

---

# FUNDAMENTOS DE SOFTWARE

---

**Paula Villanueva Núñez**

Doble grado de ingeniería informática y matemáticas

Universidad de Granada

# TEMA 1. SISTEMA DE CÓMPUTO

---

## 1. Componentes de un Sistema de Cómputo

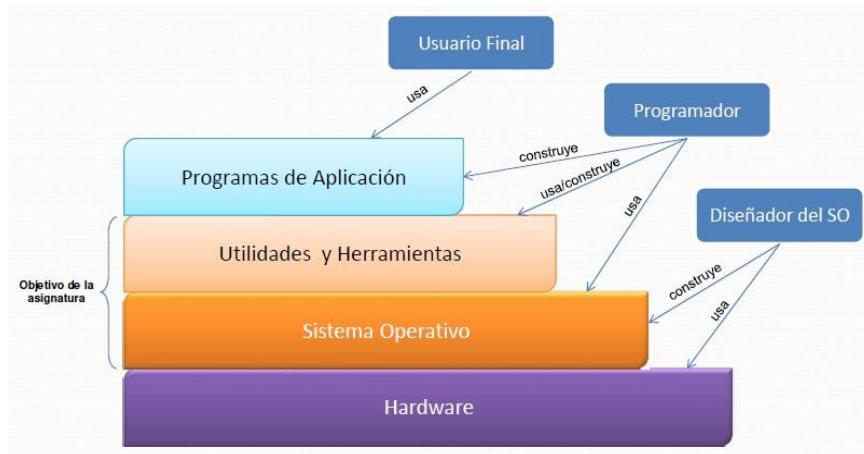
---

### 1.1 Definiciones básicas

- **Informática** (*información y automática*): conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores.
- **Computador**: máquina capaz de aceptar unos datos de entrada, efectuar con ellos operaciones lógicas y aritméticas, y proporcionar la información resultante a través de un medio de salida; todo ello sin intervención de un operador humano y bajo el control de un programa de instrucciones previamente almacenado en el propio computador.
- **Bit** (*Binari Digit*): es una posición o variable que toma el valor 0 ó 1 y es la unidad mínima de información. Además, codifica información.
- **Instrucción u orden**: conjunto de símbolos insertados en una secuencia estructurada o específica que el procesador interpreta y ejecuta.
- **Programa**: conjunto ordenado de instrucciones que dan a la computadora indicándole las operaciones o tareas que desea que realice.
- **Lenguaje de programación**: lenguaje formal diseñado para describir el conjunto de acciones consecutivas que un equipo debe ejecutar.
- **Lenguaje máquina**: lenguaje cuyas instrucciones interpretan los circuitos eléctricos de la unidad de control.
- **Hardware**: conjunto de componentes de componentes que integran la parte material de un computador; conjunto de circuitos eléctricos, cables...
- **Firmware**: bloque de instrucciones de máquina para propósitos específicos grabado en una memoria, normalmente de lectura/escritura que establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo de cualquier tipo.
- **Software** (soporte lógico): conjunto de programas, instrucciones y reglas informáticas que permiten ejecutar distintas tareas en una computadora.

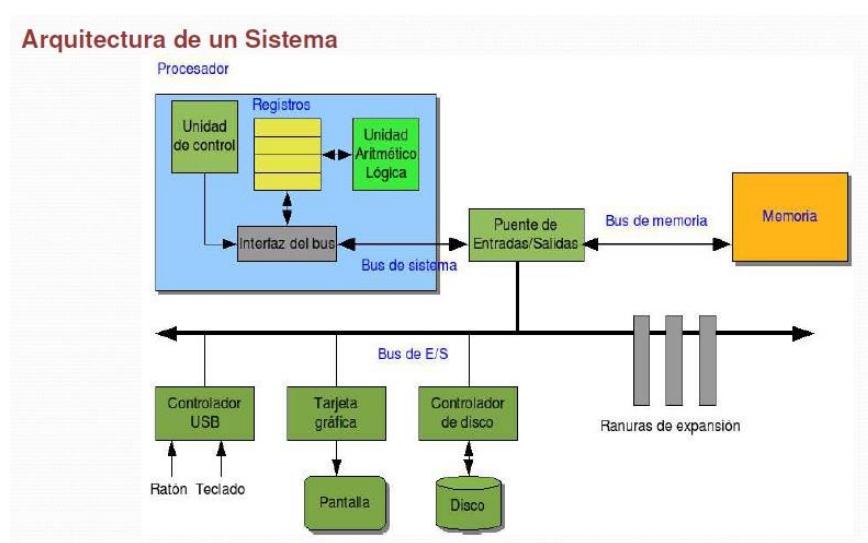
- **Sistema informático:** es un sistema que permite almacenar y procesar información; es el conjunto de partes interrelacionadas: hardware, software y personal informático.
- **Dato:** conjunto de símbolos utilizados para expresar o representar un valor numérico, un hecho, un objeto o una idea; en la forma adecuada para ser objeto de tratamiento. Es un elemento de información.
- **Byte:** unidad mínima para direccionar. Además, 1 Byte = 8 bits.

