llamadas a `GetMessage`.

La función `DispatchMessage` envía el mensaje al procedimiento de ventana, donde será tratado adecuadamente.



#### 4.1.5. Definición de funciones

En esta parte definiremos, entre otras cosas, los procedimientos de ventana que se encargan de procesar los mensajes que lleguen a cada ventana.

#### 4.1.6. Procedimiento de ventana

Cada ventana tiene una función asociada. Esta función se conoce como procedimiento de ventana, y es la encargada de procesar adecuadamente todos los mensajes enviados a una determinada clase de ventana. Es la responsable de todo lo relativo al aspecto y al comportamiento de una ventana.

Normalmente, estas funciones están basadas en una estructura *“switch”* donde cada *“case”* corresponde a un determinado tipo de mensaje.

*Sintaxis:*

```
LRESULT CALLBACK WindowProcedure(
    HWND hwnd, // Manipulador de ventana
    UINT msg, // Mensaje
    WPARAM wParam, // Parámetro palabra, varía
    LPARAM lParam // Parámetro doble palabra, varía
);
```

- hwnd es el manipulador de la ventana a la que está destinado el mensaje.
- lmsg es el código del mensaje.
- lParam es el parámetro de tipo palabra asociado al mensaje.
- lParam es el parámetro de tipo doble palabra asociado al mensaje.

Podemos considerar este prototipo como una plantilla para crear nuestros propios procedimientos de ventana. El nombre de la función puede cambiar, pero el valor de retorno y los parámetros deben ser los mismos. El miembro lpfnWndProc de la estructura WNDCLASS es un puntero a una función de este tipo, y esa función es la que se encargará de procesar todos los mensajes para esa clase de ventana. Cuando registremos nuestra clase de ventana, tendremos que asignar a ese miembro el puntero a nuestro procedimiento de ventana.

*Prototipo de procedimiento de ventana:*

```
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM,
    LPARAM);
```

*Implementación de procedimiento de ventana simple:*

```

/* Esta función es llamada por la función del API
DispatchMessage() */
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg,
WPARAM wParam, LPARAM lParam)
{
switch (msg) /* manipulador del mensaje */
{
case WM_DESTROY:
PostQuitMessage(0); /* envía un mensaje WM_QUIT a la cola
de mensajes */
break;
default: /* para los mensajes de los que no nos ocupamos */
return DefWindowProc(hwnd, msg, wParam, lParam);
}
return 0;
}

```

**4.2. Caso de estudio 1: Simulación del aterrizaje de un cohete****4.2.1. Fundamentos teóricos**

Un cohete disminuye su masa con el tiempo, como consecuencia del consumo de combustible, para lograr aumentar su velocidad. Se trata de un sistema de masa variable. En la descripción del movimiento de un cohete no puede emplearse la segunda ley de Newton  $F=ma$ , sino la definición general de fuerza:

$$\frac{d\vec{p}}{dt} = \vec{F}$$

Sea  $v$  la velocidad del cohete respecto al planeta, y  $u$  la velocidad constante de los gases expulsados respecto al cohete;  $v-u$  será la velocidad de los gases respecto al planeta. Suponemos que la cantidad de combustible quemado en la unidad de tiempo,  $D$ , es constante,  $D=dm/dt$ .

La masa  $m$  del cohete en el instante  $t$  valdrá  $m=m_0-Dt$ , donde  $m_0$  es la suma de la carga útil más el combustible inicial, y  $D_t$  es el combustible quemado al cabo de un cierto tiempo  $t$ .

Cuando el cohete expulsa una cantidad de combustible  $dm$ , incrementa su velocidad en  $dv$ . La variación del momento lineal será igual al momento lineal del cohete más el momento lineal de los gases expulsados en el instante  $t+dt$ , menos el momento lineal del cohete en el instante  $t$ .

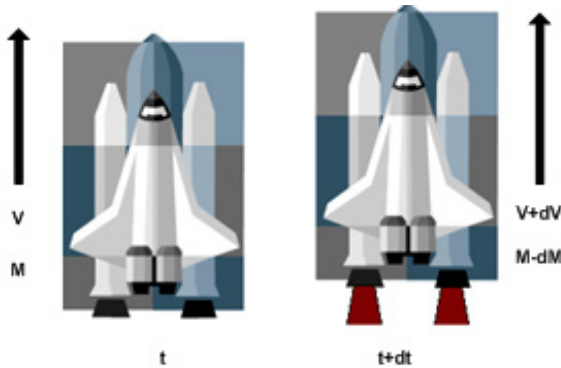
$$dp=(m-dm)(v+dv)+(v-u)dm-mv$$

Simplificando y despreciando infinitésimos de orden superior queda:



$$dp = m dv - u dm$$

La razón del cambio del momento lineal con el tiempo será entonces



$$\frac{dp}{dt} = m \frac{dv}{dt} - u \frac{dm}{dt}$$

La derivada del momento lineal con el tiempo es igual a la fuerza que actúa sobre el cohete  $F = -mg$ , donde  $g$  es la intensidad del campo gravitatorio cerca de la superficie del planeta que supondremos constante. Por tanto,

$$m \frac{dv}{dt} = uD - mg$$

Esta expresión se puede interpretar del siguiente modo: un cohete puede considerarse un móvil de masa variable  $m$  sometido a dos fuerzas en la misma dirección y en sentidos contrarios, el empuje de los gases  $uD$  y el peso  $mg$ .

La ecuación anterior la podemos escribir

$$\frac{dv}{dt} = -g + u \frac{D}{m_0 - Dt}$$

O, expresado de otra manera:

$$dv = \left( -g + u \frac{D}{m_0 - Dt} \right) * dt$$

Con esta ecuación podemos calcular la variación de la velocidad en cada intervalo infinitesimal de tiempo. Si consideramos intervalos de tiempo  $\Delta t$  suficientemente pequeños (supongamos por ejemplo  $\Delta t = 0.05$  seg.) y no tenemos en cuenta el efecto del rozamiento, podemos considerar que durante el intervalo la masa se mantiene constante, y podemos utilizar la siguiente función para calcular la velocidad final al cabo de un tiempo  $t$ :



Suponemos que la masa total inicial (MASAT\_INI), la masa inicial del combustible (MASAC\_INI), la velocidad de los gases (u), y el intervalo de tiempo  $\Delta t$  (INTERVAL) son unas constantes definidas en el programa.

```
double fVelocidad (double v0, double D, double& Dt, double
g, double t, BOOL swEncendido) {
double tf=0;
double v=v0;
if (swEncendido) Dt=Dt+D*INTERVAL; // Consume combustible
con motor encendido.
while (tf<t) {
tf=tf+INTERVAL;
v=v+fVelocidadInterval(D, Dt, g, swEncendido);
}
return v;
}
double fVelocidadInterval (double D, double Dt, double g,
BOOL swEncendido) {
double wD;
if (swEncendido) wD=D;
else wD=0; // con el motor apagado no hay empuje
return (-g + ((u*wD)/( MASAT_INI - Dt))) * INTERVAL;
}
```

NOTA: Si se llama a la función con alguno de los parámetros u ó D a 0, retorna la velocidad de caída libre  $v=g*t$ .

#### 4.2.2. Definición del proyecto

Se trata de simular el aterrizaje de un cohete en la superficie de un planeta. El cohete parte de una posición inicial a una altura determinada del suelo y, mediante los controles y variables a nuestra disposición, deberemos maniobrar para que al llegar a la superficie tenga una determinada velocidad de descenso. En caso de que al llegar al suelo se supere esta velocidad el cohete se “estrella”. Una velocidad negativa significa que desciende, mientras que una velocidad positiva significa que asciende.

La simulación finaliza cuando se produce una de las tres situaciones siguientes:

1. El cohete alcanza la superficie (\*) con una velocidad de descenso inferior a 0.1 m/s o bien cambia de sentido de movimiento en la franja considerada como superficie.
2. El cohete alcanza el suelo con una velocidad de descenso superior a 0.1 m/s.
3. El cohete alcanza una distancia del suelo superior a la altura inicial (3.200 m).

En el primer caso consideramos que el aterrizaje es correcto, en el segundo, el cohete se estrella y en el tercero el cohete “se pierde”.



Con el motor apagado, el cohete se desplaza en caída libre (consideramos despreciable el efecto de rozamiento).

(\*) Para representar la superficie consideraremos una franja de 100 m de altura desde el suelo (es decir, cualquier punto dentro de esta franja lo consideraremos la superficie).

Disponemos de los siguientes controles y variables:

#### Controles:

- Encendido/apagado del motor (sólo gasta combustible con el motor encendido).
- Inicio (volver a realizar la simulación).
- Parar/continuar.

#### Variables:

- Fuerza de empuje ( $u \cdot D$ ). Consideramos que  $u$  es constante y lo que varía es la cantidad  $D$  (combustible quemado por segundo). El valor inicial de  $u \cdot D$  es de 50.000 Newtons, lo que significa que la pérdida inicial de combustible por unidad de tiempo  $D$  es de 50 Kg/s (ya que  $u = 1.000 \text{ m/s}$ ).
- Valor de la gravedad (lista de 3 valores seleccionables):
  - $11,18 \text{ m/s}^2$  (Saturno);
  - $9,81 \text{ m/s}^2$  (Tierra);
  - $3,87 \text{ m/s}^2$  (Marte).

Adicionalmente debemos controlar que el cohete no se quede sin combustible ya que, en caso que éste se agote, la fuerza de empuje será 0 (caída libre).

Las siguientes magnitudes estarán fijadas por programa:

- Altura inicial del cohete "DIST\_INI" (3.200 m).
- Masa inicial de combustible "MASAC\_INI" (10.000 Kg).
- Velocidad de escape de los gases " $u$ " (1.000 m/s).
- Masa inicial del cohete "MASAT\_INI" (carga útil más combustible inicial) 15.000 kg.
- La velocidad inicial del cohete  $v_0$  (0 m/s).
- Intervalo de tiempo "INTERVAL" (0,05 s).



#### 4.2.3. Interfaz gráfica

En este apartado describimos la interfaz gráfica disponible para realizar el proyecto:

- Representación del cohete mediante una figura cilíndrica. Con el motor encendido aparece una zona de color rojo.
- Escala vertical en divisiones de 100 m en la que se representa la distancia hasta la superficie.
- Representación del porcentaje de combustible restante mediante una figura rectangular de color azul. La parte azul representa el combustible que queda en el tanque.
- Representación de la superficie del planeta mediante una franja horizontal de 100 m.
- Visualizador que muestra la velocidad actual del cohete. Si desciende es negativa, si asciende es positiva. Inicialmente a 0.
- Visualizador que muestra la distancia actual en metros entre el cohete y la superficie. Inicialmente a 3.200 m.
- Visualizador que muestra la fuerza total sobre el cohete en cada instante de tiempo (Peso – Empuje).
- Visualizador que muestra el valor de la fuerza de empuje. Inicialmente a 10.000 N.
- Visualizador que muestra el tiempo total transcurrido en segundos desde el inicio de la maniobra.
- Visualizador que indica si la simulación ha finalizado con éxito o no (es decir, si el cohete se estrella o no).
- Control de motor encendido/apagado. Botón de tipo bitmap con una imagen de un rayo para encender y una imagen de un aspa para apagar.
- Control de parar/continuar.
- Control Inicio de simulación.
- Lista para seleccionar la gravedad. Al cambiar el valor se reinicia la simulación.
- Control para aumentar en 1.000 unidades el valor del empuje.
- Control para aumentar en 100 unidades el valor del empuje.
- Control para disminuir en 1.000 unidades el valor del empuje.
- Control para disminuir en 100 unidades el valor del empuje.



Al finalizar la simulación, tanto si el aterrizaje ha sido correcto como si no, se presentan textos (visualizadores) en pantalla con las magnitudes estadísticas.

El fichero aterrizaje.dev incluye todos los subprogramas necesarios para desarrollar las funciones del programa desarrollo.cpp.

Los archivos proporcionados son los siguientes:

| ARCHIVO               | DESCRIPCIÓN                                                                                                                                                                                                                                                                                                                |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| aterrizaje.dev        | Archivo de proyecto para poder compilar y ejecutar todos los subprogramas del proyecto.                                                                                                                                                                                                                                    |
| aterrizaje.h<br>IDS.h | Contienen la definición de los tipos necesarios para el funcionamiento del programa principal (aterrizaje.cpp). Estos ficheros no deben ser modificados.                                                                                                                                                                   |
| aterrizaje.cpp        | Contiene el código del programa principal. Incluye la función principal (que en este caso se llama WinMain al tratarse de un proyecto con ventanas para Windows), el bucle principal de mensajes y todas las funciones que se necesitan para representar gráficamente el cohete. Éste fichero tampoco debe ser modificado. |
| *.rc, *.res y *.bmp   | Ficheros de recursos propios del proyecto. No deben ser modificados.                                                                                                                                                                                                                                                       |
| desarrollo.h          | En este archivo se debe incluir el diseño de la estructura de datos necesaria para el cálculo de las estadísticas. Inicialmente contiene tan sólo el nombre de la estructura y queda como tarea incluir todos los tipos de datos necesarios para el diseño completo de stDatosEstadisticos.                                |
| desarrollo.cpp        | En este archivo se deben incluir los subprogramas por implementar. Si se necesitara implementar alguna otra acción o función además de las exigidas en el apartado anterior, también deben ser incluidas aquí y tener presente que sólo se deben modificar los archivos desarrollo.h y desarrollo.cpp.                     |
| Test.txt:             | Fichero opcional de configuración de aterrizaje para que sea más fácil de probar. Está configurado para aterrizar con éxito con los valores por defecto. La presencia de este fichero inhabilita los controles. Si se desea utilizar los controles se puede cambiar el nombre de este fichero y quedará deshabilitado.     |
| Makefile.win          | Archivo Makefile personalizado para este proyecto.                                                                                                                                                                                                                                                                         |

Describimos las funciones por desarrollar en el programa desarrollo.cpp:

- **Función para calcular la velocidad:** Esta función obtiene el valor de la velocidad del cohete en función del tiempo transcurrido: double fVelocidad (double v0, double D, double& Dt, double g, double t, BOOL swEncendido) ;



Siendo:

- $v_0$ : Velocidad inicial en el momento en que se llama a la función con  $t=0$ ;
- $D$ : Masa de combustible (kg) que se quema por unidad de tiempo (s).
- $Dt$ : Masa total de combustible quemado. Pasado por referencia.
- $g$ : Valor de la aceleración de la gravedad.
- $t$ : Tiempo total transcurrido mientras se mantienen constantes ( $D$ ) y ( $g$ ).
- $swEncendido$ : Indicador de motor encendido/apagado.

Debe tenerse en cuenta que en el momento en que se detecta una modificación en una de las variables ( $D$ ) o ( $g$ ), el valor del parámetro ( $t$ ) que se pasa a dicha función debe inicializarse a 0, con lo que, en este caso, el valor de  $v_0$  deberá ser el valor de la velocidad justo antes de la modificación.

- **Acción para controlar el cambio de parámetro  $D$  (variación de empuje):**

Esta acción debe controlar los parámetros con los que se llama a la función `fVelocidad`. Cuando se detecta un cambio en el valor de  $D$  (por modificación de la fuerza de empuje o bien por apagado/encendido del motor), debe llamarse a la función `fVelocidad` con los nuevos valores iniciales ( $t$ ,  $v_0$ ,  $D$ ). La función debe actualizar el parámetro `newD` y el valor de  $D$  en caso de que se acabe el combustible.

```
void fCtrVelocidad (int& newD, double& TtD, double& v0D,  
double& v, double& D, double& Dt, BOOL swEncendido);
```

Siendo:

- `newD`: Indica si ha habido un cambio en el valor de  $D$  (por modificación de empuje o bien por apagado/encendido del motor).
- Si no se detecta ningún cambio el valor es 0.
- Si se detecta algún cambio el valor es 1. Una vez detectado el cambio y llamado a la función `fVelocidad`, debe inicializarse el valor.
- `TtD`: Tiempo total transcurrido desde el último cambio de  $D$ . Al inicio de la simulación vale 0.



- vOD: Velocidad inicial en el momento de realizar el último cambio de D.
  - v: Velocidad actual del cohete. Al inicio de la simulación vale 0.
  - swEncendido: Indicador de encendido/apagado del motor.
- **Función para calcular el % de combustible disponible:** Esta función debe retornar el % de combustible disponible en el depósito:
- `double fPctDeposito (double Dt);` Siendo Dt el combustible quemado hasta el momento t. La masa inicial de combustible se fija por programa como constante. Tener presente que cuando no queda combustible para quemar, la variable D (combustible quemado por unidad de tiempo) debe ser 0.
- **Función para calcular la distancia recorrida por el cohete:** Esta función calcula el espacio total recorrido por el cohete desde el inicio de la simulación (parámetro distancia). Se toma como origen el punto inicial de la simulación. Un valor positivo significa que el cohete está más cerca de la superficie que en el instante inicial. El valor máximo que puede tomar es la distancia inicial del cohete a la superficie (significa que ha llegado a la superficie).

Para calcular la distancia utilizaremos la expresión en forma diferencia, pues supondremos que en un intervalo de tiempo suficientemente pequeño ( $\Delta t$ ) la velocidad se mantiene constante.

$$\Delta e = v * \Delta t$$

Esta expresión nos indica el incremento (o disminución) del espacio recorrido en un intervalo de tiempo  $\Delta t$ .

Con la explicación anterior y teniendo presente la solución adoptada para la implementación de la función fVelocidad, definir la función para obtener la altura a la que se encuentra el cohete en el momento t:

```
double fAltura (double& distancia, double velocidad);
```

El significado de los parámetros se ha explicado anteriormente. La altura inicial es DIST\_INI.

- **Función para controlar el final de la simulación:** Esta función retorna la situación en que se encuentra la simulación:
- ```
int fSituacion (double altura, double v);
```



Siendo:

- altura: La distancia cohete-superficie.
- v: La velocidad.
- 
- Valores de retorno:
- 0: No se ha llegado a la superficie.
- 1: Se ha llegado a la superficie y el aterrizaje ha sido correcto.
- 2: Se ha llegado a la superficie y el cohete se ha estrellado.
- 3: El cohete se ha perdido.

- **Acción para inicializar variables:** Esta acción inicializa las variables al comenzar la simulación. Debe tenerse en cuenta que al iniciar la simulación el motor del cohete está apagado.

```
void fInicializa (int& newD, double& altura, double& e0,  
double& velocidad, double& v0, double& Tt, double& D,  
double& Dt, double& Ftotal, double& Fempuje);
```

Siendo:

- newD: Indica si se ha introducido un nuevo valor del empuje (valores 0,1).
- altura: La distancia cohete-superficie hasta el momento Tt.
- e0: Espacio recorrido al inicio de la simulación.
- velocidad: velocidad del cohete hasta el momento Tt.
- v0: Velocidad inicial al inicio de la simulación.
- Tt: Tiempo total transcurrido desde el inicio de la simulación.
- TtD: Tiempo total transcurrido desde el último cambio de D. Al inicio de la simulación vale 0.
- D: Pérdida de masa de combustible por unidad de tiempo.
- Dt: Masa de combustible quemada.
- Ftotal: Peso total del cohete (carga útil + combustible).
- Fempuje: Fuerza de empuje  $u \cdot D$ .
- g:Gravedad

- **Función para obtener la coordenada "y" del cohete relativa al inicio de la simulación:** Esta función realiza un cambio de escala para poder representar la posición del cohete en pantalla. Retornará el número



de pixels del eje “y” que hay que desplazar el cohete en relación a la posición inicial. Un valor positivo significa que el cohete está más cerca de la superficie.

Debe tenerse en cuenta que la distancia inicial cohete-superficie se representa mediante una distancia gráfica de 480 pixels (constante DIST\_PIX).

```
int fyCohete (double altura);
```

Siendo:

altura: La distancia cohete-superficie hasta el momento Tt.

- **Función para obtener la coordenada “y” que marca el nivel del depósito:** Esta función realiza un cambio de escala para poder representar la posición del nivel del depósito de combustible. Retornará el numero de pixels del eje “y” que hay que desplazar el nivel en relación a la posición inicial. Debe tenerse en cuenta que el depósito lleno se representa mediante una distancia gráfica de 360 pixels (constante TANQUE\_ALTO)

```
int fyDeposito (double pct);
```

Siendo:

pct: porcentaje de combustible del depósito.

- **Función para obtener la fuerza total sobre el cohete (peso-empuje):** La fuerza total sobre el cohete será la diferencia entre el peso (negativo) y el empuje (positivo). El peso es la masa del cohete en el instante t por la gravedad.

```
double fFtotal (double Dt, double D, double g, BOOL  
swEncendido);
```

Siendo:

- Dt: Masa de combustible quemada.
- D: Pérdida de masa de combustible por unidad de tiempo.
- swEncendido: Indicador de encendido del motor.
- Constantes:
- u: Velocidad escape de los gases.
- MASAT\_INI: Masa total inicial del cohete (carga + combustible).



- **Acción para guardar datos estadísticos:** Con el objetivo de calcular los datos estadísticos necesarios para comparar los resultados obtenidos en los diferentes aterrizajes, se deberá guardar la información necesaria para obtener dichos resultados. Estos datos deberán guardarse en una tupla cuya estructura se debe definir. Prever un máximo de 1000 registros. Esta acción se ejecuta al inicio de la simulación, al final de la misma y cada vez que se pulsa uno de los 5 botones identificados.

La acción es la siguiente:

```
void fRegistraDatos (int idBoton, double Tt, double Mc,
BOOL swInicio, BOOL swFin, double g, stDatosEstadisticos&
DatEst);
```

Siendo:

- idBoton: Identificador del botón pulsado.
    - 4 - Encender/Para motor.
    - 5 - Empuje + 100.
    - 6 - Empuje + 1000.
    - 7 - Empuje - 100.
    - 8 - Empuje - 1000.
  - Tt: Instante de tiempo en el que se pulsa el botón (en segundos).
  - Mc: Masa de combustible en el momento que se pulsa el botón.
  - swInicio: Indica el inicio de simulación.
  - swFin: Indica el fin de simulación.
  - g: Valor de la gravedad.
  - stDatosEstadisticos: Estructura definida para almacenar los datos estadísticos.
- **Función para obtener datos estadísticos:** Con esta función se deben obtener las siguientes magnitudes estadísticas: número de veces que se ha pulsado cada botón, tiempo medio transcurrido entre pulsación de cada botón, número total de pulsaciones, consumo medio, consumo total y tiempo total transcurrido. Se llamará a esta función para presentar los resultados en pantalla al finalizar la simulación. La función es la siguiente: `double fObtieneDatos (int idBoton, int tEstadistica, stDatosEstadisticos DatEst);`

Siendo:

- idBoton: Identificador del botón pulsado.
- tEstadistica: Tipo de estadística. Valores posibles:



- 1 - Número de pulsaciones del botón identificado por idBoton.
- 2 - Tiempo medio entre pulsaciones del botón identificado por idBoton.
- 3 - Número total de pulsaciones.
- 4 - Consumo medio.
- 5 - Consumo total.
- 6 - Tiempo total.
- DatEst: Variable donde se guardan los datos registrados.
- Retorna el valor solicitado en función de los parámetros de entrada.

#### 4.3. Caso de estudio 2: Definición del proyecto de control de un ascensor

El objetivo de este proyecto es establecer la lógica de control de prioridades de un sistema de 2 ascensores (A y B) en un edificio de 8 pisos y planta baja (0 a 8). Los ascensores pueden ser llamados desde el exterior o desde el interior. La llamada exterior se efectúa mediante un botón situado en cada planta, en la zona intermedia delimitada entre las puertas exteriores de los 2 ascensores. Cada ascensor dispone de un conjunto de 8 botones que representan cada uno de los pisos. Presionar un botón exterior significa dar una "orden de llamada" a cualquiera de los 2 ascensores para que se dirija a la planta desde la que se ha efectuado la llamada y se pare en ella. Presionar un botón interior de un ascensor significa dar una "orden de destino" a dicho ascensor para que se dirija a la planta correspondiente al botón presionado y se pare en ella. El sistema permite memorizar las órdenes de llamada y las ordenes de destino pendientes de cada planta y ascensor, las cuales permanecen vigentes mientras la orden no se haya ejecutado. Consideramos que una orden de llamada se ha ejecutado si cualquiera de los 2 ascensores se ha parado en la planta. Consideramos que una orden de destino se ha ejecutado si el ascensor desde el que se ha efectuado dicha orden se ha parado en el destino seleccionado. En el estado inicial del sistema los 2 ascensores se encuentran en la planta baja.

A continuación detallamos las reglas que deben tenerse en cuenta para el control del movimiento:

- El sistema debe atender todas las órdenes vigentes.
- El sistema quedará en el estado determinado por la última orden vigente ejecutada.
- Una orden de destino interior de cualquiera de los 2 ascensores tiene prioridad respecto a una orden de llamada exterior desde cualquier planta.

- Una orden de llamada desde un botón exterior será atendida por el ascensor que esté más cerca de dicha planta.
- Una orden de destino desde un botón interior será atendida por el ascensor desde el que se ha efectuado dicha orden.
- Las órdenes de destino se priorizan según dos parámetros: sentido del viaje (ascendente o descendente) y distancia hasta el destino. El sentido tiene prioridad sobre la distancia hasta el destino. Una distancia menor tiene prioridad sobre una distancia mayor.
- El ascensor A tiene prioridad sobre el B. Esto significa que, en una situación en la que los 2 ascensores se encuentren en las mismas condiciones, la orden de llamada será atendida por el ascensor A.
- Una orden de llamada no será atendida por un ascensor que está atendiendo una orden de destino. No obstante, se considera que la orden se ha ejecutado si cualquiera de los 2 ascensores se para en dicha planta, independientemente del sentido del viaje del ascensor que se para.

#### 4.3.1. Implementación

A continuación describimos la acción que se debe implementar, el funcionamiento del programa principal y las estructuras de datos sobre las que se deben apoyar.

Esta acción se incluye en un programa principal que utiliza componentes gráficos de Windows para una mejor visualización del comportamiento del sistema. El proyecto `Ascensor.dev` incluye todos los programas necesarios para que al completar el programa `desarrollo.cpp` los programas simulen el comportamiento de un ascensor.

Los archivos utilizados son los siguientes:

| ARCHIVO        | DESCRIPCIÓN                                                                                                                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ascensor.dev   | Archivo de proyecto en el que se definen el resto de archivos. Necesario para compilar y ejecutar el programa.                                                                                                               |
| main.cpp       | Programa principal que contiene las rutinas de control de los componentes gráficos y tratamiento de eventos (teclado, ratón, temporizador, etc.).                                                                            |
| datos_main.h   | Archivo de cabecera que contiene las constantes, definiciones de tipo y estructuras utilizadas en los programas.                                                                                                             |
| desarrollo.cpp | Archivo que contendrá la acción por desarrollar. La versión que se entrega contiene únicamente la definición de la acción ( <code>mCtrlAscensor</code> ) y un retorno fijo de variables para que sea posible su compilación. |
| Makefile.win   | Archivo Makefile personalizado para este proyecto.                                                                                                                                                                           |



#### 4.3.2. Funcionamiento del programa principal

El programa principal está programado en entorno Windows. En la ventana de ejecución aparecen representados esquemáticamente los 2 ascensores (A ascensor izquierdo, B ascensor derecho) en 2 columnas verticales. Cada ascensor ocupa una región rectangular de cada columna, que se va desplazando a medida que va atendiendo órdenes de llamada o destino. En la parte intermedia comprendida entre las columnas (plantas) aparecen los botones de planta y al lado de cada bloque se representan los botones de cada ascensor.

Los botones se pueden “pulsar” mediante el teclado o mediante el botón izquierdo del ratón. En el caso del teclado, se utilizan las teclas del 0 al 8, (solas en el caso de los botones de llamada de planta o en combinación con las teclas Ctrl o Alt para los ascensores A o B respectivamente). Las 2 primeras reglas de movimiento ya están implementadas en el programa principal.

#### 4.3.3. Implementación de la rutina de control de movimiento

Debemos construir una acción (mCtrlAscensor) que implemente las reglas de control para el movimiento de los ascensores.

Dicha acción tiene las siguientes características:

##### Parámetros de entrada:

- Información relativa al estado de cada conjunto de 9 botones (botones de llamada desde cada planta y botones de destino de cada ascensor). Se almacena independientemente en un vector de 9 elementos. Cada elemento es una tupla definida de la siguiente manera:

```
typedef struct stEstadoBoton {
double XBoton;      //Posición X del botón (se utiliza para
dibujar el botón)
double YBoton;      //Posición Y del botón (se utiliza para
dibujar el botón)
BOOL   VBoton;      //Llamada vigente (FALSE no vigente, TRUE
vigente)
} ;
```

**Variables utilizadas:** stEstadoBoton BotonP[9], stEstadoBoton BotonA[9], stEstadoBoton BotonB[9].

- Información relativa a la planta actual y sentido de movimiento de cada ascensor. El sentido se almacena en una variable tipo int con la siguiente codificación: 0 → ascensor parado, 1 → ascensor en movimiento, sentido ascendente, 2 → ascensor en movimiento,



sentido descendente. Variables utilizadas: int PlantaAct\_A, int Sentido\_A, int PlantaAct\_B, int Sentido\_B.

#### Parámetros de salida:

- Información de la posición de destino (planta) de cada ascensor. Esta información se almacena en un vector de 2 elementos tipo int con el siguiente significado: elemento [0] → Planta de destino del ascensor A, [1] → Planta de destino del ascensor B. Variable utilizada: int Destino[2].
- NOTA: Al estar definidos como un vector, los valores se pasan por referencia, con lo que pueden actualizarse desde dentro de la acción para ser recogidos desde el módulo que llama a la acción.

#### Definición de la acción:

```
void mCtrlAscensor (stEstadoBoton BotonP[9], stEstadoBoton
BotonA[9], stEstadoBoton BotonB[9], int PlantaAct_A, int
Sentido_A, int PlantaAct_B, int Sentido_B, int Destino[2]);
```

### 4.4. Caso de estudio 3: El juego del frontón

El objetivo es controlar el movimiento de una bola (pelota) sobre una superficie rectangular que simula el terreno de juego. Paralela a la pared lateral derecha hay una figura rectangular más pequeña que simula una pala móvil.

#### Reglas del juego

Si la bola toca el lado derecho del cuadrado se termina el juego. La pala rectangular permite al usuario evitar que la bola toque ese lado.

Si la bola toca cualquiera de los otros lados rebota hacia el interior del cuadrado.

#### Control del juego

- Con las teclas d (difícil) y f (fácil) se regula la velocidad de la pelota.
- Con las teclas de flecha hacia arriba o hacia abajo se controla el movimiento de la pala a lo largo del lado derecho del cuadrado, dentro de la superficie.
- Con el botón izquierdo del ratón se coloca la pelota en la posición inicial (dentro del terreno) y la pala cerca del lado derecho. Con el botón derecho del ratón se inicializa el movimiento.
- Con la barra espaciadora se para o reanuda el movimiento.

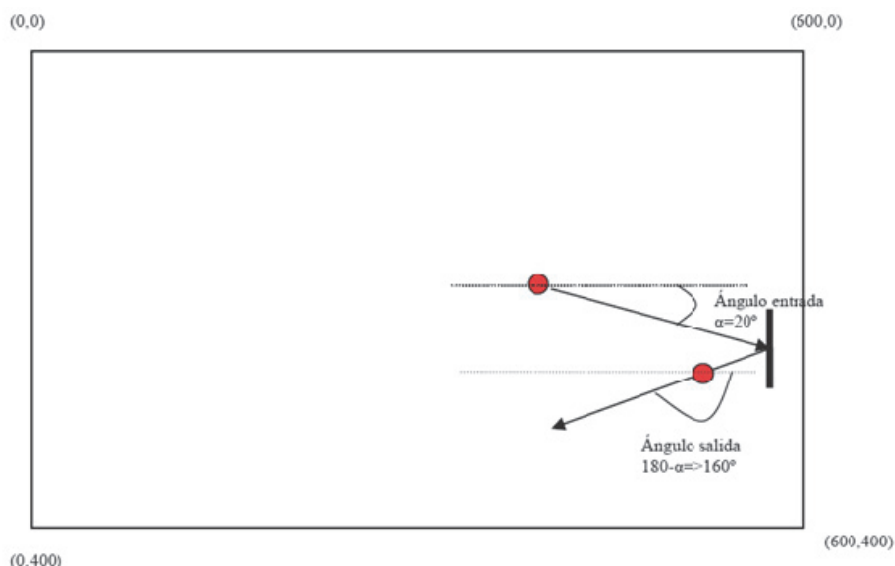


### Tareas que realizar en la programación:

- Definir una acción void **mMoverPelota** (int tipo, double& x, double& y, double v, double& Alpha) que calcule los nuevos valores **x**, **y**, **Alpha** siendo:
  - tipo: Determina el tipo de movimiento.
    - 0 nuevo ángulo. Toca el borde.
    - 1 nuevas coordenadas x, y. Mismo ángulo.
    - 2 nuevo ángulo. La pelota toca la pala.
  - x, y: Coordenadas de posición de la pelota (inicialmente se pasan los valores correspondientes al instante t-1 y se deben calcular los valores en el instante t).
  - v: Velocidad
  - Alpha: Ángulo en grados (inicio en el cuadrante inferior derecho. Inicialmente se pasan los valores correspondientes al instante t-1 y se deben calcular los valores en el instante t).
- La acción mMoverPelota debe calcular lo siguiente:
  - Primero calculamos el nuevo ángulo de movimiento en caso de que el tipo sea cero o dos (según la Figura 4.4.1.).
  - Si el tipo de movimiento es 0, variamos el ángulo de la trayectoria. Si toca los bordes laterales "Angulo salida" = 180 - "Angulo entrada".
  - Si toca los bordes superior o inferior "Angulo salida" = 360 - "Angulo entrada".
  - Convertimos un ángulo negativo en positivo sumando 360 grados.
  - Una vez obtenido el ángulo, calculamos la posición del siguiente instante (temporizador) en todos los casos, la ecuación de movimiento:  $x_t = x_{t-1} + v * \cos(\text{Alpha})$ ,  $y_t = y_{t-1} + v * \sin(\text{Alpha})$ . Se deben convertir los grados en radianes para poder aplicar la ecuación anterior.

Los ficheros necesarios para implementar las funciones y mostrar la parte gráfica son los siguientes:

Figura 4.4.1  
Explicación gráfica del  
cálculo de nuevo ángulo  
cuando la pelota toca la  
pala (tipo movimiento 2).  
En este caso  
 $\text{Alpha} = 180 - \text{Alpha}$ .  
Inicialmente el ángulo se  
fija en  $20^\circ$ ,  $\text{ALTO} = 400$ ,  
 $\text{ANCHO} = 600$ .



| ARCHIVO                        | DESCRIPCIÓN                                                                                                   |
|--------------------------------|---------------------------------------------------------------------------------------------------------------|
| <a href="#">fronton.dev</a>    | Archivo de proyecto en el que se definen el resto de archivos. Necesario para compilar y ejecutar el programa |
| <a href="#">desarrollo.cpp</a> | En este archivo deben implementarse las funciones y acciones descritas.                                       |
| <a href="#">desarrollo.h</a>   | Contiene la declaración de constantes.                                                                        |
| <a href="#">funciones.h</a>    | Contiene las cabeceras (prototipos) de las funciones y acciones por desarrollar.                              |
| <a href="#">Makefile.win</a>   | Contiene las opciones de compilación para este proyecto.                                                      |

## 4.5. Caso de estudio 4: Movimiento de un cuerpo en caída libre

Se debe simular, en un entorno gráfico, el movimiento de un cuerpo en caída libre, teniendo en cuenta que la resistencia o fuerza de rozamiento del aire es proporcional al cuadrado de la velocidad:

$$F_R = -kv^2$$

Se dispone de la infraestructura gráfica necesaria para simular el movimiento, siendo necesario únicamente implementar la función para calcular la posición y velocidad del cuerpo en movimiento en un instante dado suponiendo que hay rozamiento (fichero desarrollo.cpp). La función que calcula los valores sin rozamiento ya está codificada.

Al ejecutar el programa, aparece una ventana Windows. En la pantalla se presentarán 2 trayectorias: la trayectoria seguida por una esfera en caída libre (en color rojo) y la trayectoria seguida por la esfera con rozamiento (en color verde), esta última implementada a partir de la función “conrozamiento” descrita más adelante.

- Funcionamiento de los eventos del programa:
- Con el botón izquierdo del ratón situamos el punto inicial de caída e iniciamos el movimiento.
- Con la pulsación de la tecla Ctrl paramos el movimiento.
- Con la liberación de la tecla Ctrl reanudamos el movimiento.
- Para cada esfera, se muestra un visualizador con la velocidad alcanzada en cada momento.

Los programas necesarios para la implementación son los siguientes:

| ARCHIVO                            | DESCRIPCIÓN                                                                                                    |
|------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <a href="#">CaídaLibreW_V1.dev</a> | Archivo de proyecto en el que se definen el resto de archivos. Necesario para compilar y ejecutar el programa. |
| <a href="#">CaídaLibreW_V1.cpp</a> | Este fichero contiene la parte gráfica y el bucle principal de tratamiento de mensajes.                        |
| <a href="#">CaídaLibreW_V1.h</a>   | Este fichero contiene las constantes y definiciones de prototipos.                                             |
| <a href="#">desarrollo.cpp</a>     | Este fichero es el que contiene las funciones por implementar.                                                 |
| <a href="#">Makefile.win</a>       | Este fichero contiene las opciones de compilación para este proyecto.                                          |

#### 4.5.1. Base teórica

Suponiendo que la fuerza de rozamiento es proporcional al cuadrado de la velocidad y sabiendo que dicha fuerza se opone a la fuerza de la gravedad ( $m*g$ ), la fuerza total ejercida sobre el cuerpo en caída libre es  $F = m*g - k*v^2$ . Aplicando la segunda ley de Newton, la ecuación del movimiento de un cuerpo en caída libre viene determinada por la siguiente expresión:

$$m \frac{d^2 y}{dt^2} = mg - kv^2 \quad (1)$$

donde suponemos que el cuerpo cae según el eje “y”, “m” es la masa del cuerpo, “k” es el factor de rozamiento y “g” es el valor de la gravedad. Para una esfera de radio 0,3 metros, el valor de “k” es, aproximadamente, de 0,009 N·s<sup>2</sup>/m<sup>2</sup>



Definimos la velocidad límite como la velocidad máxima alcanzada por el cuerpo en caída libre. Esta velocidad se alcanza cuando la fuerza de rozamiento iguala a la fuerza de gravedad, es decir, cuando la fuerza total ejercida sobre el cuerpo es cero o, dicho de otra manera, cuando la aceleración del cuerpo es 0. Se puede demostrar que la velocidad límite es:

$v_L = \sqrt{\frac{mg}{k}}$ , por lo que para una esfera de masa unidad y radio de 0,3 metros, la velocidad límite es aproximadamente de 33,01514 m/s. Llamaremos “distancia límite” a la distancia recorrida por el cuerpo cuando se alcanza la velocidad límite.

#### 4.5.2. Función que implementar

Se deberá implementar una función que, dependiendo del valor de un parámetro de entrada, retorne la velocidad o bien la distancia alcanzada en el instante t. El prototipo de la función es el siguiente:

```
double conrozamiento (int vRetorno, double k, double tf,
int & lim),
```

donde vRetorno puede tener el valor 1 ó 2 según se quiera retornar la distancia o la velocidad, k es el factor de rozamiento, tf es el tiempo transcurrido desde el instante inicial y lim es una variable que indica si la distancia calculada por la función corresponde a la distancia límite o no (lim=0 no; lim=1 sí). Suponemos que en el instante inicial (t=0), la distancia y la velocidad iniciales son cero.

Para resolver la ecuación de movimiento (1), descomponemos el movimiento en intervalos pequeños de tiempo ( $\Delta t=0,05$  s) y consideramos constante la aceleración en cada intervalo. En estas condiciones la distancia recorrida al final de cada intervalo es:  $d=d_0 + v_0*\Delta t + a*(\Delta t^2)/2$  y la velocidad al final de cada intervalo es:  $v=v_0 + a*\Delta t$ ,

donde  $v_0$  y  $d_0$  son la velocidad inicial y distancia inicial al inicio de cada intervalo. En la práctica significa construir un bucle mientras el tiempo total acumulado de intervalos sea menor o igual que  $tf$ .

Al final de cada intervalo (iteración) recalculamos la aceleración del intervalo siguiente según la expresión (1):  $a = g - (k*v^2/m)$ .

A efectos prácticos consideramos que se ha alcanzado la distancia límite cuando la aceleración es inferior a un determinado valor Epsilon (Epsilon=0,001).



### 4.5.3. Tareas para incorporar la función implementada

Se deberán copiar en el disco, en el mismo directorio, los siguientes programas:

CaidaLibreW\_V1.dev, CaidaLibreW\_V1.cpp, CaidaLibreW\_V1.h y desarrollo.cpp.

Abrir el fichero CaidaLibreW\_V1.dev y posicionarse en el fichero desarrollo.cpp e implementar la función:

```
double conrozamiento (int vRetorno, double k, double tf,
int & lim).
```

## 4.6. Problemas propuestos

### 4.6.1. Enunciado 1

Definir un programa que, dado un polígono convexo, verifique si un punto P entrado por el usuario está dentro o fuera de la superficie delimitada por los lados del polígono. Se utilizará el botón izquierdo del ratón para dibujar el polígono. Cada clic determinará un vértice y el doble clic indicará el último vértice. Al detectar este doble clic, el programa dibujará el contorno del polígono de V vértices. A partir de este momento, mediante un clic con el botón derecho del ratón, se aceptará como entrada el punto P por verificar y se pintará dicho punto de color verde o rojo en función de si está dentro o fuera del polígono.

Se propone, como “algoritmo” para solucionar el problema, la descomposición del polígono en “n” triángulos ( $n=V-2$ ) con un vértice común  $X_0$ . En la figura 4.6.1 de ejemplo se observa la descomposición del polígono convexo de 5 vértices en 3 triángulos. Dado un punto P se comprueba si dicho punto está o no en el interior de cualquiera de los triángulos en que se descompone el polígono.

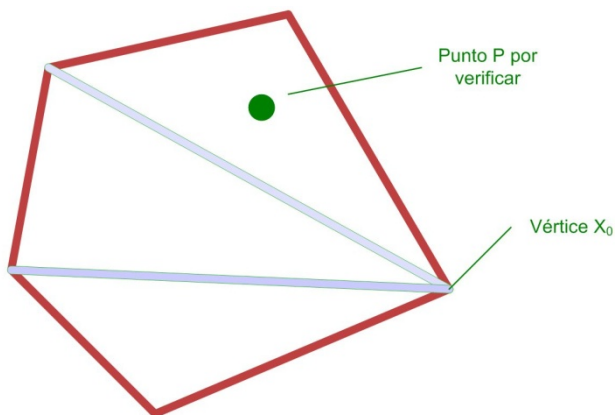


Fig. 4.6.1  
Ejemplo de  
descomposición de  
un polígono.



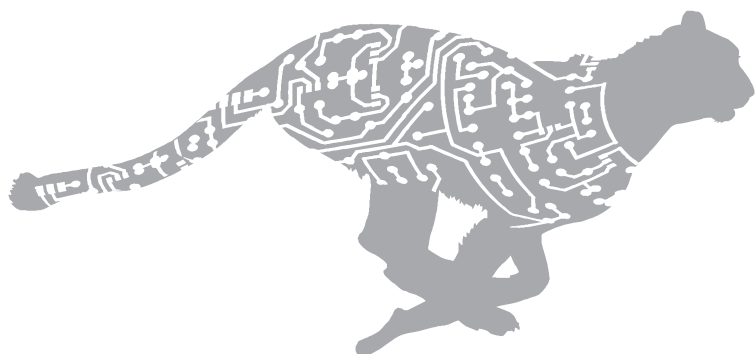
#### 4.6.2. Enunciado 2

Definir un programa que simule las funciones de una máquina expendedora de productos. El programa debe permitir realizar las siguientes funciones:

- **Selección de un producto:** Se presentará en pantalla la relación de los 5 productos disponibles. La selección se puede simular mediante un clic en un botón asociado a cada producto (de manera similar a la selección del piso en el proyecto del ascensor). Cada producto tendrá un precio asignado (entre 1,00 y 10,00 euros). Se admiten fracciones de 10 céntimos. Si no hay disponibilidad de un producto no se permitirá su pago.
- **Pago del producto seleccionado:** Mediante un botón de 10 céntimos y otro de 1 euro, se simulará el pago del producto. El importe máximo que se puede introducir es de 10 euros. Mediante otro botón se marcará como “finalizada” la introducción de monedas. En ese instante debe verificarse si el importe introducido es igual o superior al producto seleccionado y, además, en caso de tener que devolver cambio, si existen suficientes monedas en la máquina para devolverlo. En caso de no haber suficiente cambio, se “devuelve” el dinero introducido.
- **Recarga de existencias:** Mediante un botón se simulará la reposición de existencias del producto seleccionado. Puede haber un máximo de 10 unidades por producto.
- **Recarga de cambio:** Mediante 2 botones se podrán reponer las monedas de 10 céntimos y de euro para el cambio. Se puede recargar un máximo de 40 monedas de 10 céntimos y 20 monedas de euro.
- **Recogida de monedas:** Mediante un botón se podrán recoger las monedas existentes en la máquina.

Deben aparecer visualizadores que indiquen los siguientes datos:

- Producto seleccionado.
- Existencias de producto por cada ítem.
- Monedas disponibles “en máquina”. Se actualiza cuando se recarga el cambio, se recogen las monedas o se sirve un producto previamente seleccionado.
- Monedas introducidas una vez seleccionado el producto. Es el importe introducido para pagar el producto seleccionado. Se actualiza cuando se paga y queda a cero cuando se sirve el producto o cuando se devuelve el dinero introducido.



→ 5



# Las excepciones y su manejo

## 5.1. Introducción

Las excepciones son un marco de trabajo para implementar la tolerancia a fallos. En este capítulo se consideran las excepciones y su manejo, y cómo desarrollarlas en lenguajes de programación concretos para tiempo real. En la actualidad cualquier lenguaje de programación de tiempo real tiene las herramientas necesarias para facilitar la construcción de sistemas altamente fiables. Por ejemplo: herramientas de gestión de excepciones incorporadas directamente al lenguaje de programación.

El mecanismo de manejo de excepciones (*Exception Handling*) es útil para la implementación de sistemas tolerantes a fallos (*Fault Tolerant*) en el dominio del diseño de **software**, utilizando estrategias de "*forward error recovery*". El método presentado es ampliamente utilizado y accesible desde diferentes lenguajes de programación (Ada, Java, C++, etc.). No obstante, centraremos el enfoque en el lenguaje C/C++.

Existen varias razones para utilizar este método, pero quizás la más interesante sea que permite **separar** la gestión de errores de la codificación "normal", lo cual ayuda en gran medida a la legibilidad de los programas. Este método también permite **centralizar** la gestión de errores, lo que ayuda a simplificar el mantenimiento del código debido al mecanismo de **propagación** del error.

Los métodos alternativos para gestionar los errores de software se basan habitualmente en la evaluación de los valores de retorno de las funciones. Esto implica que el retorno de una función no puede utilizarse para otros fines. Además, el código resultante contiene bloques "*if*" adicionales para dicha evaluación y tratamiento.



El concepto global del manejo de excepciones es sencillo:

- Se activa mediante software (programa) un “flag” de error cada vez que se produce una condición o situación errónea.
- Hay un mecanismo que continuamente está a la espera de que se haya activado el “flag” de error.
- Este mecanismo hace una llamada a la rutina de tratamiento de error correspondiente. Si la rutina que ha activado el flag de error no tiene definido un bloque de tratamiento de excepciones, la excepción se “propaga” a los niveles superiores hasta llegar al nivel principal (main).

### 5.1.1. Flujo de programa

La activación del “flag” de error se efectúa simplemente activando (*raise*, *throw*) un error. Cuando el error es activado, el sistema en su conjunto responde al mismo capturando el error. Cuando esto ocurre, se ejecuta el bloque de código de tratamiento de error más cercano al lugar en que se ha producido dicho error.

Una de las características principales del manejo de excepciones es que el error se puede **propagar** fuera de los límites de la función en que se ha producido. Es decir, si se detecta un error en una función de nivel inferior, y no se ha especificado rutina de tratamiento de error en dicha función, el error puede propagarse a funciones de nivel superior hasta llegar a una función en que se haya definido (por ejemplo la función principal (main)). Esto permite centralizar en un único lugar el bloque de tratamiento de error.

A pesar de lo expuesto anteriormente, en el caso concreto de las rutinas POSIX Pthreads-win32 utilizadas en los ejemplos escritos en este libro, no utilizamos el manejo de excepciones, dado que todas las funciones incluidas en la librería devuelven un entero como código de retorno, siendo el valor 0 el que indica que el resultado ha sido correcto. Este mecanismo puede aplicarse sin demasiados problemas cuando el nivel de anidamiento de funciones es bajo. Un ejemplo de implementación podría ser:

```
// rutina general tratamiento errores
// muestra mensaje en un fichero de 'log' y acaba el
programa
#include "check.h"
..
rc = pthread_setschedparam(thread, policy, &param);
checkResults("pthread_setschedparam()\n", rc);
..

rc = pthread_getschedparam(thread, &policy, &param);
checkResults("pthread_getschedparam()\n", rc);
```

## 5.2. Manejo de excepciones en C++

En C++, el bloque de código en el que se quiere definir una rutina de tratamiento de errores empieza por la sentencia **try {...}**, seguida por una sentencia **catch {...}**. Dentro del bloque **try** se pueden activar errores mediante la sentencia **throw**. Dentro del bloque **catch** se incluye la lógica para el tratamiento de cada error. El siguiente trozo de pseudocódigo muestra la idea:

```
try {    ...
    ...
    throw Exception() // se activa el error
    ...
} catch( Exception e ) // se trata el error
{
    ...
    ...
}
```

El bloque **try** encierra las instrucciones que podrían ocasionar excepciones. Debe haber por lo menos un bloque **catch** (también como manejador **catch**) inmediatamente después del bloque **try**. Cada manejador **catch** especifica entre paréntesis un manejador de excepción que representa el tipo de excepción.

Aunque el sistema de manejo de excepciones no está implementado en C, puede implementarse un sistema de características parecidas mediante la ayuda de las sentencias *setjump()/longjump()*. Estas sentencias pueden ser peligrosas ya que provocan que se limpie la pila sin llamar a los destructores y esto puede provocar serios problemas.

## 5.3. Ejemplos resueltos

Ejemplo 1. Diseñamos una clase que sea capaz de almacenar información sobre el error producido y de mostrar además un mensaje con el error. El siguiente código únicamente muestra un mensaje:

```
class CException
{
public:
    char* message;
    CException( char* m ) { message = m };
    Report();
}
```

Una vez construida esta clase, podemos ver un ejemplo en pseudocódigo de cómo utilizarla. Suponemos que las funciones *Initialize()* y *Run()* son



funciones complejas que además llaman a subfunciones anidadas de varios niveles.

```
try {
    Initialize();
    Run();
    Shutdown();
} catch( CException e )
{
    e.Report();
}
```

Aunque la rutina de tratamiento de errores desarrollada en este ejemplo es muy simple, podemos observar que con el mecanismo de manejo de excepciones evitamos tener que controlar el retorno de cada una de las funciones y subfunciones, centralizando en un único punto del programa el tratamiento de los errores.

Ejemplo 2. Construir un programa C++ que implemente el manejo de excepciones a varios niveles. Para ello definiremos 4 acciones que sencillamente lanzarán una excepción en caso de que el valor del parámetro de entrada sea 1, 2, 3 ó 4 respectivamente. En caso de que el valor del parámetro sea 5 se lanzará excepción en todas ellas.

```
void f(int lugar_error);
void g(int lugar_error);
void h(int lugar_error);
void i(int lugar_error);
```

La rutina de tratamiento de error será de una clase similar a la del ejemplo 1 del apartado 5.3, donde la función Report() puede sustituirse simplemente por un cout:

```
class Err {
public:
    const char* error;
    const char* lugar;
    Err(const char* arg1, const char* arg2) :
error(arg1), lugar(arg2) {
        cout<<"Error en rutina: " <<error<<" lugar:
"<<lugar<<endl;
    }
};
```

Utilizando esta clase, las excepciones se activan mediante la sentencia **throw Err("funcion xxx","lugar yyy")** y se capturan mediante la sentencia **catch (Err& e) { }**



La acción `f(int lugar_error)` tiene tratamiento local de excepciones, mientras que el resto de acciones no tienen tratamiento local, dejándose éste en la rutina `main`. Dentro de `main` se pide entrar el valor para la variable `"lugar_error"`. Todas las excepciones que se lanzan son de tipo **Err** (el tipo de la clase que trata la excepción) excepto la acción `h(int lugar_error)`, que lanza una excepción de tipo **int** para mostrar el funcionamiento de **catch (...)**.

Todas las acciones se llaman desde `main` excepto la acción `i(int lugar_error)` que se llama desde `f`.

El programa debe mostrar outputs en pantalla que indiquen el lugar por donde pasa. Observar que pasará por lugares diferentes en función del valor de `lugar_error`.

→6



# Programación concurrente

## 6.1. Introducción

Denominamos programación concurrente a la notación y técnicas de programación que expresan el paralelismo potencial y resuelven los problemas resultantes de la sincronización y la comunicación. La importancia de la programación concurrente está en que proporciona un entorno abstracto donde estudiar el paralelismo sin tener que enfrascarse en los detalles de implementación [3].

La concurrencia en los lenguajes de programación permite definir programas más legibles, fáciles de mantener, aumenta la portabilidad y, sin ella, en los casos de sistemas embebidos no podría existir un sistema operativo residente. Los tres servicios básicos que debe proporcionar la programación concurrente son: definición de procesos, sincronización de procesos y comunicación entre 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. Cada proceso se ejecuta en su propia máquina virtual para evitar interferencias con otros procesos no relacionados. Los sistemas operativos modernos permiten crear procesos dentro del mismo programa accediendo, de modo compartido y sin restricciones, a la memoria común (estos procesos suelen llamarse hilos o hebras).



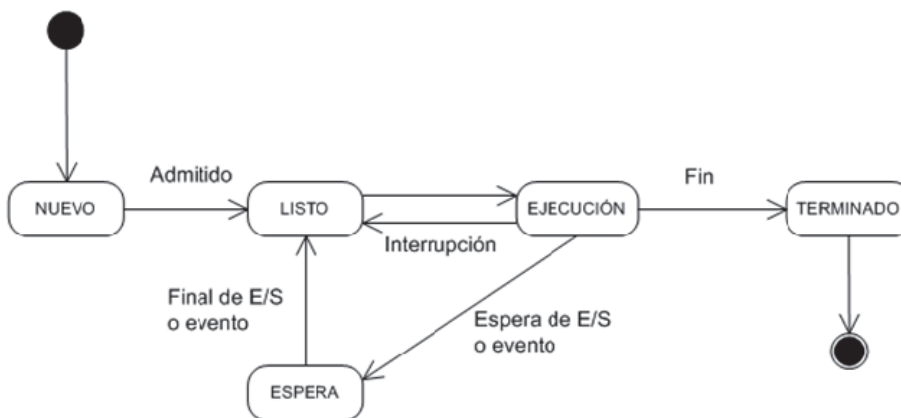
Estos hilos podemos encontrarlos a nivel de usuario y a nivel del kernel. Los hilos del kernel tienen soporte directo del sistema operativo, mientras que el resto son implementados por una biblioteca de hilos que da soporte a la creación, programación y administración de hilos sin soporte del kernel. Entre las bibliotecas de hilos de usuario están las librerías POSIX, que serán el objeto de estudio en este capítulo. Los sistemas operativos que se ajustan a POSIX deben distinguir entre la concurrencia de programas (procesos) y la concurrencia dentro de un programa (hilos).

Los hilos a nivel de usuario se pueden crear más rápidamente que los hilos del kernel.

Los modelos para relacionar los hilos de usuario y del kernel son tres: el modelo de varios a uno mapea muchos hilos de usuario con un solo hilo del kernel; el modelo uno a uno mapea cada hilo de usuario con un hilo correspondiente del kernel; y el modelo de varios a varios combina muchos hilos de usuario con un número menor o igual de hilos del kernel [6].

A medida que un proceso se ejecuta va cambiando su estado. Un proceso está en estado nuevo cuando ha sido creado, está en estado de ejecución si se están ejecutando las instrucciones, y está en espera cuando el proceso está esperando que ocurra algún evento, como la terminación de una operación de E/S o la recepción de una señal. Si el proceso se ha creado pero está en espera de ser asignado a un procesador su estado es "listo". Cuando termina la ejecución el proceso está en estado "terminado". En la Figura 6.1.1 se muestra el diagrama de estados de un proceso.

Fig. 6.1.1.  
Diagrama de estado de  
un proceso.



Cada proceso está representado con su propio bloque de control de proceso en el sistema operativo. En una cola de espera se coloca un proceso que no está en ejecución. El sistema operativo tiene dos clases de

colas: las colas de solicitudes de E/S y la cola de procesos listos. Estos procesos listos esperan a ser ejecutados en la CPU. La planificación de trabajos de largo plazo es la selección de los procesos que podrán competir por la CPU. La planificación de la CPU a corto plazo es la selección de un proceso de la cola de procesos listos.

Los procesos pueden ejecutarse en el sistema de forma concurrente. Esta ejecución requiere de un mecanismo para la creación y eliminación de procesos. Los procesos que se ejecutan en la CPU pueden ser independientes o cooperativos. Los procesos cooperativos cuentan con dos esquemas de comunicación complementarios: el de memoria compartida y el sistema de mensajes.

Los hilos permiten múltiples flujos de control que se ejecutan de manera concurrente dentro de uno de sus programas y permiten también que su programa emprenda varias tareas de cómputo al mismo tiempo, una característica que da soporte al programa orientado a eventos.

Existen variaciones entre los lenguajes de programación concurrente:

- La estructura de un proceso puede ser estática o dinámica. Si el número de procesos es fijo y conocemos el tiempo de compilación tendrá una estructura estática. Si los procesos son creados en cualquier momento y el número de procesos se determina en tiempo de ejecución la estructura será dinámica.
- Distinguimos entre procesos “anidados” (multinivel) y “planos” (nivel más alto). Con la anidación en niveles es posible crear jerarquías de procesos y establecer relaciones entre ellos. En un proceso es útil distinguir entre el proceso (o bloque) que es responsable de su creación (relación padre/hijo) y el proceso (o bloque) que es afectado por su finalización (relación guardián/dependiente). Para que un programa termine, todos los procesos creados en él deben terminar.
- La inicialización puede ser con o sin paso de parámetros (comunicación explícita).

Los programas dinámicos, interactivos, controlados por eventos, suelen incluir varias partes activas que se ejecutan de manera independiente e interactúan o cooperan de alguna manera para alcanzar las metas [2].

Un programa de procesamiento multihilos tiene que coordinar varias actividades independientes y evitar la posibilidad de que interfieran entre sí. Un hilo es un flujo de control separado que ejecuta su propio código. En C/C++ POSIX el hilo es un objeto de la clase Pthread.



## 6.2. Gestión de hebras (*threads*)

A continuación describimos brevemente la utilización de las rutinas que permiten la creación y finalización de hebras en un programa C++ con las librerías POSIX [6,15].

### 6.2.1. Creación y finalización de hebras

POSIX define las siguientes rutinas básicas para la creación y finalización de *THREADS*:

---

```
int pthread_create(pthread_t * thread, pthread_attr_t
* attr, void * (*start_routine)(void *), void * arg);
void pthread_exit(void *status);
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

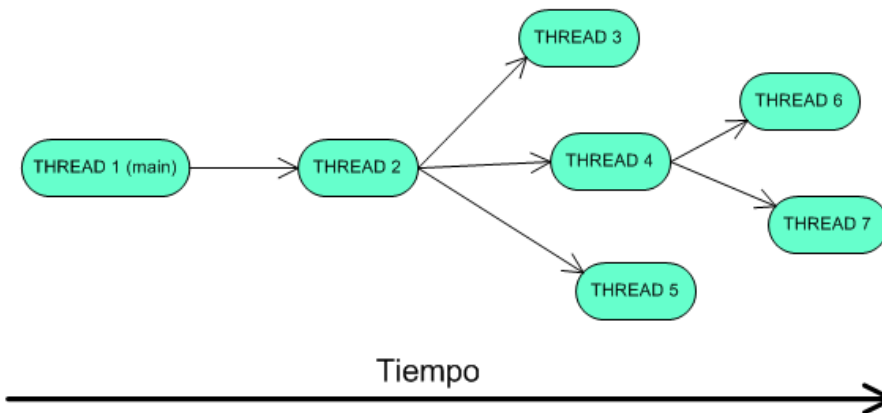
---

#### Creación de hebras:

- Inicialmente, la rutina *main()* ejecuta por defecto una hebra. Todas las demás hebras han de crearse explícitamente, programándolas mediante la rutina *pthread\_create*. Esta rutina se puede llamar un número indefinido de veces por programa, tantas como hebras diferentes se quieran crear (evidentemente limitado a los recursos disponibles del sistema).
- Argumentos de *pthread\_create*:
  - *thread*: Identificador interno de la nueva hebra creada. Es un parámetro de salida.
  - *attr*: Objeto interno que puede ser usado para establecer los atributos de la hebra. Se puede indicar el valor NULL para establecer los valores por defecto.
  - *start\_routine*: Indica la rutina (acción) que se ejecuta cuando se crea la hebra. La llamaremos rutina de inicio.
  - *arg*: Es el argumento que se le pasa a la rutina identificada por *start\_routine*. Se debe pasar por referencia como un apuntador (redefinido) de tipo void. Se puede indicar el valor NULL cuando no se quiera pasar ningún parámetro.
- Una vez creada una hebra, ésta puede a su vez crear otra hebra. No hay ninguna dependencia jerárquica implícita entre hebras creadas de esta forma. Asimismo, a menos que usemos mecanismos de planificación (prioridades), no hay un algoritmo preestablecido que nos indique cuándo se ejecutará una hebra.

### Pasando argumentos a una hebra:

La rutina `pthread_create()` permite pasar un argumento a la rutina de inicio. Para aquellos casos en que es necesario pasar más de un argumento, la solución es crear **una estructura que contenga todos los argumentos necesarios** y pasar un **puntero** a dicha estructura en la rutina `pthread_create()`.



### Atributos de una hebra:

- Una hebra se crea con unos atributos por defecto. Algunos de estos atributos se pueden modificar mediante programación, a través de las rutinas de gestión de atributos.
- Las rutinas **pthread\_attr\_init** y **pthread\_attr\_destroy** se utilizan para inicializar y destruir los atributos asociados a la hebra. Para modificar los atributos por defecto se deben utilizar otras rutinas. Dichas rutinas se discutirán más adelante.

### Finalización de hebras:

- Hay varias maneras de finalizar una hebra:
- Por retorno de la rutina llamada en la creación de la hebra.
- Por una llamada de la hebra a la rutina **pthread\_exit**.
- Por finalización de todo el proceso mediante una llamada a la subrutina EXIT.
- La rutina `pthread_exit` se puede usar para acabar explícitamente una hebra. Si la rutina **main()** finaliza con **pthread\_exit()** (recordemos que `main()` es la hebra que se crea por defecto), las otras hebras continuarán ejecutándose. Si `main()` finaliza de otra manera, todas las hebras se finalizarán automáticamente. Normalmente no se utiliza



explícitamente `pthread_exit`, sino que se deja que la hebra termine por retorno de rutina de inicio.

- La rutina `pthread_exit()` no libera recursos (memoria, ficheros abiertos, etc.).

### 6.2.2. *Join y detach* de hebras

Vamos a ver las siguientes rutinas para la gestión de hebras:

---

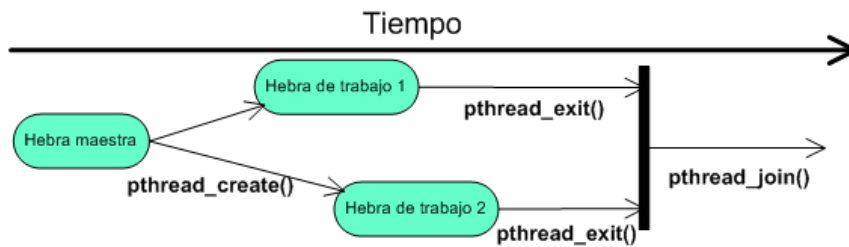
```
int pthread_join(pthread_t th, void **thread_return);
int pthread_detach(pthread_t th);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t
*attr, int *detachstate);
```

---

#### Joining:

- El “*Joining*” es un mecanismo que sirve para sincronizar la finalización de hebras. la rutina `pthread_join()` bloquea la hebra principal hasta que la hebra identificada por `th` finaliza. Por ejemplo:



- En este ejemplo, cuando se ejecuta la rutina `pthread_join()` la hebra maestra (*master*) queda a la espera de que finalice la hebra de trabajo (*worker*) para continuar con la ejecución. En caso de que se hubieran lanzado varias hebras de trabajo desde la hebra maestra, se podría indicar, mediante varias llamadas a `pthread_join` desde dicha hebra, que espere a la finalización de todas ellas antes de continuar.
- Cuando se crea una hebra, uno de sus atributos define si es *joinable* o no. Esto se puede efectuar mediante la rutina `pthread_attr_setdetachstate()`. Sólo las hebras que se han creado con el atributo “`PTHREAD_CREATE_JOINABLE`” pueden ser gestionadas por la rutina `pthread_join()`. Por defecto, al crear una hebra se crea con el atributo “*joinable*” aunque por motivos de portabilidad es

mejor establecer este atributo explícitamente si se quiere que las hebras sean joinable.

### Detaching:

- La rutina **pthread\_detach()** se puede usar para indicar que la finalización de la hebra no es susceptible de ser sincronizada mediante la rutina **pthread\_join()**. Cuando una hebra en estado *detach* finaliza, se liberan automáticamente los recursos que había adquirido.

### Establecer atributos:

- La rutina **pthread\_attr\_setdetachstate()** establece el atributo *detachstate*, el cual determina si la hebra puede ser usada o no por la rutina **pthread\_join()**. Los valores permitidos para este atributo son **PTHREAD\_CREATE\_DETACHED** o **PTHREAD\_CREATE\_JOINABLE**. Si se establece este último valor, la hebra puede ser *joinable*. En caso contrario no.

## Ejemplos

### Ejemplo 1:

En este ejemplo ([hello.cpp](#)) se crean 2 hebras (ver apartado 6.3 para configurar la estación de trabajo).

La rutina de inicio es "hello", definida de tipo (void \*), a la cual se le pasa como parámetro el puntero a la variable "i" redefinido como (void \*).

Observar que la rutina **pthread\_join()** espera a que finalicen las 2 hebras antes de seguir con la ejecución de **main()**, ya que espera unos segundos antes de que aparezca el mensaje de **system("pause");**

### Ejemplo 2:

En este ejemplo (esquemático) se muestra cómo pasar varios parámetros a la rutina de inicio "controller" mediante una estructura.

```
// creamos estructura para pase de parámetros
struct pthread_st {
    double X1;
    double X2;
    double X3;
};
...
...
int main () {
    // definimos una variable de tipo estructura creada
    pthread_st pthread_data;
    ...
}
```



```

// asignamos valores a los parámetros antes de crear la
hebra
pthread_data.X1:=1;
pthread_data.X2:=2;
pthread_data.X3:=3;
// pasamos como parámetro la dirección de la variable de
tipo estructura
// redefinida como puntero void.
pthread_create( &pthread_id, &attr, controller, (void *)
&pthread_data);
...
...
...
}
// recogida de parámetros en la rutina de inicio
void * controller (void *parms) {
double X;
double Y;
double Z;
// definimos una variable tipo puntero a estructura
pthread_st *pthread_dt;
// asignamos el valor del puntero pasado como parámetro
"parms", el cual
// 'apunta' a la variable pthread_data definida en main(),
a la
// variable local pthread_dt.
thread_data=(pthread_st *) parms; // redefinimos el tipo de
"parms"
// asignamos los valores pasados a varibales internas
// en X,Y,Z tendremos los valores 1, 2, 3.
X=thread_data->X1;
Y=thread_data->X2;
Z=thread_data->X3;
...
...
...
}

```

### 6.3. Problemas propuestos

Para poder utilizar la biblioteca POSIX se debe configurar el lugar de trabajo:

Los pasos necesarios para poder compilar nuestros programas en C++ con POSIX-threads son los siguientes:

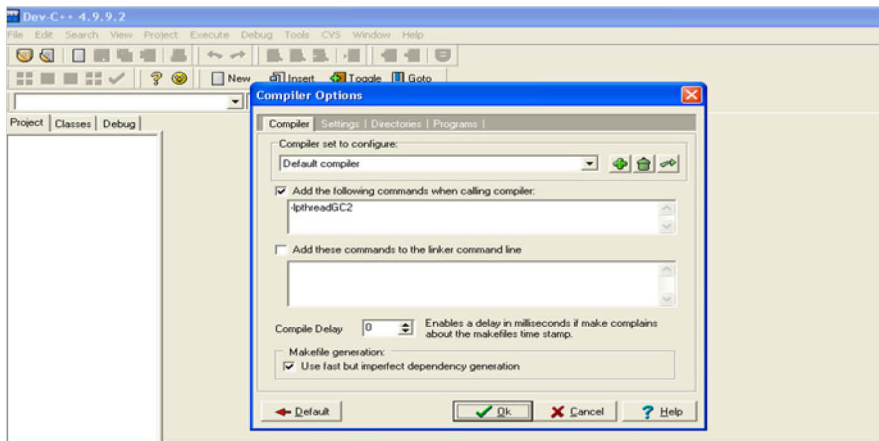
Paso 1. Copiar el archivo [libpthreadGC2.a](#), que es el archivo de biblioteca de enlace estático, en el directorio donde se encuentran las librerías de Dev-C++ (típicamente C:\Dev-Cpp\lib).

Paso 2. Copiar los archivos de cabecera `pthread.h`, `sched.h`, `semaphore.h` ([pthread.h](#), [sched.h](#) y [semaphore.h](#)) en el directorio donde se encuentran las “includes” de Dev-C++ (típicamente `C:\Dev-Cpp\include`).

Paso 3. Copiar el archivo [pthreadGC2.dll](#), que es la librería de enlace dinámico, en el directorio `system32` de **Windows** para que pueda ser encontrada cuando algún programa de los que compilemos necesite de sus servicios.

Paso 4. Ahora necesitamos configurar DEV-C++ para que sea capaz de resolver las referencias a las rutinas POSIX-threads. Para ello debemos indicar al compilador que añada el comando adicional `-lpthreadGC2` en la línea de comandos. Esto lo haremos mediante la opción de menú `Tools->Compiler Options`. En el cuadro de diálogo marcaremos el casillero “Add the following commands when calling compiler” de la pestaña “Compiler”, añadiendo el comando anterior (Fig. 6.3.1)

Fig 6.3.1.



Paso 5. **Programa de prueba.** Ahora podemos probar si la configuración ha sido correcta. Para ello cargamos el programa del ejemplo 1: `hello.cpp`.

El mensaje mostrado por el programa es el siguiente:

```
Creada hebra 1
Creada hebra 2
Iniciada hebra 2, HELLO WORLD!
Atributos destruidos.
Se queda en espera a finalizacion de las hebras
Iniciada hebra 1, HELLO WORLD!
Hebras finalizadas
Presione una tecla para continuar . . .
```

Se deberá modificar el programa anterior eliminando (comentar) las llamadas siguientes y describir en un documento los resultados obtenidos:

- llamada `pthread_exit`;
- llamada `pthread_join`.



### 6.3.1. Enunciado 1. Programación multihebra.

Partimos de un programa ejemplo (move\_armV0.cpp) que lanza 3 hebras. Cada hebra simula que da órdenes de movimiento sobre un determinado eje ( $x$ ,  $y$ ,  $z$ ) a un brazo mecánico. En este programa se incrementa indefinidamente en una unidad el valor de cada coordenada. Los valores de posición en que va pasando el brazo se registran en ficheros (uno para cada eje). Se limita el número de registros grabables en cada fichero (MAXREG) para evitar que los ficheros crezcan indefinidamente. Las hebras no finalizan (bucle infinito) hasta que lo hace el programa principal.

Deberá modificarse el programa de ejemplo de manera que el movimiento del brazo cumpla las siguientes reglas:

- La coordenada  $x$  debe estar entre los valores  $[-10, 10]$ .
- La coordenada  $y$  debe estar entre los valores  $[0, 15]$ .
- La coordenada  $z$  tiene un valor constante igual a 20.
- La siguiente posición de las coordenadas  $x$ ,  $y$  se obtiene incrementando o disminuyendo el valor de la posición anterior en una unidad, en función de que se haya llegado al límite superior o inferior de cada coordenada.
- Inicialmente el brazo se encuentra en la posición  $(0,0,20)$ .
- 

### 6.3.2. Enunciado 2. Movimiento del péndulo.

- Consideremos un péndulo compuesto por una barra de longitud  $L$  y una bola de masa  $m$ . La ecuación de movimiento (1) que nos da el ángulo  $\theta(t)$  en función del tiempo es:

$$\theta(t) = \theta_0 * \cos\left(t / \sqrt{\frac{L}{g}}\right) \quad (1)$$

donde:

- $\theta_0$  es el ángulo inicial (en radianes) correspondiente a la amplitud máxima del péndulo.
- $g$  es la gravedad ( $9.81 \text{ m/s}^2$ ).

Para calcular la energía cinética y potencial de la bola de masa  $m$  que está suspendida del brazo y la trayectoria que dicha bola efectúa sobre el plano  $x$ ,  $y$  aplicaremos las siguientes ecuaciones:

Las energías cinética y potencial se calculan a partir de:

$$E_p = m \cdot g \cdot h = m \cdot g \cdot L(1 - \cos(\theta_t)) \quad (2)$$

$$E_c = m \cdot g \cdot L \cdot (\cos(\theta_i) - \cos(\theta_0)) \quad (3)$$

Los valores de  $x$  e  $y$  se calcularán a partir de:

$$x = L \cdot \sin(\theta_i) \quad (4)$$

$$y = L \cdot (1 - \cos(\theta_i)) \quad (5)$$

Estas expresiones son válidas para un péndulo centrado en el eje  $x$  ( $x=0$ ). Si el péndulo está desplazado una distancia  $d$  en el eje  $x$ , las ecuaciones son:

$$x = d + L \cdot \sin(\theta_i) \quad (6)$$

$$y = L \cdot (1 - \cos(\theta_i)) \quad (7)$$

Definir un programa en C++ que calcule la energía cinética total de los 2 péndulos mostrados en la figura 6.3.2 y **determine el instante  $t$  en que dicha energía es máxima**. El programa debe solicitar la entrada de los siguientes parámetros:

- Ángulos iniciales  $\theta_1$ ,  $\theta_2$  validando que dichos valores sean mayores que 0 y menores o iguales a  $\pi/2$ .
- Longitudes  $L_1$  y  $L_2$  de los brazos de cada péndulo. La longitud debe ser mayor que 0.
- Masas  $m_1$  y  $m_2$  de cada péndulo.

-

A partir de las ecuaciones (1) y (3), el programa debe calcular los valores de  $\theta(t)$  y  $E_c$  de cada péndulo en cada intervalo de tiempo, partiendo de  $t=0$  y con incrementos de tiempo  $\Delta t$  de 0,01 segundos. Los valores de  $t$ ,  $\theta_1$ ,  $E_{c1}$ ,  $\theta_2$  y  $E_{c2}$  se deberán almacenar en los campos de una estructura llamada `péndulo`, organizados como una matriz `M` de dimensión [1000]. El programa finalizará cuando el número de intervalos almacenados sea igual al número de filas de la matriz (1000).

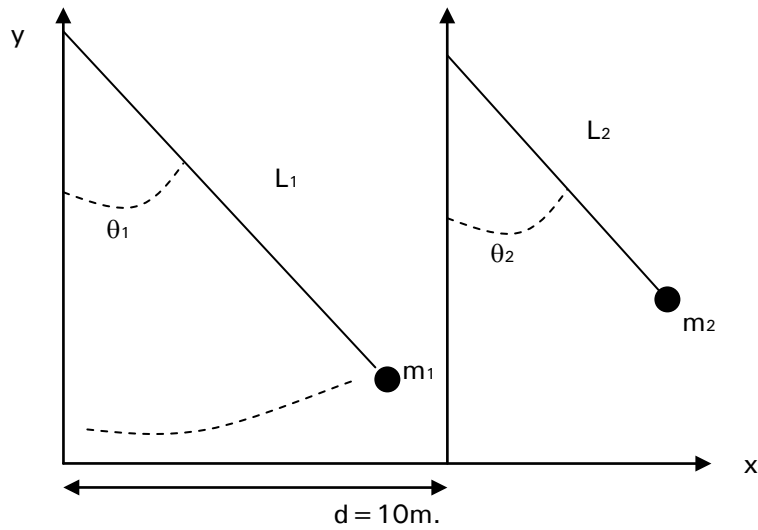
Se debe construir un programa `main()` que solicite la entrada de los valores del ángulo en radianes, la longitud del brazo y la masa de cada péndulo. Se debe comprobar que los valores introducidos sean correctos. El programa `main()` deberá utilizar las siguientes funciones y acciones:

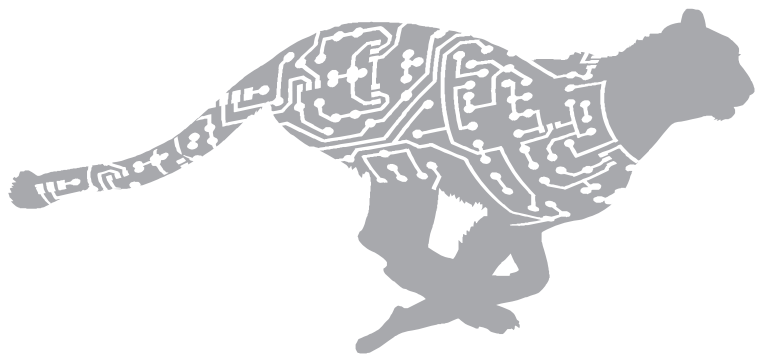
- `void llena_matriz(péndulo m[M]):` Esta acción pide al usuario que introduzca los valores  $\theta_0$ ,  $L$  y  $m$  (masa) de cada péndulo, valida sus valores y llena la matriz `m` con la información de los campos de la estructura `péndulo`.
- `double calcula_angulo(double t, double theta0, double l):` Esta función calcula el ángulo ( $\theta$ ) en cada instante de tiempo.



- `double calcula_energia(double theta, double theta0, double m, double l)`: Esta función calcula la energía cinética correspondiente al ángulo  $\theta$  (instante  $t$ ) .
- `void busca_elemento(pendulo m[M])`: Esta acción busca el elemento de la matriz en que la energía cinética total es máxima y presenta el instante  $t$  y el valor en pantalla.
- Definir dos hebras (una por cada péndulo). Deberá calcularse la energía cinética y potencial de cada péndulo.

Fig. 6.3.2.  
Movimiento de 2  
péndulos separados por  
una distancia de 10 m.





→7



# Sincronización

## 7.1. Introducción

Cuando en un sistema se están ejecutando varios procesos y uno de ellos puede afectar a los demás o verse él afectado por los otros hablaremos de un proceso cooperativo. Estos procesos pueden compartir datos y código mediante hilos. El acceso concurrente a datos compartidos puede ocasionar inconsistencia de datos. Para evitarla se utilizan mecanismos que garanticen la consistencia a través de la ejecución ordenada de procesos cooperativos o hilos que comparten un espacio lógico de direcciones (datos y código).

Se debe evitar la condición de competencia que se produce cuando varios hilos acceden a los mismos datos y los modifican de forma concurrente, y los resultados de la ejecución dependerán del orden de acceso. Para lograrlo se deben utilizar los mecanismos de sincronización y coordinación de hilos que describiremos en los siguientes capítulos.

El comportamiento correcto de un programa concurrente depende estrechamente de la sincronización y la comunicación entre procesos.

La sincronización consiste en satisfacer las restricciones en el entrelazado de las acciones de diferentes procesos. Una acción particular de un proceso sólo tiene lugar después de una acción específica de otro proceso.

La sincronización debe llevar simultáneamente a dos procesos a estados predefinidos.

La comunicación consiste en pasar información de un proceso a otro. Los dos conceptos están ligados, puesto que algunas formas de comunicación



requieren sincronización y la sincronización puede ser considerada como comunicación sin contenido.

Cuando existe un conjunto de procesos secuenciales cooperativos o de hilos que comparten datos, se debe proporcionar un mecanismo que garantice la coherencia de datos. La exclusión mutua, por ejemplo, impide la ocurrencia de una competencia por los datos compartidos. Se debe asegurar que no haya dos procesos o hilos utilizando una sección crítica del código simultáneamente. Se debe garantizar que si un proceso utiliza una variable o algún recurso compartido, los demás no puedan usarlos al mismo tiempo.

La comunicación entre procesos se basa normalmente en el uso de variables compartidas o en el paso de mensajes. Aunque las variables compartidas parecen una forma directa de pasar información entre procesos, su uso debe ser restringido debido a los problemas de actualización.

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 conjunto de hilos sección crítica. El sistema debe tener la característica de que cuando se está ejecutando en su sección crítica no debe permitir que otros hilos se ejecuten en esa misma sección. La ejecución de secciones críticas por los hilos es *mutuamente exclusiva* en el tiempo. El problema de la sección crítica es encontrar un mecanismo de cooperación que puedan usar los hilos. Este mecanismo debe cumplir con los siguientes requerimientos:

- **Exclusión mutua:** En un sistema que consta de  $n$  hilos  $\{T_0, T_1, \dots, T_n\}$ , si el hilo  $T_i$  se está ejecutando en su sección crítica, ningún otro hilo puede estar en la misma ejecución.
- **Progreso:** Cuando ningún hilo se está ejecutando en su sección crítica y algunos hilos desean entrar en sus secciones críticas, los que no están ejecutando su sección no crítica podrán participar en la decisión acerca de cuál entrará a su sección crítica a continuación. Esta situación no puede prolongarse de forma indefinida.
- **Espera limitada:** Debe existir un límite sobre el número de veces que otros hilos tienen permitido entrar en su sección crítica después de que un hilo ha hecho una solicitud para entrar en su sección crítica, y antes de que se conceda dicha solicitud. Este límite evita la inanición individual.

Si dos procesos no comparten variables, no se necesita exclusión mutua. La sincronización condicionada o de condición, es necesaria cuando un

proceso debe realizar una operación si otro proceso ya ha realizado una acción o está en un estado definido. Un ejemplo de sincronización de condición es el que se produce en la programación concurrente cuando dos procesos intercambian datos y no se comunican directamente sino mediante un búfer (sistema productor-consumidor).

El sistema operativo proporciona mecanismos como semáforos, mutex y variables de condición para controlar el acceso a datos compartidos. Muchos ordenadores proporcionan instrucciones especiales de hardware que permiten modificar o intercambiar contenidos de forma atómica.

## 7.2. Gestión de la exclusión mutua (mutex)

Mutex es una abreviación de “mutual exclusion”. Las rutinas mutex ofrecen una de las principales herramientas para implementar sistemas de sincronización de hebras y de protección de variables compartidas cuando se accede en modo escritura desde varios hilos de forma simultánea. Cuando hablamos de mutex nos referimos en realidad a la variable asociada a la exclusión mutua o “mutex variable”.

El mutex actúa como una llave, protegiendo el acceso a recursos compartidos. El concepto básico subyacente a las rutinas mutex usadas en Pthread es que únicamente una hebra puede tener el control o bloquear una variable mutex en un instante del tiempo. Es decir, si varias hebras intentan simultáneamente tomar el control de la variable, únicamente una de ellas lo conseguirá, poniéndose las demás hebras en “cola de espera” hasta que el mutex sea desbloqueado por la hebra propietaria.

Los mutex se pueden utilizar para evitar situaciones inconsistentes. En una situación, por ejemplo, en la que 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     | Ingresar: 200               |                             | 8000   |
| T4     |                             | Ingresar: 200               | 8000   |
| T5     | Actualiza saldo: 8000 + 200 |                             | 8200   |
| T6     |                             | Actualiza saldo: 8000 + 200 | 8200!! |



En este ejemplo, la hebra que actualiza el saldo debe bloquear el mutex antes de la lectura del saldo y desbloquearlo una vez actualizado su valor. A menudo las acciones que efectúa una hebra propietaria de un mutex consisten en actualizar variables globales. Ésta es la manera de garantizar que el resultado de la actualización será el mismo que se hubiera producido si solamente tuviéramos una hebra en ejecución.

La secuencia típica de instrucciones de un mutex es la siguiente:

- Inicializar y crear el mutex.
- Varias hebras intentan bloquear el mutex.
- Únicamente una de ellas lo consigue, pasando a ser la hebra propietaria.
- La hebra propietaria efectúa una serie de acciones.
- La hebra propietaria desbloquea el mutex.
- Otra hebra toma el control del mutex y se repite la secuencia.
- Finalmente se destruye el mutex.
- 

### 7.3. Creación y destrucción de la exclusión mutua

POSIX define las siguientes rutinas para la inicialización y destrucción del MUTEX:

---

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutexattr_init( pthread_mutexattr_t *attr );

int pthread_mutexattr_destroy( pthread_mutexattr_t *attr );
```

---

#### Pthread\_mutex\_init:

Las variables mutex deben declararse con el tipo **pthread\_mutex\_t**, y deben inicializarse antes de poder trabajar con ellas. Cuando se crea, la variable mutex está en estado “desbloqueada”. Hay 2 maneras de inicializar una variable mutex:

1. Estáticamente, en el momento en que se declara la variable, mediante la macro `PTHREAD_MUTEX_INITIALIZER`. En este caso se inicializa con los valores por defecto: `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
2. Dinámicamente con la rutina `pthread_mutex_init()`. Este método permite especificar los atributos del mutex, `attr`.

La variable `attr` apunta al objeto que se utiliza para establecer las propiedades del mutex y debe ser del tipo `pthread_mutexattr_t`. Si se deja a `NULL`, el mutex se inicializa con las propiedades por defecto, con lo que el resultado es equivalente a inicializarlo estáticamente.

#### **pthread\_mutex\_destroy:**

La rutina `pthread_mutex_destroy()` debe usarse para destruir un mutex no bloqueado que no deba volver a utilizarse en el programa.

#### **pthread\_mutexattr\_init/destroy:**

Las rutinas `pthread_mutexattr_init()` y `pthread_mutexattr_destroy()` se usan para crear y destruir respectivamente los objetos "atributo de mutex".

`pthread_mutexattr_init()` inicializa el objeto apuntado por `attr` con los valores por defecto. Estos atributos actúan como parámetros adicionales que se pasan en la creación del mutex. Cuando un mutex es inicializado con sus atributos, dichos atributos se copian dentro del mutex por lo que, una vez inicializado éste, los atributos ya no son necesarios para el resto del programa y se pueden destruir.

Si se desean crear los mutex con los atributos por defecto, no es necesario inicializar explícitamente con `pthread_mutexattr_init()`.

## **7.4. Bloqueo y desbloqueo de la exclusión mutua**

POSIX define las siguientes rutinas para el bloqueo y el desbloqueo de MUTEX:

---

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

---

#### **pthread\_mutex\_lock:**

La rutina `pthread_mutex_lock()` se utiliza para que una hebra solicite la propiedad del mutex especificado por la variable `mutex`. Si el mutex esta bloqueado en ese instante por otra hebra, la llamada a la rutina bloqueará la

ejecución de la hebra solicitante hasta que el mutex sea desbloqueado por la hebra propietaria.

#### **Pthread\_mutex\_trylock:**

**pthread\_mutex\_trylock()** intenta bloquear el mutex especificado por la variable *mutex*. Sin embargo, si el mutex está bloqueado, la rutina retorna de forma inmediata a la hebra solicitante (sin esperar a su desbloqueo), pero con un código de retorno que indica que el mutex está bloqueado. Esta rutina puede ser útil para evitar situaciones de “*deadlock*”.

#### **Pthread\_mutex\_unlock:**

**pthread\_mutex\_unlock()** desbloquea el mutex si es llamado por la hebra que lo ha bloqueado (hebra propietaria del mutex). Se debe llamar a esta rutina una vez la hebra ha completado las instrucciones que debían ser protegidas de accesos concurrentes. Se produce un error si el mutex ya está desbloqueado o bien la hebra no es propietaria del mutex.

Cuando varias hebras estan esperando el desbloqueo de un mutex para adquirirlo como propietarias, a menos que se haya aplicado algún mecanismo de planificación (priorización), la asignación del mutex a una u otra hebra se delega al sistema operativo, que la realizará de una forma más o menos aleatoria.

Es responsabilidad del programador hacer un buen uso de la utilización del mutex. En el siguiente ejemplo (en el que hay una hebra para la cual no se ha especificado mutex) se ha producido un error de lógica de programación ya que el resultado final será aleatorio:

| Tiempo | Hebra 1 (*)        | Hebra 2 (**)         | Hebra 3 (***)        |
|--------|--------------------|----------------------|----------------------|
| t1     | Lock               | Lock                 |                      |
| t2     | A = 2              | A = A + 1            | A = A * B            |
| t3     | Unlock             | Unlock               |                      |
|        | (*) Prioridad alta | (**) Prioridad media | (***) Prioridad baja |

### **7.5. Ejemplo: Uso de la exclusión mutua**

En este programa se muestra la utilización de mutex para garantizar la exclusividad en la actualización de variables globales.

Se trata de un ejercicio a efectos meramente didácticos.

Se toma el valor de la variable “saldo” al inicio del bucle que simula que la CPU está trabajando y se incrementa el saldo con el valor “ingreso”.

Deberemos observar el comportamiento del programa con mutex y sin mutex. [ejemplomutex.cpp](#).

## 7.6. Problemas propuestos

### 7.6.1. Enunciado 1. Vasos comunicantes.

Un sistema de vasos comunicantes se produce cuando se ponen en comunicación dos depósitos que contienen un mismo líquido que inicialmente está a distinta altura en cada uno. El nivel de uno de los depósitos baja mientras sube el del otro hasta que ambos se igualan.

Basándonos en este principio simularemos un sistema de 2 tanques cilíndricos de radio  $R_1$  y  $R_2$  conectados mediante una tubería. Suponiendo que  $R_1 \leq R_2$ , que la altura de los tanques es  $h$  y que el primero de ellos (radio =  $R_1$ ) se llena completamente de líquido, calcular la altura  $h'$  del líquido a la que se estabilizará el sistema de vasos comunicantes. Suponemos que el volumen de agua dentro de la tubería es despreciable (ver figura 7.6.1). Recordamos que el volumen de un cilindro es  $\pi \cdot R^2 \cdot h$ .

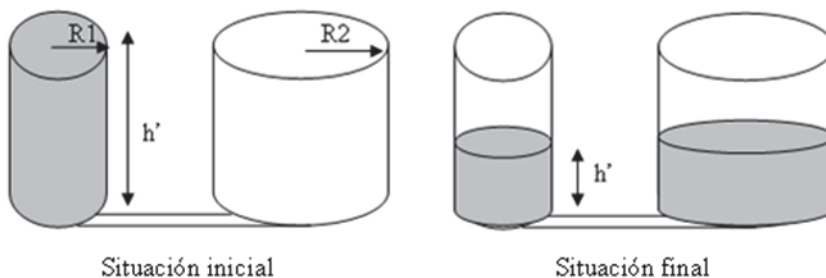


Fig.7.6.1.  
Vasos comunicantes.

Planteamos la solución de la siguiente manera: por cada disminución de altura  $\Delta h_1$  del nivel del líquido del cilindro  $R_1$ , se incrementa el nivel de la altura del líquido del cilindro  $R_2$  en  $\Delta h_2$ , siendo  $\Delta h_2 = (R_1^2 \cdot \Delta h_1) / R_2^2$ . Iremos disminuyendo el nivel inicial  $h$  del tanque  $R_1$  hasta que los niveles de los 2 tanques sean iguales ( $h'$ ). Admitimos un margen de error de 0.001 unidades.

En el programa main se deberá incluir:

- Una función `double incremento(double h1, double R1, double R2)` que calcula el incremento de nivel  $h_2$  en función de la disminución  $h_1$ .
- Una función recursiva `double nivelar(double error, double h, double R1, double R2)` que da el nivel en que se estabiliza.
- Se debe crear una hebra por cada tanque y alternar el llenado utilizando una acción de control que permita rellenar de forma



simultánea los dos tanques a partir del momento en que se ha logrado nivelarlos por la mitad.

### 7.6.2. Enunciado 2. Círculo y cuadrado.

Construir un programa que calcule el lado  $L$  del cuadrado cuya superficie sea aproximadamente igual a la de un círculo de radio  $R$ . Se solicitan como datos de entrada el radio  $R$  y la diferencia máxima permitida ( $D$ ) entre las 2 superficies. El valor de  $D$  puede ser arbitrariamente pequeño.

Utilizaremos las siguientes funciones:

- `double calcula_area_circulo (double R)` → calcula el área del círculo de radio  $R$ .
- `double calcula_area_cuadrado (double L)` → calcula el área del cuadrado de lado  $L$ .
- `double diferencia_area (double R, double L)` → obtiene la diferencia entre el área del círculo de radio  $R$  y el cuadrado de lado  $L$ .

Aplicaremos el siguiente algoritmo en la función `main`:

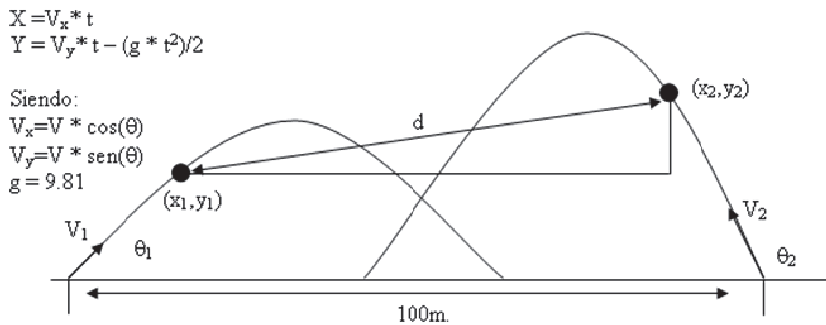
Partiendo de un cuadrado de lado  $L = R$  y del valor inicial de la variable `inc =  $R/2$` , incrementamos  $L$  en una cantidad `inc` en caso que la función `diferencia_area` retorne un valor positivo o bien disminuimos el valor de  $L$  en la negativo. Cada vez que detectamos un cambio de signo entre el valor anterior de la diferencia y el valor actual de la misma, dividimos por 2 el valor de `inc`. El programa se acaba cuando el valor absoluto de la diferencia entre las áreas es menor o igual a  $D$ .

Se deben crear dos hebras que representen dos cuadrados, e ir alternando el cálculo de las áreas respectivas.

### 7.6.3. Enunciado 3. Tiro parabólico.

Supongamos que tenemos 2 lanzadores de bolas separados por una distancia de 100 metros. Simultáneamente se lanza una bola de cada lanzador con ángulos  $\theta_1$  ( $0 < \theta_1 < \pi/2$ ) y  $\theta_2$  ( $\pi/2 < \theta_2 < \pi$ ) y velocidades  $V_1$  y  $V_2$  respectivamente (ver figura 7.6.2). Se trata de diseñar un programa que calcule la distancia entre los dos puntos de la trayectoria en los que se alcanza la altura máxima. Suponemos despreciable el rozamiento y consideramos las bolas como puntos sin dimensión. En estas circunstancias se pueden aplicar las ecuaciones del movimiento parabólico:

Fig. 7.6.2  
Ecuaciones del  
movimiento  
parabólico



La distancia entre las 2 bolas en cada posición es  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ . Para encontrar el punto correspondiente a la altura máxima, debemos calcular el máximo local de cada trayectoria. Debemos tener en cuenta que el valor de "y" no puede ser negativo (el movimiento de una bola termina cuando la coordenada "y" de la misma tiene el valor 0 después del instante inicial). Consideramos intervalos de tiempo de 0.001 segundos para calcular cada punto de la trayectoria.

#### Tareas que realizar.

- Definición de estructuras.

Para almacenar la posición de la bola:

```
typedef struct {
double x;      //Posición X del punto
double y;      //Posición Y del punto
} POSICION;
```

Para almacenar los componentes de la velocidad de la bola:

```
typedef struct {
double Vx;     //Componente x de la velocidad
double Vy;     //Componente y de la velocidad
} VELOCIDAD;
```

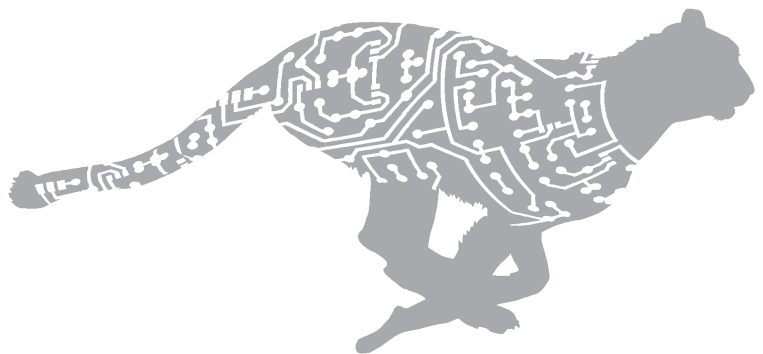
- Definir como constantes las siguientes variables:
  - Intervalos de tiempo: intervalo 0.001.
  - El valor de  $\pi$  PI 3.14159265.
  - La fuerza de gravedad como: g 9.81.
- Definir las siguientes funciones con sus respectivos prototipos:
- Función que convierte un ángulo de grados en radianes **double**  
**radian(double ang)**



- Función que calcula la trayectoria de las bolas en cada punto de la trayectoria (posiciones  $X1$ ,  $X2$ ,  $Y1$ ,  $Y2$ ) en función del tiempo y devuelve la distancia entre las dos bolas cuando están en la altura máxima. Suponemos que el origen de coordenadas está en el lanzador 1 (posición inicial de la bola 1). En el momento en que una bola vuelve a “tocar tierra” ( $y=0$ ), se mantiene en la misma posición.  
**double trayectoria(double t, POSICION Pos [2], VELOCIDAD Vel [2], double& dist);**

Definir un programa main que:

- Utilice dos vectores del tipo velocidad y posición.
- Pida al usuario introducir:
  - o Para la bola 1: la velocidad inicial mayor que cero y el ángulo en grados comprendido entre  $0 < \text{angulo} < 90$ ;
  - o Para la bola 2: la velocidad inicial mayor que cero y el ángulo en grados comprendido entre  $90 < \text{angulo} < 180$ .
- Calcular las componentes  $V_x, V_y$  de cada bola utilizando las fórmulas:
  - o  $V_x = V * \cos(\theta)$ ;
  - o  $V_y = V * \sin(\theta)$ .
  - o Los ángulos deben convertirse a radianes.
- Se debe visualizar la distancia en un mensaje cuando las bolas alcanzan la altura máxima en sus trayectorias respectivas.
- Se debe crear una acción que controle una hebra por cada bola y permita realizar los cálculos de la distancia alternando ambas hebras.







# Monitores

## 8.1. Introducción

Las secciones consideradas críticas se escriben como procedimientos, y están encapsuladas en un único módulo, llamado monitor. Como módulo, oculta todas las variables a las que se deba acceder bajo exclusión mutua y, como monitor, garantiza la ejecución con exclusión mutua de todas las llamadas a procedimientos de módulo.

Aunque el monitor proporciona la exclusión mutua, existe todavía una necesidad de sincronización de condición en él. Podrían seguir utilizándose semáforos, pero normalmente se incluye una primitiva de sincronización más sencilla, llamada *variable de condición*.

Las variables de condición se manejan mediante dos operadores que, por sus similitudes con la estructura de semáforo, serían llamados "*wait*" y "*signal*". Cuando un proceso lanza una operación *wait*, se bloquea (queda suspendido) y se ubica en una cola asociada a esa variable de condición (comparable con un *wait* en un semáforo con valor cero).

Un *wait* sobre una variable de condición siempre bloquea, a diferencia de un *wait* en un semáforo. Cuando un proceso bloqueado libera su bloqueo de exclusión mutua en el monitor, permitirá entrar a otro proceso.

Cuando un proceso ejecuta una operación *signal*, liberará un proceso bloqueado. Si no hay ningún proceso bloqueado en la variable especificada, entonces el operador *signal* no tiene efecto.

El significado de *wait* y *signal* para monitores es similar al de suspender y reanudar.



## 8.2. Exclusión mutua y variables de condición POSIX (*Condition Variables*)

Las variables de condición nos ofrecen un mecanismo adicional para sincronizar hebras. Mientras los mutex implementan la sincronización mediante el control de acceso de una hebra a determinados datos o instrucciones, las variables de condición permiten la sincronización basándose en el contenido actual de las variables.

El uso de variables de condición permite chequear el contenido de variables sin necesidad de ir leyendo continuamente el valor de las mismas, evitando de esta manera tener que dedicar recursos innecesarios que consumen CPU.

El mecanismo de las variables de condición se usa siempre conjuntamente con un mutex según la secuencia de acciones descrita a continuación:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Hebra principal</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <ul style="list-style-type: none"> <li>- Declaración e inicialización de variables globales;</li> <li>- Declaración e inicialización de la variable de condición;</li> <li>- Declaración e inicialización del mutex asociado a la variable de condición;</li> <li>- Creación de las hebras (por ejemplo A y B).</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Hebra A</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <b>Hebra B</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <ul style="list-style-type: none"> <li>- Ejecutar el código hasta el punto en el que se debe cumplir alguna condición dependiendo del valor de la variable global.</li> <li>- Bloquear el mutex asociado a la variable de condición con <code>pthread_mutex_lock</code> y comprobar el valor de la variable global.</li> <li>- Si la variable global tiene un valor determinado, llamar a la rutina <code>pthread_cond_wait()</code> para bloquear la hebra, en espera de que la hebra B envíe una señal de desbloqueo. La llamada a <code>pthread_cond_wait</code> desbloquea de forma automática el mutex asociado a la variable. De esta manera, dicha variable puede ser utilizada por la hebra B.</li> <li>- Cuando la hebra B envía la señal con la rutina <code>pthread_cond_signal</code>, continúa la ejecución y el mutex se vuelve a bloquear de forma automática</li> <li>- Desbloqueo explícito del mutex. Continúa la ejecución de la hebra A.</li> </ul> | <ul style="list-style-type: none"> <li>- Ejecutar el código.</li> <li>- Bloquear el mutex asociado a la variable de condición.</li> <li>- Cambio del valor de la variable global utilizada por la hebra A, que determina que dicha hebra haya lanzado la rutina <code>pthread_cond_wait()</code>.</li> <li>- Si el valor de la variable global ha alcanzado el valor esperado, envía una señal <code>pthread_cond_signal()</code> a la hebra A para avisar de que puede continuar con la ejecución.</li> <li>- Desbloqueo del mutex. Continúa la ejecución de la hebra B.</li> </ul> |
| <b>Hebra principal</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Join/Continuar                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

### 8.3. Creación y destrucción de variables de condición

POSIX define las siguientes rutinas para la creación y destrucción de VARIABLES DE CONDICIÓN:

---

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t*cond,
pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

---

#### Pthread\_cond\_init/destroy:

Las variables de condición deben declararse del tipo `pthread_cond_t` y deben inicializarse antes de su utilización. Hay 2 maneras de hacerlo:

- Estáticamente, con los atributos por defecto cuando se declaran mediante la macro `PTHREAD_COND_INITIALIZER`. Ejemplo:  
*pthread\_cond\_t convar = PTHREAD\_COND\_INITIALIZER;*
- Dinámicamente con la rutina `pthread_cond_init()`. El identificador interno de la condición se retorna a la hebra que llama a la rutina mediante el parámetro *cond*. Este método permite asignar atributos a la variable de condición o bien utilizar la palabra NULL para dejar los atributos por defecto (equivalente al punto anterior).

La rutina `pthread_cond_destroy()` debe utilizarse para liberar una variable de condición que no volverá a usarse.

### 8.4. Espera y señalización de variables de condición

POSIX define las siguientes rutinas para indicar espera y señal de VARIABLES DE CONDICIÓN:

---

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
*mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

---

#### Pthread\_cond\_wait:

La rutina `pthread_cond_wait()` bloquea la hebra que la invoca hasta que la variable de condición *cond* sea liberada por otra hebra mediante la llamada a la rutina `pthread_cond_signal()`. Debe llamarse a `pthread_cond_wait` una vez el mutex asociado ha sido bloqueado; en caso contrario se produce un error. Una vez bloqueada la hebra, el mutex asociado se libera automáticamente mientras dicha hebra está en espera. Después de recibir la señal, el mutex se vuelve a bloquear automáticamente y la hebra que estaba en espera reanuda su ejecución. Esto significa que el programador debe desbloquear el mutex antes de que la hebra finalice.



### **Pthread\_cond\_signal/broadcast:**

La rutina **pthread\_cond\_signal()** se utiliza para señalar o “despertar” a otra hebra que estaba en espera de la variable de condición. Esta rutina debe incluirse entre una secuencia de bloqueo/desbloqueo del mutex asociado a la variable de condición.

La rutina **pthread\_cond\_broadcast()** realiza la misma función que **pthread\_cond\_signal** pero se puede utilizar si queremos “despertar” a más de una hebra de las que estaban en espera. Si invocamos la rutina **pthread\_cond\_signal()** cuando no hay ninguna hebra en espera como resultado de la llamada a **pthread\_cond\_wait()**, se produce un error.

## **8.5. Programa de ejemplo**

Éste es un programa sencillo de utilización de variables de condición. Se crean 5 hebras que quedan en espera mientras el valor de la variable `condicionOK` sea 0. En el programa principal se efectúa **pthread\_cond\_broadcast()** para “despertar” a las hebras, pero vuelven a quedarse a la espera mientras el valor de `condicionOK` sea 0.

**Notar** que no hay un orden preestablecido para “despertar” a las hebras.

Analizar la secuencia de bloqueo/desbloqueo del mutex tanto desde la hebra creada como desde `main` ([ejemplocondvariables.cpp](#)). Otro ejemplo es [Condvar\\_2.cpp](#).

## **8.6. Problemas propuestos**

### **8.6.1. Enunciado 1.**

Describe brevemente el funcionamiento del programa para el ejemplo de mutex [ejemplomutex.cpp](#) y para el ejemplo de variables de condición [ejemplocondvariables.cpp](#).

### **8.6.2. Enunciado 2.**

Describe el comportamiento del programa [ejemplocondvariables.cpp](#) con las siguientes modificaciones:

- Utilizando el `Sleep(1)` en la acción `void *inc_contador(void *idp);`
- Sin el `Sleep(1)` en la acción `void *inc_contador(void *idp);`
- Sin el condicional `contador < contador_LIMIT` en la acción `void *watch_contador(void *idp);`
- Con el condicional `contador < contador_LIMIT` void `*watch_contador(void *idp);`

### 8.6.3. Enunciado 3.

Definir un programa que utilice el mecanismo de variables de condición para obtener el movimiento de un brazo mecánico sobre el eje z que dependa de las posiciones sobre los ejes x e y. Se deben modificar las reglas de movimiento en el programa [move\\_armVO.cpp](#) de la siguiente manera:

- La coordenada x debe estar entre los valores [-10, 10].
- La coordenada y debe estar entre los valores [0, 15].
- La coordenada z toma el valor  $x + y$  siempre y cuando el valor de x sea mayor o igual a 0 y el valor de y sea menor o igual que 10. Si no se cumple alguna de las 2 condiciones anteriores, el valor de z no se modifica. En el resto de casos, hay que recalculr el valor de z cada vez que se modifica el valor de x o el de y.
- La siguiente posición de las coordenadas x, y se obtiene incrementando o mermando el valor de la posición anterior en una unidad en función de que se haya llegado al límite superior o inferior de cada coordenada.
- Inicialmente el brazo se encuentra en la posición (0,0,0).

A continuación se presenta un ejemplo de resultado esperado. Observar que el valor de z se recalcula cada vez que hay un cambio en x o y, excepto si  $x < 0$  ó  $y > 10$ .

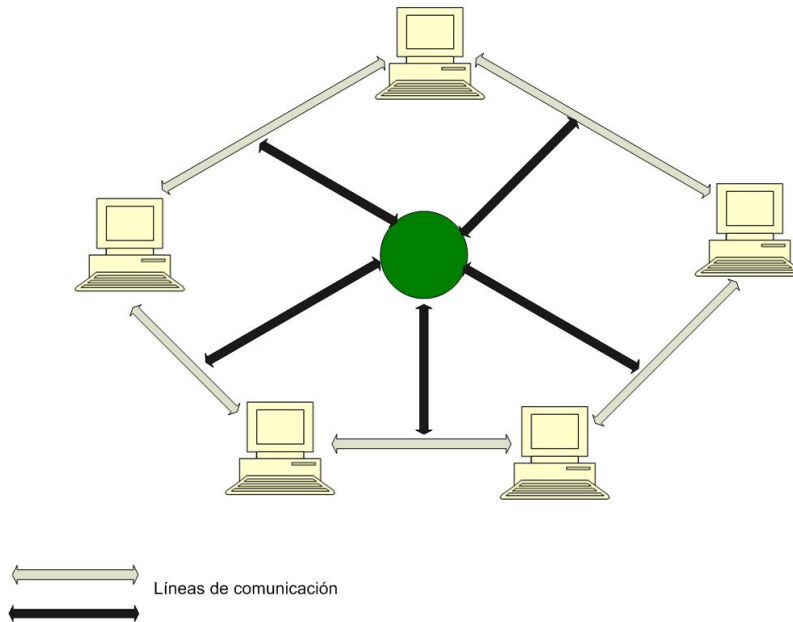
| Fichero ejex.txt      | Fichero ejey.txt      | Fichero ejez.txt                       |
|-----------------------|-----------------------|----------------------------------------|
| [Id: 2 ] Coord x: 1   | [Id: 4 ] Coord y: 1   | [Id: 1 ] Coord z: 0 Xpos:0 Ypos: 0     |
| [Id: 6 ] Coord x: 2   | [Id: 8 ] Coord y: 2   | [Id: 3 ] Coord z: 1 Xpos:1 Ypos: 0     |
| [Id: 10 ] Coord x: 3  | [Id: 12 ] Coord y: 3  | [Id: 5 ] Coord z: 2 Xpos:1 Ypos: 1     |
| [Id: 14 ] Coord x: 4  | [Id: 16 ] Coord y: 4  | [Id: 7 ] Coord z: 3 Xpos:2 Ypos: 1     |
| [Id: 18 ] Coord x: 5  | [Id: 20 ] Coord y: 5  | [Id: 9 ] Coord z: 4 Xpos:2 Ypos: 2     |
| [Id: 22 ] Coord x: 6  | [Id: 24 ] Coord y: 6  | [Id: 11 ] Coord z: 5 Xpos:3 Ypos: 2    |
| [Id: 26 ] Coord x: 7  | [Id: 28 ] Coord y: 7  | [Id: 13 ] Coord z: 6 Xpos:3 Ypos: 3    |
| [Id: 30 ] Coord x: 8  | [Id: 32 ] Coord y: 8  | [Id: 15 ] Coord z: 7 Xpos:4 Ypos: 3    |
| [Id: 34 ] Coord x: 9  | [Id: 36 ] Coord y: 9  | [Id: 17 ] Coord z: 8 Xpos:4 Ypos: 4    |
| [Id: 38 ] Coord x: 10 | [Id: 40 ] Coord y: 10 | [Id: 19 ] Coord z: 9 Xpos:5 Ypos: 4    |
| [Id: 42 ] Coord x: 9  | [Id: 44 ] Coord y: 11 | [Id: 21 ] Coord z: 10 Xpos:5 Ypos: 5   |
| [Id: 46 ] Coord x: 8  | [Id: 48 ] Coord y: 12 | [Id: 23 ] Coord z: 11 Xpos:6 Ypos: 5   |
| [Id: 50 ] Coord x: 7  | [Id: 52 ] Coord y: 13 | [Id: 25 ] Coord z: 12 Xpos:6 Ypos: 6   |
|                       |                       | [Id: 27 ] Coord z: 13 Xpos:7 Ypos: 6   |
|                       |                       | [Id: 29 ] Coord z: 14 Xpos:7 Ypos: 7   |
|                       |                       | [Id: 31 ] Coord z: 15 Xpos:8 Ypos: 7   |
|                       |                       | [Id: 33 ] Coord z: 16 Xpos:8 Ypos: 8   |
|                       |                       | [Id: 35 ] Coord z: 17 Xpos:9 Ypos: 8   |
|                       |                       | [Id: 37 ] Coord z: 18 Xpos:9 Ypos: 9   |
|                       |                       | [Id: 39 ] Coord z: 19 Xpos:10 Ypos: 9  |
|                       |                       | [Id: 41 ] Coord z: 20 Xpos:10 Ypos: 10 |
|                       |                       | [Id: 43 ] Coord z: 19 Xpos:9 Ypos: 10  |
|                       |                       | [Id: 45 ] Coord z: 19 Xpos:9 Ypos: 11  |



#### 8.6.4. Enunciado 4.

Definir un programa que sincronice el funcionamiento de cinco servidores que necesitan compartir las líneas de comunicación que se encuentran ubicadas en medio de ellos como se muestra en la Figura 8.6.1. Los servidores se pasan información interna, transmitiéndola o recibiendo a través de esas líneas de comunicación.

Fig 8.6.1.  
Conexión de  
servidores.



Cada servidor tiene agentes que gestionan la información y una línea de comunicación a cada lado. Para gestionar la información son necesarias dos líneas de comunicación, una que recibe mensajes y otra que los envía. Cada servidor puede usar únicamente las líneas que están a su derecha y a su izquierda. Si cualquier servidor coge una línea y la otra está ocupada, se quedará esperando con la línea asignada hasta que pueda usar la otra línea para luego empezar a gestionar la información. Simulamos que el servidor está gestionando la información mediante un *sleep* de un milisegundo. Una vez transcurrido este tiempo se liberan las dos líneas.

El algoritmo que se debe implementar consiste en que cada servidor toma las líneas de forma individual. Se trata de definir un programa que permita que los servidores siempre puedan gestionar información. Se debe tener en cuenta que:

- Dos servidores contiguos no pueden transmitir información a la vez (*exclusión mutua*).
- El servidor que termina de transmitir debe ceder las líneas para su posterior utilización (*interbloqueo*).

Se deberá crear una rutina de tratamiento de hebra. Cada hebra simulará un servidor, por lo que deberán crearse cinco hebras.

#### **8.6.5. Enunciado 5.**

Diseñar un programa que simule la venta de entradas por internet. Se tienen 300 entradas disponibles y 4 vendedores. Para simular el proceso de venta, el número de entradas será un recurso que no podrán actualizar varios vendedores a la vez. Cada vendedor será una hebra. La cantidad de entradas disponible para vender podemos definirla como una variable global cuyo bloqueo dependerá de un semáforo responsable de garantizar que las hebras la actualicen de una en una.





# Semáforos

## 9.1. Introducción

Un semáforo es un entero no negativo que sólo puede ser actualizado mediante los procedimientos *wait* y *signal*. Las ejecuciones de dichos procedimientos son atómicas (indivisibles).

**wait(S):** Si el valor del semáforo, S, es mayor que cero, entonces disminuye su valor en uno; en caso contrario, demora el proceso hasta que S sea mayor que cero ( y entonces disminuye su valor).

**signal(S):** Incrementa el valor del semáforo, S, en uno.

La provisión de una primitiva de semáforo tiene como consecuencia la introducción de un nuevo estado para un proceso, llamado suspendido. También introduce dos nuevas condiciones de error:

- **Interbloqueo:** Un conjunto de procesos suspendidos no pueden continuar.
- **Aplazamiento indefinido:** Un proceso es incapaz de continuar mientras los recursos no estén disponibles para él (también llamado inanición; *lockout* o *starvation*).

Los semáforos son unos tipos de datos protegidos que se utilizan para el acceso exclusivo a recursos compartidos y para la sincronización. El concepto fue introducido por Dijkstra [10] a mediados de los sesenta.

Un semáforo tiene asociada una variable entera cuyo valor inicial no puede ser negativo. Además de asignarle un valor inicial, sobre el semáforo se

pueden realizar las operaciones wait y post. Sea *sval* el valor de la variable asociada al semáforo *sem*:

La operación wait (*sem*) actúa de la siguiente manera sobre el semáforo *sem*: Si *sval* > 0, disminuye en una unidad el valor de *sval*, Si *sval* ≤ 0, disminuye en una unidad el valor de *sval* y suspende la ejecución de la hebra hasta que *sval* > 0.

La operación post (*sem*) actúa de la siguiente manera sobre el semáforo *sem*: Si *sval* ≤ 0 y hay alguna hebra en espera bloqueada por el semáforo *sem*, lo “despierta” e incrementa el valor de *sval* en una unidad. Si *sval* > 0, únicamente incrementa el valor de *sval*.

Existen semáforos con y sin nombre. En nuestros ejemplos utilizaremos semáforos sin nombre. Para inicializar un semáforo sin nombre utilizaremos la función `int sem_init(sem, pshared, sval)`. Para inicializar un semáforo con nombre se utiliza la función `sem_t *sem_open()`. No profundizaremos más en las diferencias entre los 2 tipos de semáforos.

## 9.2. Ejemplo de utilización de semáforos

---

### Hebra principal

Declaración e inicialización de variables globales y tipos.

Inicializar semáforo (establecer valores de los parámetros *pshared* y *sval*).

Creación de las hebras (por ejemplo A y B).

---

#### Hebra A

- Ejecutar el código hasta el punto P1.
- Bloquear la ejecución de la hebra mediante `sem_wait(sem, sval)`. La hebra se bloquea en función del valor de la variable *sval* (si *sval* ≤ 0).
- Cuando la hebra B envía la señal con la rutina `sem_post`, continúa la ejecución de la hebra A.

#### Hebra B

- Ejecutar el código hasta el punto P2.
  - Enviar una señal al semáforo *sem* mediante la rutina `sem_post()`. Si el valor de *sval* antes de enviar la señal es ≤ 0 significa que había alguna hebra bloqueada por el semáforo.
  - Continúa la ejecución de la hebra B.
- 

### Hebra principal

Join/Continuar

---

## 9.3. Inicialización/destrucción de semáforos

POSIX define las siguientes rutinas para la inicialización y destrucción de SEMÁFOROS:

---

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, int value);
int sem_destroy(sem_t *sem);
```

---

### **Sem\_init/destroy:**

La rutina **sem\_init()** se utiliza para inicializar un semáforo sin nombre referenciado por la variable *sem*. El valor de la variable *value* es el que adquiere el semáforo después ser inicializado. Una vez inicializado y antes de ser destruido, el semáforo puede utilizarse para realizar llamadas a las rutinas **sem\_wait()**, **sem\_post()**, y **sem\_destroy()**.

Si el valor del argumento *pshared* es diferente de 0, significa que el semáforo puede usarse para ser compartido entre diferentes procesos. Si el valor de este argumento es 0, el semáforo puede utilizarse para ser compartido entre diferentes hebras del mismo proceso. En nuestras prácticas utilizaremos el valor 0 para este argumento.

Si se pretende inicializar un semáforo que ya está inicializado, el resultado de la llamada provocará un comportamiento indefinido del semáforo.

La rutina **sem\_destroy()** se utiliza para destruir un semáforo sin nombre referenciado por la variable *sem*. Únicamente se puede destruir un semáforo si se ha inicializado con la rutina **sem\_init()**. Si se pretende utilizar un semáforo que se ha destruido, el comportamiento del mismo quedará indefinido a menos que se vuelva a inicializar con **sem\_init()**.

Es importante asegurarnos de no destruir un semáforo que está bloqueando la ejecución de una hebra, pues el comportamiento del programa quedará indefinido.

## **9.4. Espera y señalización de semáforos**

POSIX define las siguientes rutinas para la inicialización y destrucción de SEMÁFOROS:

---

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
```

---

### **sem\_wait:**

La rutina **sem\_wait()** bloquea el semáforo referenciado por *sem*. Si el valor del semáforo (*sval*) es menor o igual a 0 (podemos obtener dicho valor



mediante la llamada a **sem\_getvalue**, que lo retorna en la variable *sva/*. Entonces, la hebra invocante no retorna de la llamada a **sem\_wait** quedando a la espera de recibir una señal de desbloqueo. Si el valor de *sva/* es positivo, entonces la llamada a **sem\_wait** no se queda a la espera y la hebra continúa la ejecución.

#### **sem\_post/ sem\_getvalue:**

La rutina **sem\_post()** desbloquea el semáforo referenciado por *sem* enviando una señal a dicho semáforo.

Si el valor del semáforo (*sva/*) después de ejecutar esta rutina es mayor que 0, significa que no había ninguna hebra bloqueada esperando la señal. En este caso, el valor del semáforo simplemente se incrementa en una unidad.

Si el valor del semáforo (*sva/*) después de ejecutar esta rutina es menor o igual a 0, significa que una de las hebras que estaba esperando la señal retorna de la llamada a **sem\_wait()**. Si se definen políticas de planificación, la hebra que se activa se selecciona en función de éstas. Por ejemplo, si hemos definido una política SCHED\_FIFO, se selecciona la hebra de mayor prioridad, y si hay varias en espera con la misma prioridad, se desbloquea la que lleva más tiempo esperando. En Windows POSIX no es posible definir ninguna política de planificación, únicamente SCHED\_OTHER.

La rutina **sem\_getvalue()** nos devuelve el valor del semáforo *sem* en la variable referenciada por *sva/*.

### **9.5. Programa de ejemplo de semáforos**

El programa [ProdConsV1.cpp](#) contiene el ejemplo típico del problema productor/consumidor que utiliza elementos de una lista encadenada. Es un ejemplo de mecanismo de sincronización entre 2 procesos (hebras) basado en un semáforo. La hebra “consumidora” debe esperar a que haya al menos un elemento en la lista, es decir, la lista no puede estar vacía.

Esto se consigue mediante **sem\_wait(&consume\_peticion)**, cuyo valor se inicializa a 0, con lo que se bloquea el semáforo.

La hebra “productora” debe añadir nuevos elementos a la lista mientras no esté llena y desbloquear el semáforo (**sem\_post**).

En el programa del ejemplo no se controla que la lista tiene un límite (está llena).

La lista se irá llenando si el tiempo de “proceso” de la hebra productora es inferior al de la hebra consumidora.

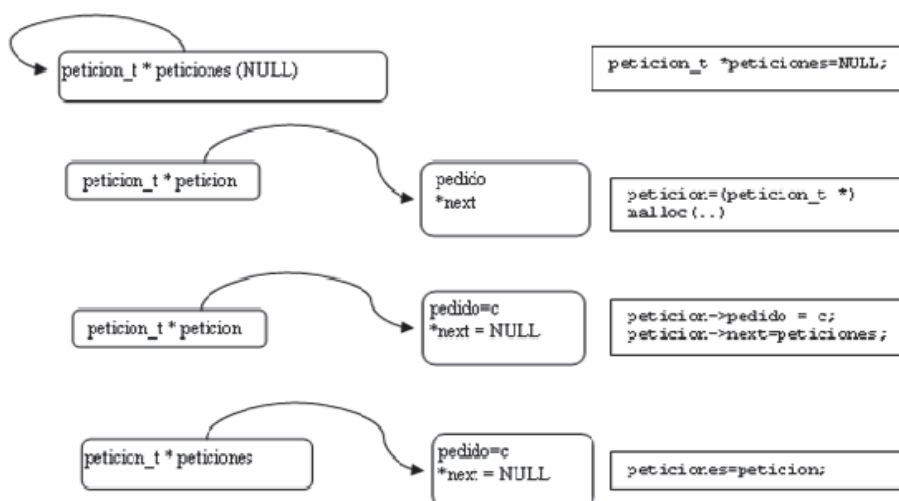
Se pueden establecer los parámetros correspondientes con los valores de los tiempos de espera GET\_DELAY y PROCESS\_DELAY, los cuales simulan dichos tiempos de proceso.

Si  $GET\_DELAY < PROCESS\_DELAY$  entonces se “produce” más rápidamente de lo que se “consume”.

En el siguiente gráfico se muestra una explicación de la inserción de un elemento en la lista encadenada.

Variable global de peticiones (puntero a una estructura de tipo `peticion_t`)

```
// variable global que contendrá los elementos de la lista
encadenada
peticion_t *peticiones = NULL;
añadir petición:
peticion_t *peticion;
peticion = (peticion_t *) malloc(sizeof(peticion_t));
c=contador_pedido++;
// insertamos nuevo elemento en la lista
peticion->pedido = c;
peticion->next = peticiones;
peticiones = peticion;
```



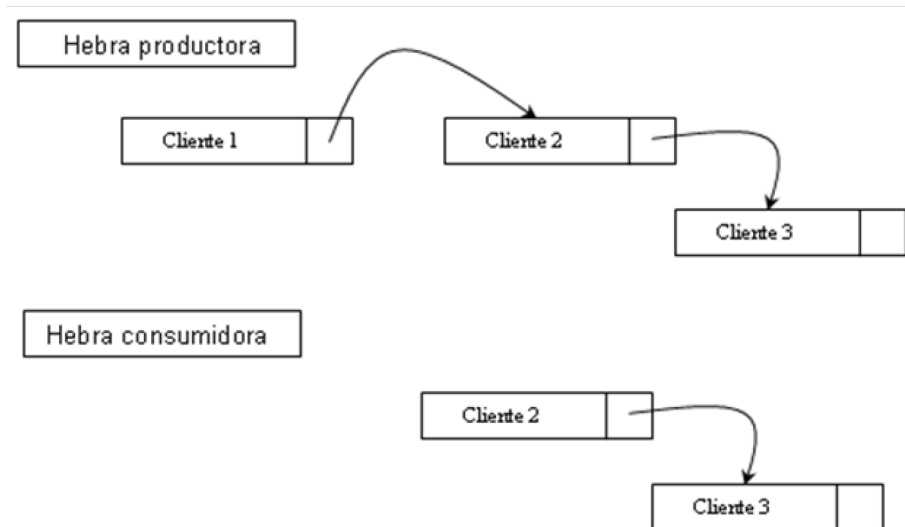


## 9.6. Problemas presupuestos

### 9.6.1. Enunciado 1.

Diseñar un mecanismo de sincronización entre procesos mediante la utilización de semáforos. Se utiliza asimismo el mecanismo mutex para evitar la actualización simultánea de variables compartidas.

Se utiliza como ejemplo el típico problema del proceso productor /consumidor utilizando una lista encadenada. El productor añade elementos a la lista, mientras que el consumidor los elimina. Así pues, tenemos un ejemplo de mecanismo de sincronización entre 2 procesos (hebras) basado en un semáforo.



La hebra consumidora debe esperar a que haya al menos un elemento en la lista, es decir, la lista no puede estar vacía. Esto se consigue mediante la rutina `sem_wait()`, cuyo valor se inicializa a 0 con lo que se bloquea el semáforo. La hebra productora debe añadir nuevos elementos a la lista mientras no esté llena y desbloquear el semáforo mediante la rutina `sem_post()`.

En el programa del ejemplo se puede añadir un número ilimitado de elementos a la lista, pues no se controla si la lista tiene un límite (está llena). La lista se irá llenando si el tiempo de proceso de la hebra productora es inferior al de la hebra consumidora. Se pueden establecer los parámetros correspondientes con los valores de los tiempos de espera `GET_DELAY` y `PROCESS_DELAY`, los cuales simulan dichos tiempos de proceso.

Si `GET_DELAY < PROCESS_DELAY` entonces se “produce” más rápidamente de lo que se “consume”.

La tarea consiste en modificar el programa de ejemplo `ProdConsV1.cpp` de manera que incorpore el control de “lista llena”. Suponemos que la lista admite un máximo de 10 elementos. Considerar la inclusión de un segundo semáforo para realizar el control.

### 9.6.2. Enunciado 2.

Diseñar un programa que muestre en pantalla los números pares y los impares desde el número 1 hasta una constante definida mediante programación. Desde “main” deberá llamarse a una acción que será la que determine si el número es par o impar y lo presente en pantalla. El programa puede plantearse con una única acción (`par_impar`) o con 2 acciones (`par`, `impar`). En caso de utilizar una sola acción, ésta deberá tener un parámetro que indique si debe presentar números pares o impares; en caso de utilizar 2 acciones, este parámetro no será necesario.

### 9.6.3. Enunciado 3.

Modificar el programa del punto anterior para que utilice dos hebras, una para imprimir los valores pares y otra para imprimir los impares. No importa el orden en que se muestren en pantalla.

### 9.6.4. Enunciado 4.

Modificar el programa del punto anterior para que imprima los números en orden ascendente (1, 2, 3, 4, 5, etc.).

Sugerencia: Utilizar 2 semáforos y una variable global (actualización protegida por mutex) para controlar el orden de ejecución de cada hebra.

### 9.6.5. Enunciado 5.

Definir un programa en C++ que determine el instante  $t$  en que la distancia entre las bolas de los 2 péndulos mostrados en la figura 6.3.2 del enunciado 2 del capítulo 6 sea mínima. El programa debe solicitar la entrada de los siguientes parámetros:

- Ángulos iniciales  $\theta_1$ ,  $\theta_2$  validando que dichos valores sean mayores que 0 y menores o iguales a  $\pi/2$ .
- Longitudes  $L_1$  y  $L_2$  de los brazos de cada péndulo. La longitud debe ser menor o igual a 5.
- Masas  $m_1$  y  $m_2$  de cada péndulo



A partir de las ecuaciones (1), (4), (5) y (6) del enunciado 2 del capítulo 6, el programa debe calcular los valores de  $\theta(t)$   $x$ ,  $y$  de cada péndulo en cada intervalo de tiempo, partiendo de  $t=0$  y con incrementos de tiempo  $\Delta t$  de 0,01 segundos. Los valores de  $t$ ,  $x_1$ ,  $y_1$ ,  $x_2$ ,  $y_2$  se deberán almacenar en los campos de una estructura llamada péndulo organizados como una matriz  $M$  de dimensión [1000]. Los 2 péndulos están separados por una distancia  $d$  de 10 metros. El programa finalizará cuando el número de intervalos almacenados sea igual al número de filas de la matriz (1000).

Utilizad las siguientes funciones y acciones:

- void `llena_matriz(pendulo m[M]);` Esta acción pide al usuario que introduzca los valores  $\theta_0$ ,  $L$  y  $m$  (masa) de cada péndulo, valida sus valores y llena la matriz  $m$  con la información de los campos de la estructura péndulo.
- double `calcula_angulo(double t, double theta0, double l);` Esta función calcula el ángulo ( $\theta$ ) en cada instante de tiempo.
- void `calcula_trayectoria(int d, double theta, double l, double& x, double& y);` Esta acción calcula la posición  $x, y$  correspondiente al ángulo  $\theta$  (instante  $t$ ). La variable  $d$  toma el valor 0 para el péndulo 0 y 10 para el péndulo 2.
- void `busca_elemento(pendulo m[M]);` Esta acción busca el elemento de la matriz en que la distancia entre las 2 bolas es mínima y presenta el instante  $t$  y el valor de dicha distancia en pantalla.
- double `distancia(double x1, double y1, double x2, double y2);` función que calcula la distancia entre 2 puntos  $(x1, y1)$   $(x2, y2)$ . Se aplica la siguiente fórmula:  $\text{distancia} = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$ .
- Se debe diseñar una función que controle el movimiento de los dos péndulos sincronizando el movimiento a lo largo del tiempo. Primero debe moverse un péndulo y después el otro. No deben moverse al mismo tiempo.

#### 9.6.6. Enunciado 6.

Definir un programa que permita combinar dos hebras, una que represente el proceso A, que lee datos de un búfer de tamaño fijo y los imprime, y otra que represente el proceso B, que escribe datos en el mismo búfer si existe espacio disponible. Ambas hebras pueden ser bloqueadas. La hebra del proceso A está bloqueada cuando el búfer está vacío y la hebra del B está bloqueada cuando el búfer está lleno. Las hebras deben cooperar entre sí para que sólo estén bloqueadas cuando sea necesario.

Se puede utilizar un semáforo que garantice que el proceso B escribirá sólo cuando haya espacio disponible. No es necesario utilizar variables globales. El cero puede indicar el estado de bloqueo y otro disponible.

### 9.6.7. Enunciado 7.

Definir un programa que permita simular el proceso de composición de una molécula de dióxido de carbono ( $\text{CO}_2$ ). Esta molécula es resultado de la unión de tres átomos, dos de oxígeno y uno de carbono, en la que por cada átomo de carbono se toman dos de oxígeno. Cada átomo se representará por una hebra y se deben asociar las tres hebras para formar el dióxido de carbono. Se usarán dos semáforos: uno deberá controlar las hebras de oxígeno y otro las hebras de carbono. En la Figura 9.6.1 se muestra el funcionamiento de los semáforos para construir una molécula de  $\text{CO}_2$ . Generalizar el programa para producir  $n$  moléculas.

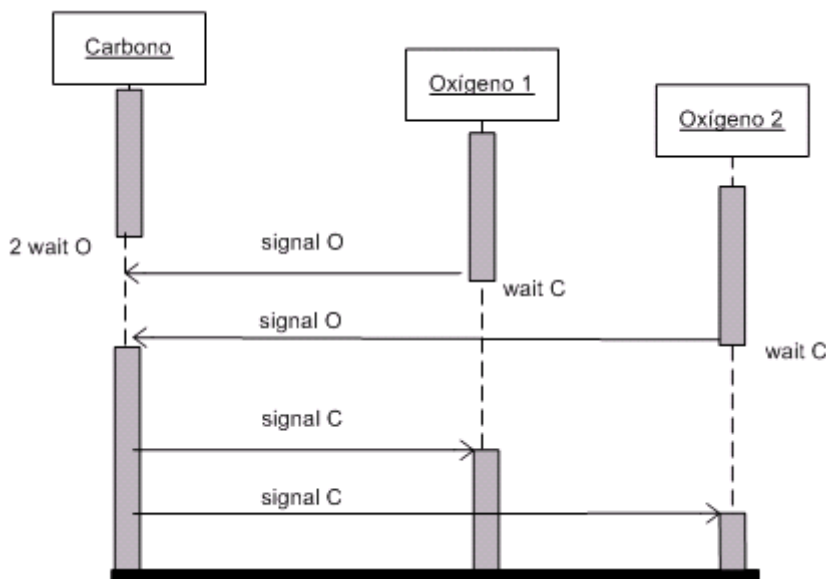


Fig 9.6.1.  
Secuencia de  
creación de una  
molécula de  $\text{CO}_2$

### 9.6.8. Enunciado 8. Programación concurrente (hebras): El juego del frontón

Este enunciado es una ampliación del caso de estudio del capítulo de eventos y se trata de modificarlo con las características de la programación concurrente.

Recordemos que cada ventana tiene una función asociada, que esta función se conoce como **procedimiento de ventana**, y que es la encargada de procesar adecuadamente todos los mensajes enviados a una determinada clase de ventana, es decir, la responsable de todo lo relativo al aspecto y al comportamiento de una ventana.

Normalmente estas funciones están basadas en una estructura "switch" donde cada "case" corresponde a un determinado tipo de mensaje o evento.



En la versión inicial del programa, utilizamos el evento WM\_TIMER para simular el movimiento de las bolas y de la pala. El evento WM\_TIMER se activa mediante la siguiente instrucción: SetTimer( hwnd, 1, Timeout, NULL ) incluida en la lógica del evento WM\_RBUTTONDOWN. Una vez activado el temporizador, se ejecuta el evento WM\_TIMER cada Timeout milisegundos siendo Timeout una variable tipo integer que contiene el valor de “refresco” del temporizador.

La lógica del evento es la siguiente:

```
case WM_TIMER: // movemos la pelota según Timeout
    //Empieza el movimiento con el botón derecho del ratón
    if (bMover) {
        // i=1 la pelota toca la pala , i=0 la pelota no toca
        la pala
        i=mTocaPala (X, Y, XP, YP);
        // j=0 --> No toca el borde, j=1 --> Superior, j=2 --
        > Inferior,
        // j=3 --> Izquierdo, j=4 --> Derecho
        j=mTocaBorde (X, Y);
        // calculamos tipo de movimiento:
        // 0 --> nuevo ángulo. Toca el borde,
        // 1 --> nuevas coordenadas x, y. Mismo ángulo
        // 2 --> nuevo ángulo. Toca la pala
        if (i==1) tipo=2; // toca la pala, nuevo ángulo
        else {
            if (j>0) tipo=0; // toca el borde, nuevo ángulo
            else tipo=1; // no se modifica el ángulo, nuevas
            coordenadas
        }
        if (j==4) { //Si tocamos el borde derecho, acabamos
        el juego
            bMover=FALSE;
            nPelotas=0;
        }
        else { // Si no calculamos nuevo ángulo/posición
            mMoverPelota (tipo, X, Y, V, Alpha);
        }
        // Para forzar el repintado de la ventana (mensaje
        WM_PAINT)
        // Invalidamos únicamente la zona limitada por la
        pelota
        // para que no se note tanto el parpadeo.
        lRect.left=(int) (X-2*RADIO);
        lRect.top=(int) (Y-2*RADIO);
        lRect.right=(int) (X+2*RADIO);
        lRect.bottom=(int) (Y+2*RADIO);
        lpRect=&lRect;
        //Cuando se acaba el juego (la pelota toca el borde
        derecho),
        //invalidamos TODA la ventana.
```

```

    if (bMover) InvalidateRect(hwnd, lpRect, TRUE);
    else InvalidateRect(hwnd, NULL, TRUE);
}
break;

```

El objetivo es sustituir el control actual del movimiento de las bolas y la pala por una rutina “controller”. Dicha rutina se ejecutará mediante una hebra independiente de la hebra principal del programa (WinMain) y sustituirá el control de movimiento actualmente efectuado por el evento WM\_TIMER. Adicionalmente, se extenderá el juego para que puedan mantenerse 2 hebras de ejecución excluyente, cada una con su propio conjunto de posiciones y grado de dificultad.

Dado que las variables que utiliza el evento WM\_TIMER para comunicarse con el resto de funciones del procedimiento de ventana (WindowProcedure) son locales con respecto a dicho procedimiento, y dado que la nueva rutina de control debe ser independiente del procedimiento de ventana, deberán convertirse dichas variables en globales para establecer la comunicación.

Asimismo, para poder generalizar el tratamiento del movimiento a más de una hebra, deberá construirse una estructura (struct) con las variables de entrada necesarias para la rutina “controller”. El pase de parámetros debe realizarse a través de una estructura ya que la rutina pthread\_create solamente admite el pase por referencia de un parámetro. Puede tomarse como modelo la siguiente estructura:

```

struct pthread_st {
    HWND hwnd; // manejador de pantalla
    double X; // coordenada x bola
    double Y; // coordenada y bola
    int XP; // coordenada x pala
    int YP; // coordenada y pala
    double V; // velocidad
    double Alpha; // ángulo trayectoria
    BOOL bMover; // para indicar si la bola está en
    movimiento o parada (barra espaciadora)
    int hebra; // identifica la hebra
};

```

Dividimos el ejercicio en 2 partes:

1) Sustituir el control del movimiento actual por la rutina “controller”, que se ejecutará como una hebra independiente de la hebra principal. Establecer como bucle de “refresco” un tiempo de 10 milisegundos. La activación del juego se realiza dentro del evento WM\_RBUTTONDOWN (botón derecho del ratón), por lo que la creación de las hebras deberá incluirse dentro de este evento.



II) Añadir una segunda hebra a la rutina "controller". La ejecución de las 2 hebras será excluyente mediante la utilización de 2 semáforos. Cada hebra debe mantener su propio conjunto de posiciones de bolas y pala, así como el grado de dificultad (velocidad de la bola) controlada mediante las teclas "d" y "f". Para resaltar de manera visual cada una de las 2 hebras, las bolas y la pala se pintarán de colores diferentes para cada una de ellas. Mediante la tecla "ctrl." (evento WM\_KEYDOWN valor case 17), se podrá alternar la ejecución de una hebra u otra.

Al activar una hebra después de haber pulsado la tecla Ctrl y antes de colorear los nuevos elementos, deberá "limpiarse" la ventana de los elementos gráficos utilizados por la hebra anterior. Para ello deberá utilizarse la siguiente secuencia de instrucciones:

```
bMover=FALSE;  
InvalidateRect(hwnd, NULL, TRUE);  
bMover=TRUE;
```

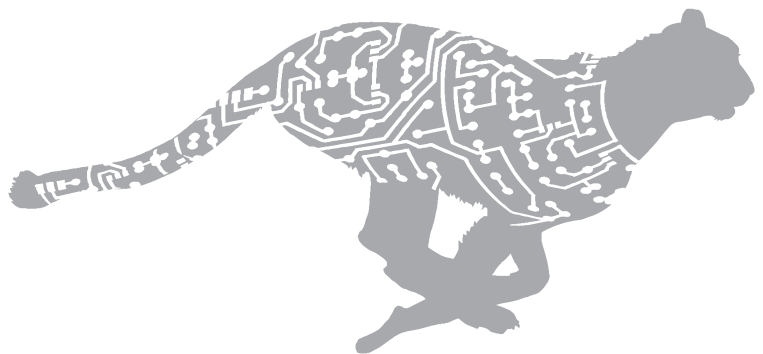
### Reglas del juego

Si la bola toca el lado derecho del cuadrado se termina el juego. La pala rectangular permite al usuario evitar que eso ocurra.

Si la bola toca cualquiera de los otros lados o la pala, rebota hacia el interior del cuadrado.

### Control del juego

- Con las teclas "d" (difícil), y "f" (fácil) se regula la velocidad de la pelota.
- Con las teclas de flecha hacia arriba o hacia abajo se controla el movimiento de la pala a lo largo del lado derecho del cuadrado, dentro de la superficie de la ventana.
- Con el botón izquierdo del ratón se coloca la pelota en la posición inicial (dentro del terreno) y la pala cerca del lado derecho. Con el botón derecho del ratón se inicializa el movimiento.
- Con la barra espaciadora se para o reanuda el movimiento.
- Con la tecla Ctrl se alterna la ejecución de las hebras.



→ 10



# Bases de datos

## 10.1. Conceptos básicos

Una base de datos o banco de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso [19]. La mayoría de las bases de datos están en formato digital (electrónico), ya que éste ofrece un amplio rango de soluciones al problema de almacenar datos. Existen programas denominados “*sistemas gestores de bases de datos*”, abreviados SGBD, que permiten almacenar los datos de manera que se pueda acceder a ellos posteriormente de forma rápida y estructurada. Las aplicaciones más usuales son para la gestión de empresas e instituciones públicas. También son ampliamente utilizadas en entornos científicos con el objeto de almacenar la información experimental. Aunque las bases de datos pueden contener muchos tipos de datos, algunos de ellos se encuentran protegidos por las leyes de varios países. En España, por ejemplo, los datos personales se encuentran protegidos por la Ley Orgánica de Protección de Datos de Carácter Personal (LOPD). Las aplicaciones para acceder a bases de datos deben ser robustas (no fallar con facilidad), portables (que se puedan trasladar a otros entornos y plataformas) y fáciles de mantener.

### 10.1.1. Las características de las bases de datos

Las operaciones fundamentales que tiene que brindar una base de datos son las siguientes: Crear (**Create**), Leer (**Read**) Actualizar (**Update**) y Eliminar (**Delete**) de forma rápida. Además, la base de datos deberá permitir buscar información que sea consistente y válida. La corrección de datos deberá ser fácil. Para mantener la consistencia y la validez se necesitan transacciones atómicas, es decir, una serie compleja de acciones que se



considere como una única operación: la transacción se producirá de forma completa o no se producirá en ninguna de sus partes. Un sistema efectivo de transacciones debe proporcionar atomicidad, consistencia, aislamiento y durabilidad (ACID-*Atomicity*, *Consistency*, *Isolation*, *Durability*). La consistencia se refiere a que el estado de la base de datos sea consistente antes y después de la transacción. El aislamiento se refiere a que sólo tenga acceso a la transacción la persona que la realiza. La durabilidad se refiere a que si la transacción se ha confirmado no desaparezca posteriormente. Los datos de una base de datos deben ser persistentes, es decir, no cambian ni desaparecen por sí mismos. Además, las bases de datos deben ser extensibles y portables.

### 10.1.2. Tipos de bases de datos

- **Bases de datos estáticas:** Éstas son bases de datos de sólo lectura, utilizadas primordialmente para almacenar datos históricos que posteriormente se pueden utilizar para estudiar el comportamiento de un conjunto de datos a través del tiempo, realizar proyecciones y tomar decisiones.
- **Bases de datos dinámicas:** En ellas la información almacenada se modifica con el tiempo, permitiendo operaciones como la actualización, el borrado y la adición de datos, además de las operaciones fundamentales de consulta.

### 10.1.3. Modelos de bases de datos

Un modelo de base de datos es básicamente una “descripción” del *contenedor de datos* (el lugar en el que se guarda la información), así como de los métodos para almacenar y recuperar información de esos contenedores.

- **Bases de datos jerárquicas:** Como su nombre indica, almacenan su información en una estructura jerárquica. En este modelo los datos se organizan de una forma similar a la de un árbol visto al revés, en donde un *nodo padre* de información puede tener varios *hijos*. El nodo que no tiene padres es llamado *raíz*, y a los nodos que no tienen hijos se los conoce como *hojas*. Las bases de datos jerárquicas son especialmente útiles en el caso de aplicaciones que manejan un gran volumen de información y datos muy compartidos, puesto que permiten crear estructuras estables y de gran rendimiento. Una de las principales limitaciones de este modelo es su incapacidad para representar eficientemente la redundancia de datos.

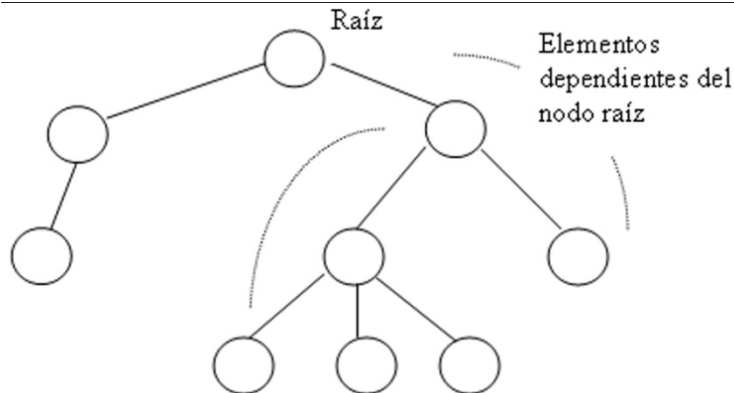


Fig 10.1.1  
Base de datos  
jerárquica

- **Bases de datos relacionales:** Éste es el modelo utilizado en la actualidad para modelar problemas reales y administrar datos dinámicamente. Tras ser postulados sus fundamentos en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José (California), no tardó en consolidarse como un nuevo paradigma en los modelos de bases de datos. Su idea fundamental es el uso de “relaciones”. Estas relaciones podrían considerarse en forma lógica como conjuntos de datos llamados “tuplas”. Pese a que ésta es la teoría de las bases de datos relacionales creadas por Codd, normalmente se conceptualiza de una manera más fácil de imaginar, que consiste en pensar en cada relación como si fuese una tabla que está compuesta por registros que representarían las tuplas (las filas de la tabla), y campos (las columnas de la tabla). En este modelo, el lugar y la forma en que se almacenen los datos no tienen relevancia (a diferencia de otros modelos como el jerárquico y el de red). Esto tiene la considerable ventaja de que es más fácil de entender y de utilizar para un usuario esporádico de la base de datos. La información puede ser recuperada o almacenada mediante “consultas” que ofrecen una amplia flexibilidad y poder para administrarla. El lenguaje más habitual para construir las consultas a bases de datos relacionales es el SQL (*Structured Query Language*) o *Lenguaje estructurado de consultas*, un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales. El término “claves de tabla” se utiliza para acceder a determinados registros de la tabla, mientras que un “índice” es un mecanismo que mejora el rendimiento de una base de datos. Las claves pueden ser primarias o secundarias. La clave primaria es una clave candidata que incluye un campo o una combinación de campos que identifican un registro de manera exclusiva: su valor debe ser único y no nulo. El resto de claves se denominan claves alternativas o secundarias. La relación entre dos tablas se crea denominando un campo común a dos tablas. Este campo común debe ser la clave



primaria de una de las tablas. Las claves externas permiten garantizar lo que se conoce como integridad referencial.

- **Bases de datos multidimensionales:** Son bases de datos para desarrollar aplicaciones muy concretas, como las que se utilizan para la creación de Cubos OLAP (*On-Line Analytical Processing*). Se utilizan en el campo de la Inteligencia Empresarial (o *Business Intelligence*), y su objetivo es agilizar la consulta de grandes cantidades de datos. Se construyen estructuras multidimensionales (o *Cubos OLAP*) que contienen datos resumidos de Sistemas Transaccionales (*OnLine Transaction Processing*). Se usan en informes de negocios de ventas, marketing, informes de dirección, minería de datos (extracción del conocimiento explícito de datos) y áreas similares. En la minería de datos se aborda la solución a problemas de predicción, clasificación y segmentación.

Básicamente no se diferencian demasiado de las bases de datos relacionales (una tabla en una base de datos relacional podría serlo también en una base de datos multidimensional). Se trata, más bien, de una diferencia conceptual. En las bases de datos multidimensionales los campos o atributos de una tabla pueden ser de dos tipos: o bien representan las dimensiones de la tabla, o bien representan métricas que se desean estudiar.

- **Bases de datos orientadas a objetos:** Almacenan en la base de datos los objetos completos (**estado y comportamiento**). Una base de datos orientada a objetos es una base de datos que incorpora todos los conceptos importantes del paradigma de objetos:
  - o *Encapsulación:* Propiedad que permite ocultar la información al resto de los objetos, impidiendo así accesos incorrectos o conflictos.
  - o *Herencia:* Propiedad a través de la cual los objetos heredan un comportamiento dentro de una jerarquía de clases.
  - o *Polimorfismo:* Propiedad de una operación mediante la cual puede ser aplicada a distintos tipos de objetos.

En bases de datos orientadas a objetos, los usuarios pueden definir operaciones sobre los datos como parte de la definición de la base de datos.

- **Bases de datos documentales:** Permiten la indexación a texto completo y, en líneas generales, realizar búsquedas más potentes. Tesauro es un sistema de índices optimizado para este tipo de bases de datos.
- **Bases de datos deductivas:** Son sistemas de bases de datos pero con la diferencia de que permiten hacer deducciones a través de inferencias. Se basan principalmente en reglas y hechos que son almacenados en las bases de datos. Las bases de datos deductivas

son también llamadas bases de datos lógicas, puesto que se basan en la lógica matemática.

- **Bases de datos transaccionales:** Son bases de datos cuyo único fin es el envío y recepción de datos a grandes velocidades, por lo que la redundancia y duplicación de información no constituyen un problema, como ocurre con las demás bases de datos. Para poder aprovecharlas al máximo permiten algún tipo de conectividad a bases de datos relacionales.
- **Bases de datos distribuidas:** (SGBDD) La base de datos y el software SGBD pueden estar distribuidos en múltiples sitios conectados por una red. Hay de dos tipos:
  - Distribuidos homogéneamente: Utilizan el mismo SGBD en múltiples sitios.
  - Distribuidos heterogéneamente: Dan lugar a los SGBD federados o sistemas multibase de datos en los que los SGBD participantes tienen cierto grado de autonomía local y tienen acceso a varias bases de datos autónomas. Muchos de estos emplean una arquitectura cliente-servidor.

## 10.2.Diseño de bases de datos

El diseño de bases de datos se lleva a cabo en tres etapas: diseño conceptual, diseño lógico y diseño físico.

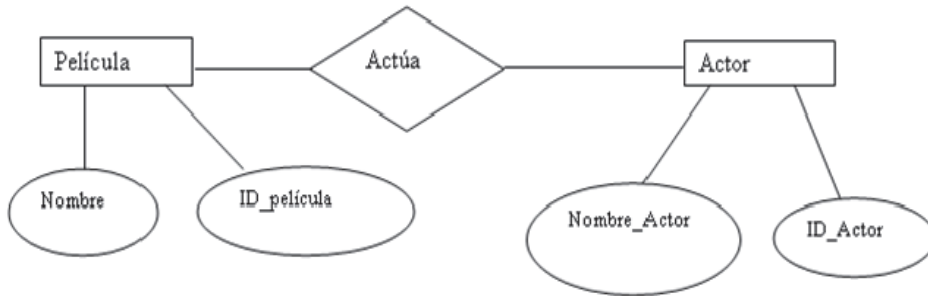
**Diseño conceptual:** Se construye el esquema de la información del proceso objeto del estudio de manera independiente del diseño físico. Este esquema, que suele denominarse esquema conceptual, representa la relación entre los datos y su significado (semántica). Es decir, la lógica de los datos. Durante el diseño se construyen las entidades, que son unos objetos que pueden tener múltiples propiedades (atributos) y que servirán para definir el modelo de datos en la estructura de la base de datos. El esquema conceptual se construye utilizando la información que se encuentra en la especificación de los requisitos de usuario. Cada registro que forma una entidad se denomina "ocurrencia" o "ejemplar de la entidad".

En el modelo de entidad-relación se representan de forma gráfica las relaciones entre las entidades y sus atributos. Para representar las entidades se usa un rectángulo. Existen dos tipos de entidades: entidades fuertes ( regulares) y entidades débiles. Las entidades fuertes tienen atributos clave y pueden existir de forma independiente, mientras que las entidades débiles no tienen atributos clave propios y su existencia depende de otras entidades. Las relaciones entre dos entidades se representan mediante un rombo; los nombres de las entidades se ponen dentro de un



rectángulo y los de los campos dentro de un óvalo. En la Figura 10.3.1 describimos un ejemplo.

Fig 10.2.1  
Esquema del modelo  
de entidad-relación.



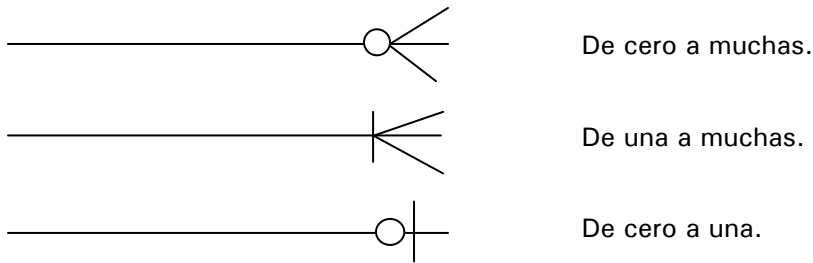
El objetivo del diseño conceptual es comprender:

- La perspectiva que cada usuario tiene de los datos.
- La naturaleza de los datos, independientemente de su representación física.
- El uso de los datos a través de las áreas de aplicación.

Al estudiar un proceso del mundo real (empresa), se deben especificar primero los requerimientos para su automatización. Estos requerimientos son el punto de partida para definir el esquema conceptual. La notación del modelo entidad-relación es la más conocida para representar el diseño conceptual. Durante todo el proceso de desarrollo del esquema conceptual éste se prueba y se valida con los requisitos de los usuarios. El esquema conceptual es una fuente de información para el diseño lógico de la base de datos.

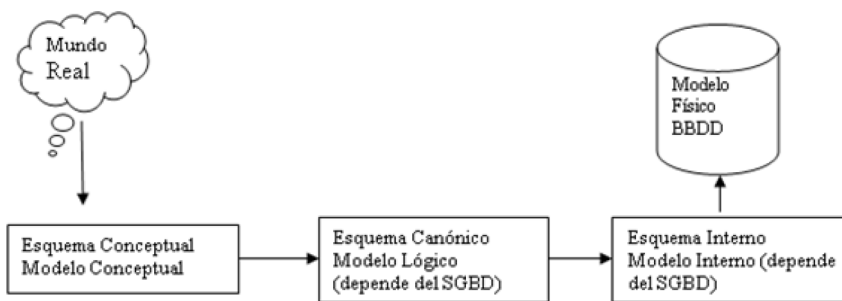
Para definir el modelo entidad-relación debe considerarse la cardinalidad que indica el número de relaciones en las que una entidad puede aparecer. La cardinalidad mínima muestra el número mínimo de asociaciones en las que puede estar cada entidad (de cero a una). La cardinalidad máxima indica el número máximo de relaciones en las que puede estar cada entidad (de una a muchas).

En los esquemas entidad/relación la cardinalidad se puede indicar con los siguientes gráficos:



En las líneas de la relación se pueden indicar roles. Los roles representan la función de una entidad en una determinada relación. Describen propiedades de las entidades y sus relaciones. En este modelo se representan con un círculo, dentro del cual se coloca el nombre del atributo.

Ejemplo:



Los identificadores son atributos que de forma unívoca seleccionan un registro.

Los pasos para diseñar una base de datos son los siguientes:

1. Encontrar entidades (conjuntos de entidades).
2. Identificar atributos de las entidades.
3. Buscar identificadores.
4. Especificar las relaciones y cardinalidades.
5. Identificar entidades débiles.
6. Especializar y generalizar entidades donde sea posible.

### 10.3.MySQL: un gestor de base de datos

Para desarrollar los proyectos y ejercicios propuestos en este capítulo utilizaremos el gestor de base de datos MySQL. Los motivos que justifican dicha elección son los siguientes:

- Al ser un producto OpenSource no es necesaria la adquisición de licencias para su instalación.
- Se trata de un producto con reconocido prestigio, fiabilidad, velocidad, rendimiento, facilidad de administración y conexión con otros productos (por ejemplo con POSIX), bien documentado, y con una buena evolución y un buen soporte.
- El gestor de BDD MySQL está siendo utilizado en muchos entornos productivos, por lo que ofrece confianza.
- Los gestores de bases de datos OpenSource hace tiempo que son una alternativa real para las empresas. Son productos cada vez mas evolucionados y con mas funcionalidades, y las empresas que los desarrollan tienen también cada vez más volumen de negocio.
- MySQL dispone de una herramienta de diseño de bases de datos, como es MySQL Workbench, que permite definir modelos relacionales, generar las sentencias SQL y construir la base de datos, así como realizar ingeniería inversa (construir el modelo de datos a partir de una base de datos existente).

## 10.4.Instalación de MySQL

La descarga de los programas se realiza desde [www.mysql.com](http://www.mysql.com). Mediante el programa de instalación Windows, se crea la instancia como servicio Windows.

El entorno se puede configurar mediante el programa WinMySQLAdmin. La instalación crea el usuario *root* con privilegios de administrador. A partir de aquí se puede ir configurando la base de datos.

En la tabla 10.4.1 se resumen el contenido del directorio de instalación y sus subdirectorios.

| Directorio | Contenido                                    |
|------------|----------------------------------------------|
| bin        | Programas cliente y el servidor MySQL.       |
| data       | Ficheros de registro (logs), bases de datos. |
| Docs       | Documentación.                               |
| examples   | Programas y scripts de ejemplo.              |
| include    | Ficheros de inclusión.                       |
| lib        | Bibliotecas.                                 |
| scripts    | Scripts de utilidades.                       |
| share      | Ficheros con mensajes de error.              |

## 10.5. Ejemplos sentencias MySQL

Definimos 3 tipos diferenciados de sentencias: de creación de datos (DDS), de manipulación de datos (DMS) y de administración de la BDD.

**Sentencias de definición DDS** (*Data Definition Statements*): Son aquellas que permiten la creación y definición de objetos de la base de datos (desde la creación de la propia base de datos hasta la creación de tablas, índices, procedimientos almacenados, etc). A continuación presentamos unos ejemplos sencillos de las más habituales:

| Comando/sentencia                                                                                                                 | Descripción                                                                                                                                                                                                                                                                                                                                                                                                                                               | Ejemplo de uso                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CREATE {DATABASE   SCHEMA} [IF NOT EXISTS] nombre_db<br><br>[especificacion [, especificacion] ...]                               | <p>Crea una base de datos con el nombre "nombre_db". Se necesita tener permiso de CREATE en la BDD.</p> <p>Las bases de datos en MySQL se implementan como directorios que contienen ficheros que se corresponden a tablas en la base de datos.</p> <p>Las opciones "especificación" pueden darse para especificar características de la base de datos. Estas características se almacenan en el fichero db.opt en el directorio de la base de datos.</p> | CREATE DATABASE DB_PRUEBA;                                                                                                                                          |
| CREATE [TEMPORARY] TABLE [IF NOT EXISTS] nombre_tabla<br><br>[(definicion_tabla,...)]<br><br>[opcion_tabla]<br>[sentencia_select] | <p>CREATE TABLE crea una tabla con el nombre dado. Debe tener el permiso CREATE para la tabla.</p> <p>Se puede usar la palabra TEMPORARY al crear una tabla. En este caso es visible sólo para la conexión actual, y se borra automáticamente cuando la conexión se cierra. Las cláusulas de "definicion_tabla" permiten especificar las características de las columnas (campos) de la tabla.</p>                                                        | CREATE TABLE TB_CLIENTE (codigo INTEGER(12), nombre CHAR(20), apellido1 CHAR(40), apellido2 CHAR(40), fecha_nacimiento DATE);                                       |
| CREATE [UNIQUE FULLTEXT SPATIAL] INDEX nombre_indice<br><br>[USING tipo_indice]<br><br>ON nombre_tabla (nombre_columna,...)       | <p>Permite crear un índice asociado a una tabla. Aunque la sentencia CREATE TABLE permite crear índices en el momento de creación de la tabla (cláusula "definición_tabla"), también se pueden crear los índices con posterioridad a la creación de la tabla mediante la sentencia CREATE INDEX.</p>                                                                                                                                                      | <p>CREATE UNIQUE INDEX IDX_CLIENTE_01 ON TB_CLIENTE (codigo INTEGER(12));</p> <p>CREATE INDEX IDX_CLIENTE_02 ON TB_CLIENTE (apellido1(40), apellido2 CHAR(40));</p> |



| Comando/sentencia                                                | Descripción                                                                                                                                                                         | Ejemplo de uso                                                                                                                           |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| CREATE PROCEDURE<br>nombre_procedimiento<br>([parametro1[,...]]) | Permite la creación de procedimientos almacenados. Son trozos de código escritos en lenguaje propio de BDD, normalmente utilizados para centralizar y agilizar los accesos a datos. | CREATE PROCEDURE<br>PRC_CUENTA (OUT<br>param1 INT)<br><br>BEGIN<br><br>SELECT<br>COUNT(*) INTO<br>param1 FROM<br>TB_CLIENTE;<br><br>END; |

**Sentencias de manipulación DMS (*Data Manipulation Statements*):** Son aquellas que permiten la inserción, actualización y recuperación de los datos en una BDD. En la tabla siguiente presentamos una selección de las más significativas con ejemplos sencillos de uso y una breve descripción.

| Comando/sentencia                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Descripción                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Ejemplo de uso                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| SELECT<br><br>[ALL   DISTINCT  <br>DISTINCTROW ]<br><br>[HIGH_PRIORITY]<br><br>[SQL_CACHE  <br>SQL_NO_CACHE]<br>[SQL_CALC_FOUND_ROWS]<br><br>select_expr, ...<br><br>[INTO OUTFILE<br>'nombre_fichero'<br>opciones_exportacion<br><br>  INTO DUMPFILE<br>'nombre_fichero']<br><br>[FROM clausula_from<br><br>[WHERE clausula_where]<br><br>[GROUP BY {nombre_columna<br>  expresion   posicion}<br><br>[ASC   DESC], ... [WITH<br>ROLLUP]<br><br>[HAVING clausula_where]<br><br>[ORDER BY {nombre_columna<br>  expresion   posicion}<br><br>[ASC   DESC] , ...]<br><br>[LIMIT {[offset,] numero_filas<br>  numero_filas OFFSET<br>offset}] | No nos equivocamos mucho si decimos que probablemente es la sentencia más utilizada. No debemos olvidar que el objetivo último de un gestor de base de datos es almacenar información para su posterior recuperación. La sentencia SELECT permite recuperar información previamente almacenada en la BDD. Aunque la velocidad de recuperación depende en gran medida del diseño de las tablas (índices asociados), la sentencia SELECT utiliza una serie de cláusulas, principalmente la "clausula_where" y la "clausula_from", que determinan el resultado de la recuperación.<br><br>La cláusula GROUP BY permite agrupar registros que tienen el mismo valor en las columnas en ella indicadas. | SELECT nombre,<br>apellido1, apellido2<br>FROM TB_CLIENTE<br>WHERE codigo = 123;<br><br><br>SELECT COUNT(*)<br>FROM TB_CLIENTE<br>WHERE nombre<br>='POL'; |

| Comando/sentencia                                                                                                                                                                                                                                      | Descripción                                                                                                                                                                                                                                                                                                                                                                                             | Ejemplo de uso                                                                                                                                                                                                                                                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [PROCEDURE<br>nombre_procedimiento(lista_argumentos)]                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                              |
| Opción 1:<br><br>INSERT [LOW_PRIORITY   DELAYED   HIGH_PRIORITY] [IGNORE]<br><br>[INTO] nombre_tabla<br>[(nombre_columna,...)]<br><br>VALUES ({expr   DEFAULT},...),(...),...<br><br>[ ON DUPLICATE KEY UPDATE<br>nombre_columna = expresion,<br>... ] | Permite insertar registros en una tabla. La cláusula "ON DUPLICATE KEY" permite modificar el contenido de algún campo de un índice único antes de la inserción para evitar la clave duplicada.<br><br>En la opción 1, los valores que se pretenda insertar se indican directamente en la misma sentencia; en la opción 2, se obtienen del resultado de la selección especificada en la cláusula SELECT. | INSERT INTO<br>TB_CLIENTE (codigo, nombre, apellido1, apellido2) VALUES<br>(124, 'VICTOR', 'PORTAL', 'PEREZ');<br><br>INSERT INTO<br>TB_CLIENTE_HIST<br>(codigo, apellido1, apellido2, fecha_nacimiento)<br><br>SELECT (codigo, apellido1, apellido2, fecha_nacimiento)<br>FROM TB_CLIENTE<br>WHERE<br>apellido1 = 'GARCIA'; |
| Opción 2:<br><br>INSERT [LOW_PRIORITY   HIGH_PRIORITY] [IGNORE]<br><br>[INTO] nombre_tabla<br>[(nombre_columna,...)]<br><br>SELECT ...<br><br>[ ON DUPLICATE KEY UPDATE<br>nombre_columna = expresion,<br>... ]                                        |                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                              |
| DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM<br>nombre_tabla<br><br>[WHERE<br>clausula_where]<br><br>[ORDER BY ...]<br><br>[LIMIT numro_filas]                                                                                                          | La sentencia DELETE se utiliza para eliminar registros de una tabla. Si se quieren eliminar todos los registros, se puede utilizar la sentencia TRUNCATE que es más eficiente. En caso de querer eliminar únicamente determinados registros, se debe utilizar la sentencia DELETE, indicando en la cláusula WHERE los registros que hay que eliminar.                                                   | DELETE FROM<br>TB_CLIENTE WHERE<br>apellido1 = 'GARCIA';                                                                                                                                                                                                                                                                     |
| TRUNCATE TABLE<br>nombre_tabla                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                              |
| UPDATE [LOW_PRIORITY] [IGNORE] nombre_tabla<br><br>SET columna1 = expr1 [, columna2 = expr2 ...]<br><br>[WHERE clausula_where]<br><br>[ORDER BY ...]                                                                                                   | La sentencia UPDATE permite modificar los valores de los campos de una tabla. Los registros por modificar están determinados por la clausula WHERE y los valores que tomarán los campos de la tabla después de la actualización están determinados                                                                                                                                                      | UPDATE TB_CLIENTE<br>SET<br>apellido1 = 'LOPEZ'<br>where codigo = 124;                                                                                                                                                                                                                                                       |



| Comando/sentencia                                                                                                                                                                                        | Descripción                                                                                                                                                                                                                                            | Ejemplo de uso                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| [LIMIT numero_filas]                                                                                                                                                                                     | por las expresiones expr1, expr2, etc.                                                                                                                                                                                                                 |                                                           |
| USE nombre_bdd                                                                                                                                                                                           | El comando USE db_name le dice a MySQL que use la base de datos db_name como la base de datos por defecto para los comandos siguientes. Sigue siendo la base de datos por defecto hasta el final de la sesión o hasta que se realiza otro comando USE. | USE DB_PRUEBA;                                            |
| SHOW [FULL] TABLES [FROM nombre_bdd] [LIKE 'modelo']<br>SHOW [FULL] COLUMNS FROM nombre_tabla [FROM nombre_bdd] [LIKE 'modelo']<br>SHOW {DATABASES   SCHEMAS} [LIKE 'modelo']<br>SHOW GRANTS FOR usuario | Muestra características u objetos de la base de datos o de la instalacion MySQL.                                                                                                                                                                       | Muestra las tablas de la BDD por defecto:<br>SHOW TABLES; |

### Sentencias de administración de la BDD: Presentamos las más básicas:

| Comando/sentencia                                                                                                                                                                                                                                                                    | Descripción                                                                                                                                                                                                                                       | Ejemplo de uso                                                                                                                                                                                                                                                                             |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CREATE USER usuario<br>[IDENTIFIED BY<br>[PASSWORD] 'password']<br><br>[, usuario [IDENTIFIED BY<br>[PASSWORD] 'password']]<br>...                                                                                                                                                   | Permite crear nuevas cuentas (usuarios) para que puedan acceder a la BDD. Se puede asignar una contraseña utilizando la cláusula IDENTIFIED BY.                                                                                                   | CREATE USER usuario1<br>IDENTIFIED BY pwd001;                                                                                                                                                                                                                                              |
| GRANT tipo_privilegio<br>[(columnas)] [, tipo_privilegio<br>[(columnas)]] ...<br><br>ON [tipo_objeto]<br>{nombre_tabla   *   *.*  <br>nombre_bdd.*}<br><br>TO usuario [IDENTIFIED<br>BY [PASSWORD]<br>'password']<br><br>[, usuario [IDENTIFIED<br>BY [PASSWORD]<br>'password']] ... | Se utiliza la sentencia GRANT para dar acceso a los usuarios indicados en la cláusula TO a los objetos indicados en la clausula ON. Se pueden indicar privilegios generales a nivel global de MySQL, a nivel de base de datos o a nivel de tabla. | Privilegios generales a nivel global de mySQL:<br><br>GRANT ALL ON *.* TO usuario1;<br><br>Privilegios generales a nivel global de BDD:<br>GRANT ALL ON DB_PRUEBA.* TO usuario1;<br><br>Privilegios generales a nivel global de tabla:<br>GRANT ALL ON DB_PRUEBA.TB_CLIENTE.* TO usuario1; |

## 10.6.MySQL en C++

Una vez creada una BDD con MySQL, podemos acceder a ella mediante sentencias desde el propio entorno de comandos de MySQL o bien desde programas. En esta sección veremos cómo configurar el compilador Dev C++ para poder trabajar con MySQL desde el programa, así como algunos ejemplos de conexión y acceso a objetos de la base de datos.

El compilador necesita la librería estática `libmysql.a` que debe existir en el directorio `\lib` y debe referenciarse mediante la inclusión de la entrada `-lmysql` en las opciones para añadir comandos en la línea de comandos del linker (ver figura 10.6.1). Adicionalmente, en el momento de la ejecución, debe estar disponible la librería dinámica `libmysql.dll` (puede copiarse por ejemplo en el directorio donde se ejecuta el programa). También deben copiarse bajo el directorio `/includes` del Dev-C++ las cabeceras `mysql/mysql.h` y `mysql/mysql.h`, que incluyen la definición de tipos propios.

Una vez realizadas las acciones anteriores, y suponiendo que tenemos la base de datos arrancada, lo primero que tenemos que hacer para poder acceder a los objetos de la BDD es establecer una conexión con MySQL. Para ello utilizamos las sentencias incluidas en la función `dbconnect` descrita a continuación:

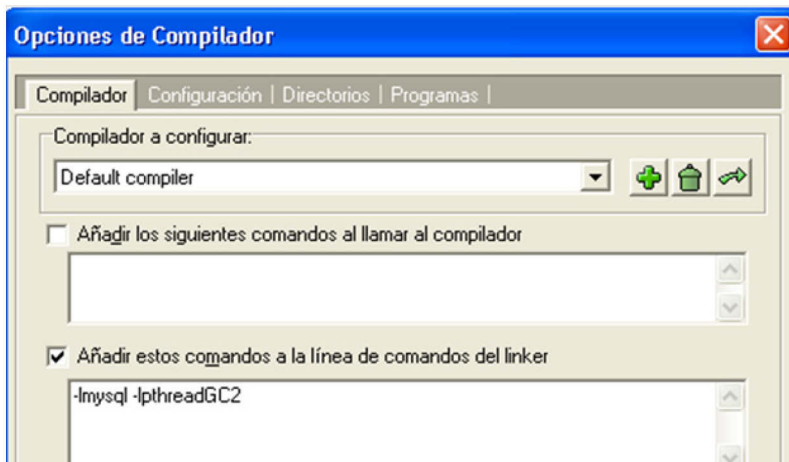


Fig 10.6.1

```
int dbconnect (void)
// abre conexion con MySQL
{
    MYSQL          *myData;
    char strUser[20]="usuario"; //MySQL user
    char strPwd[20]="password"; //MySQL password usuario
    char strdb[20]="db_prueba";//MYSQL data base
```



```

        if(!(myData = mysql_init(0)))
        {
            cout << "ERROR: Imposible crear objeto myData." <<
endl;
            getchar();
            return 1;
        }
        if(!mysql_real_connect(myData, NULL, strUser, strPwd,
strdb, MYSQL_PORT, NULL, 0))
        {
            cout << "Error de conexión en puerto " << MYSQL_PORT
<< " ERROR: " << mysql_error(myData) << endl;
            mysql_close(myData);  getchar();
            return 1;
        }

        if(mysql_select_db(myData, "db_prueba"))
        {
            cout << "ERROR select_db: " << mysql_error(myData) <<
endl;
            mysql_close(myData);  getchar();
            return 1;
        }

        return 0;
    }

```

La recuperación de registros de una tabla MySql desde C++ se realiza mediante la utilización de "resultsets". Los pasos para establecer un resultset de lectura son los siguientes:

- Construir la sentencia "select" que ejecutar:

```

strcpy(strSQL,"select * from tb_cliente where
apellidol='GARCIA'");

```

en la cadena strSQL tenemos la sentencia que ejecutar.

- Ejecutar la sentencia mediante la función mysql\_query:

```

if(mysql_query(myData, strSQL))
strcpy(strError,mysql_error(myData));

```

el manejador de la base de datos es myData y la cadena strSQL contiene la sentencia que ejecutar.

- Recuperar los registros leídos mediante las funciones mysql\_store\_result y mysql\_fetch\_row. La función mysql\_store\_result transfiere al cliente el resultado de la consulta realizada (resultset) utilizando para ello la estructura MYSQL\_RES. La función mysql\_fetch\_row recupera el siguiente registro del resultset. Si no hay mas registros devuelve NULL.

```

    if(!(res = mysql_store_result(myData)))
    strcpy(strError,mysql_error(myData));
    while ((row = mysql_fetch_row(res)) // recupera registros
    del resultset
    { .... } ;
    suponemos res una variable del tipo MYSQL_RES y row una variable
    del tipo MYSQL_ROW.

```

Si la sentencia que vamos a ejecutar no devuelve registros (por ejemplo una sentencia INSERT), el paso 3 no es necesario, pues la función `mysql_query` la ejecutará directamente. La acción `mysql_close(myData)` cierra la conexión con la base de datos.

Podemos combinar el acceso a la base de datos con la utilización de hebras. En caso de que varias hebras de un mismo programa necesiten acceder a la base de datos, podremos establecer una conexión para cada una de ellas. De esta manera, cada hebra podrá acceder a la base de datos de manera independiente a las demás.

A continuación presentamos un ejemplo de utilización de diferentes sentencias de acceso a base de datos combinadas con el uso de hebras desde un programa C++. Suponiendo que se ha instalado y arrancado MySQL y se ha configurado Dev-C++ según lo indicado en esta sección, deberán seguirse los siguientes pasos:

- Desde la línea de comandos de MySQL, creamos la base de datos `rocketdb` mediante la sentencia **CREATE DATABASE rocketdb**;
- Desde la línea de comandos, ejecutamos el script [CreateTables.sql](#) mediante la instrucción **source [directorio>CreateTables.sql]**. Previamente debemos copiar el script anterior en el directorio deseado. Una vez realizado esto, tendremos creado el usuario `rocket` y las tablas `t_sim_pos` y `t_sim_vel`.
- Ahora ya podemos ejecutar el programa [Delete Insert.cpp](#). Este programa primero elimina los registros que pudiera haber en las tablas creadas anteriormente y, a continuación, realiza una serie de inserciones en las mismas tablas. Se utiliza la función **mysql\_affected\_rows(myData)** después de la sentencia "delete" para saber el número de registros eliminados. Cada tabla es tratada por una hebra independiente.
- En el programa [Delete Insert Select.cpp](#) se muestra un ejemplo que utiliza sentencias delete, insert y select. Recupera registros mediante un "resultset".
- En [4] se presenta un ejemplo completo de acceso a base de datos utilizando como base los programas de programación concurrente y bases de datos descritos en este libro.



## 10.7. Problemas propuestos

### 10.7.1. Enunciado 1.

Se debe construir un programa para calcular los valores de la presión total en función del tiempo ( $t$ ), la densidad del aire ( $\rho$ ), la velocidad ( $V$ ) y la altura ( $z$ ) en cada instante de muestreo. Establecemos intervalos de muestreo cada 0,5 metros de altura.

Dividimos el problema en dos partes:

- Zona de pista ( $z$  constante)
- Zona de despegue ( $z$  creciente)

Utilizamos la expresión siguiente:  $PT = PE + PD$  (1)

Donde  $PT$  es la presión total;  $PE$  la presión estática, y  $PD$  la presión dinámica (Pitot)

La presión dinámica se calcula según la expresión

$$PD = \frac{1}{2} * \rho * V^2 \quad (2)$$

donde  $\rho$  es la densidad del aire, y obtenemos:

$$PT = PE + \frac{1}{2} * \rho * V^2 \quad (1 \text{ bis})$$

Según la ecuación de los gases perfectos:  $PE = \rho * R * T$  (3) donde  $R = 287,05 \text{ J/KgK}$  y como los gases están en equilibrio hidrostático, podemos utilizar la ecuación  $dPE/dz = -\rho * g$ . Combinando las dos ecuaciones tendremos

$$dPE = -\frac{PE * g}{R * T} * dz \quad (4)$$

La temperatura varía en función de la altura según las expresiones siguientes:

Tramo 1  $0 \leq z < 11000 \rightarrow T = T_0 - \alpha_1 * z$  donde  $\alpha_1 = 6,5 \text{ K/Km}$  y  $T_0 = 288,15 \text{ K}$

Tramo 2  $11000 \leq z < 20000 \rightarrow T = T_{ii}$  (constante = 216,65 K con los valores anteriores)

Tramo 3  $20000 \leq z \leq 32000 \rightarrow T = T_{ii} + \alpha_2 * (z - 20000)$  donde  $\alpha_2 = 1,0 \text{ K/Km}$

**Parte 1. Zona de pista:** En la zona de pista, la altura no varía, la densidad ( $\rho$ ) y la presión estática ( $PE$ ) se mantienen constantes. Suponemos que  $PE=PE_0=1,01325*10^5$  Pa y  $\rho_0=1,225$  Kg/m<sup>3</sup>

La aceleración, de  $1,5$  m/s<sup>2</sup>, es constante. El tiempo que está en pista antes del despegue es de 40 segundos (o sea que la velocidad final en el momento antes del despegue es de 60m/s).

## Parte 2. Zona de despegue:

Consideraremos la zona de despegue desde 0 hasta 10000 metros, por tanto aplicaremos la ecuación de variación de la temperatura correspondiente al tramo 1 en la ecuación (4) y obtenemos la ecuación (5).

$$dPE = -\frac{PE * g}{R * (T_0 - \alpha_1 * z)} * dz \quad (5)$$

Una vez calculada la presión estática correspondiente a la altura  $z$ , calculamos el valor de  $\rho$  con la ecuación (3) :  $\rho = \frac{PE}{R * (T_0 - \alpha_1 * z)}$ . Con los valores de  $PE$ ,  $V$  y  $\rho$  podemos calcular la presión total  $PT$  con la ecuación (1 bis).

Para el cálculo del diferencial de presión que nos da la ecuación (5) consideramos intervalos de muestreo a cada 0,5 m de altura, suponemos una pendiente de despegue del 10% (sen  $\alpha = 0.09950372$ ) y una aceleración de  $0,5$  m/s<sup>2</sup>. Con estos valores, el valor de  $t$  en función de  $z$  es

$$t = \sqrt{\frac{2 * z}{0,5 * 0,09950372}} \quad \text{y la velocidad es } V(t) = V_0 + t, \text{ donde } V_0 = 60 \text{ m/s y } t = 0 \text{ en}$$

el instante inicial de despegue ( $z=0$ ). La simulación acaba cuando la altura llega a 10.000 metros.

- Se debe diseñar un mecanismo de control, por medio de una hebra, que sincronice un sistema de alarmas que se encienda a lo largo del vuelo 10 segundos antes de cambiar de etapa del vuelo. Para ello deberá utilizarse un semáforo que mantenga la hebra en espera hasta que se cumpla la condición de alarma.
- Los datos se deben guardar al final de la simulación en una base de datos, con la estructura necesaria para almacenar la información de la simulación y poder mostrarla en un gráfico.



### 10.7.2. Enunciado 2.

Con los ejemplos del péndulo y del movimiento parabólico, almacenar los datos en una tabla que identifique los datos del péndulo con un identificador y los del movimiento con otro diferente. Ambos programas deben ejecutarse a la vez e ir grabando la información en la base de datos. Cada programa graba los datos en un búfer de la memoria de forma independiente. Al terminar la simulación se deberá grabar la información del búfer en la base de datos.

### 10.7.3. Enunciado 3.

Diseña un programa que gestione la lista de pasajeros que podrá abordar el vuelo de las 12:00 h con destino a Londres de TACA 456.

Hay una lista de pasajeros de 500 personas que desean volar a Londres durante el día. Se han establecido exclusiones utilizando una matriz de 4x4. Esta matriz se llena con números del 1 al 600 generados de forma aleatoria, por lo que pueden repetirse. El vuelo de TACA tiene capacidad para 500 personas. El programa deberá mostrar en pantalla el listado con los datos de las 500 primeras personas de la lista cuyo identificador no se encuentre en la diagonal principal de la matriz de exclusiones.

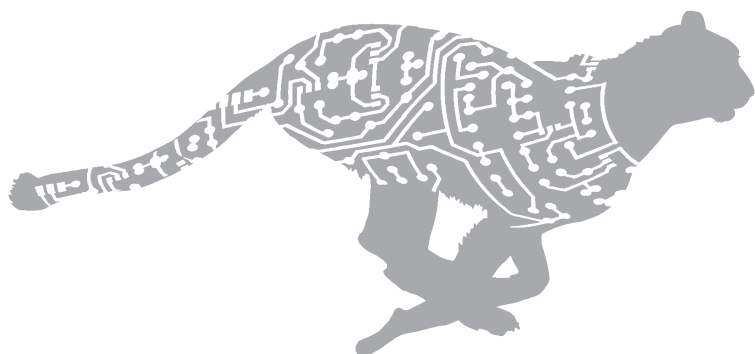
La información del pasajero incluye el nombre, el primer apellido, la edad, el DNI, y el identificador. El identificador es un número correlativo que identifica su número en la lista. Estos datos deberán almacenarse en una tabla de 500 elementos como máximo. El identificador y la edad son variables enteras positivas. El nombre, el primer apellido y el DNI son del tipo string.

Se deberán diseñar:

- Acciones para introducir los datos de cada estructura.
- Una función binaria para determinar si el identificador está o no en la diagonal de la matriz respectiva.
- Estructuras: tablas y tuplas.

En una tabla/estructura se relaciona a cada pasajero con alquileres de coches; y en otra tabla, con los destinos de abordaje y reservas de hoteles.

Al finalizar se debe grabar la información y las tablas necesarias en una base de datos.







## Bibliografia

- [1] ARINC AEE Committee (1999), *Avionics Application Software Standard Interface*.
- [2] Allworth, S. y R. Zobel. (1987), *Introduction to Real Time Software Design*. ISBN 0-387-91307-6. 140 p.p MacMillan, Springer Verlag New York Inc.
- [3] Ben-Ari, M. (2005) *Principles of Concurrent and Distributed Programming*. ISBN-13 9780321312839 384 pp. Addison Wesley.
- [4] Bernadi, J. (2011) *Disseny del sistema de processament de dades en vol d'un coet espacial*. Proyecto Final de carrera. Directores: Josefina López-Herrera, Miquel Sureda. Titulació: Ingeniería Superior Aeronáutica, Universidad Politécnica de Cataluña, Terrassa.
- [5] Booch, G., J. Rumbaugh, e I. Jacobson (1999) *The Unified Modeling Language Use Guide* ISBN-13: 9780201571684 512 pp. Addison Wesley.
- [6] Burns, Alan y Andy Wellings. (2003) *Sistemas de tiempo real y lenguajes de programación* ISBN 84-7829-058-3. Pearson Educación. S.A.
- [7] Buttazzo, G. (2000) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kliver Academic Publishers, Norwell, MA.
- [8] Calvez, J. (1993) *Embedded Real-Time Systems: A Specification and Design Methodology*, John Wiley & sons.
- [9] Cooling, J. (1991) *Software Design for Real-Time Systems*, Chapman and Hall.



- [10] Dijkstra, E. (1965) *"Solution of a Problem in Concurrent Programming Control"* *Communication of the ACM* **8**(9): 569.
- [11] Douglass, B. (1999) *Doing Hard Time: Developing Real-Time Systems with UML Object Frameworks and Patterns*. Addison-Wesley.
- [12] Fiege L., M. Mezini, G. Mühl y A.P. Buchmann. (2002) *Engineering Event-based Systems with Scopes*. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309{333, Málaga, Spain, Springer-Verlag.
- [13] Fiege, L., G. Mühl Rebeca *Event-Based Electronic Commerce Architecture*, 2000. <http://www.gkec.informatik.tu-darmstadt.de/rebeca>.
- [14] Fiege L., G. Mühl y F. C. Gärtner. (2002) *A Modular Approach to Build Structured Event-based Systems*. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385{392, Madrid, Spain, 2002. ACM Press.)
- [15] Laplante, A. (2004) *Real-Time System Design and Analysis*. IEEE PRESS Wiley-Interscience. **ISBN** 0-471-22855-9 3ed.
- [16] Larman, C. (2002) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Second Edition. Prentice Hall, Englewood Cliffs, NJ.
- [17] Liu, J. (2000) *Real Time Systems*, Prentice Hall. Upper Saddle River. NJ.
- [18] Silberschatz, A. y P. Galvin. (1994) *Operating System Concepts and Computer Networks*. Prentice Hall.
- [19] Silberschatz, A., H. Korth y S. Sudarshan. (2001) *Database System Concepts*. McGraw Hill.
- [20] Stroustrup, B. y M. A. Ellis. (2001) *The Annotated C++ Reference Manual*. Addison-Wesley, Boston, MA.
- [21] Tanenbaum, A. S. (2003) *Sistemas operativos modernos*. **ISBN** 0-13-0311358-0 976pp. Pearson Educación México.
- [22] Vahid, F. y T. Givargis. (2002) *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, Hoboken, NJ.
- [23] Van Harmelen, Mark (2001). *Object Modeling and User Interface Design: Designing Interactive Systems*. **ISBN**-13 9780201657890 480 pp. Addison Wesley.