Bytes	bits
1 KB = $2^{10}$ B	1 Kb = $2^{10}$ b
1 MB = $2^{10}$ KB	1 Mb = $2^{20}$ b
1 GB = $2^{10}$ MB	1 Gb = $2^{30}$ b
1 TB = $2^{10}$ GB	1 Tb = $2^{40}$ b
1 PB = $2^{10}$ TB	1 Pb = $2^{50}$ b



## 2 Capa Hardware

### 2.1 Estructura de un Ordenador

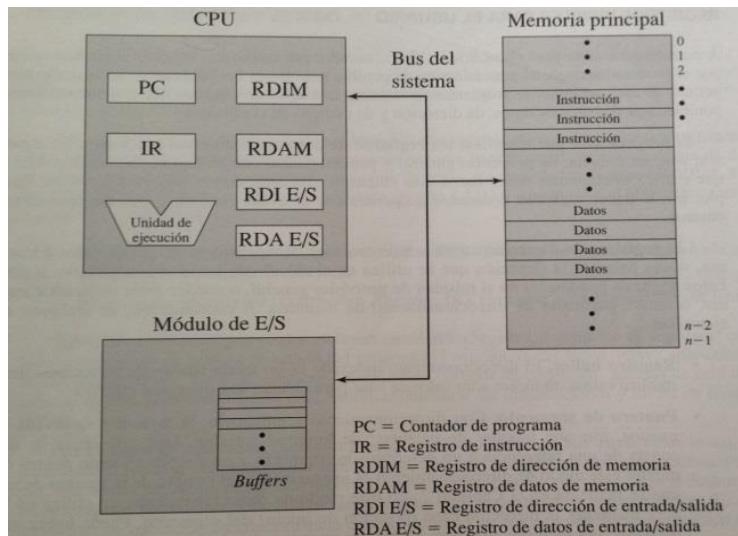


### 2.2 Elementos básicos

- **Procesador:** controla el funcionamiento del computador y realiza funciones de procesamiento de datos. Cuando sólo hay un procesador, se denomina **CPU** (Central Processing Unit). Una de sus funciones es el intercambio de datos, para ello se compone de registros internos:
  - **Registro de dirección de memoria (RDIM):** especifica la dirección de memoria de la siguiente lectura o escritura.
  - **Registro de datos de memoria (RDAM):** contiene los datos leídos o que van a ser escritos en memoria.
  - **Registro de dirección de E/S (RDI E/S):** especifica dispositivo de E/S.
  - **Registro de datos E/S (RDA E/S):** permite intercambio de datos entre dispositivos E/S y procesador.
- **Memoria principal/real/primaria:** conjunto de posiciones definidas mediante direcciones secuenciales que contiene un patrón de bits interpretable como instrucción o dato. Es volátil.
- **Módulo de E/S:** transfiere datos entre dispositivos externos, memoria y procesador. Contiene buffers (zonas de almacenamiento internas que mantienen temporalmente los

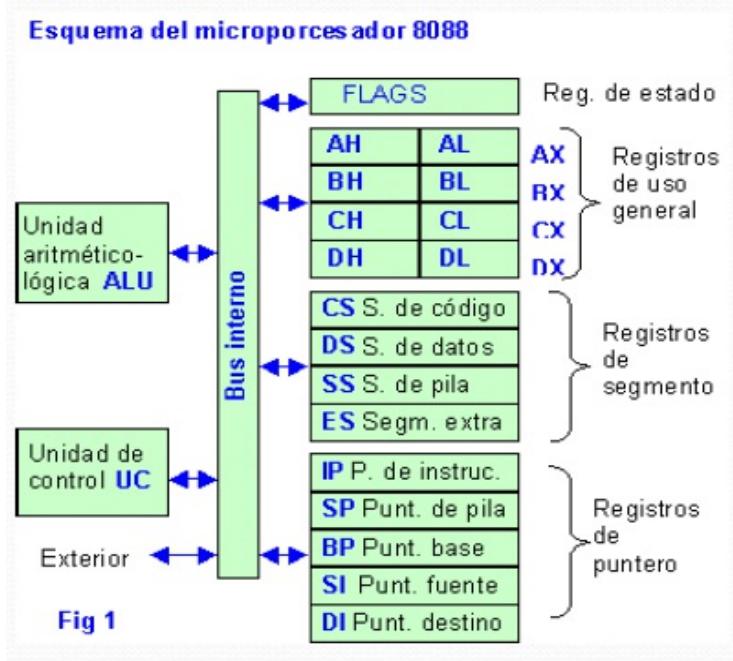
datos hasta que se puedan enviar).

- **Bus de sistema:** proporciona comunicación entre procesadores, memoria principal y módulos de E/S.



## 2.3 Registros del Procesador

- **Registros visibles para el usuario/programador:** permiten al programador en lenguaje máquina o en ensamblador minimizar las referencias a memoria principal optimizando el uso de registros. Tipos:
  - **Registros de datos:** de propósito general con cualquier instrucción de la máquina que realice operaciones con datos aunque hay restricciones.
  - **Registros de dirección:** contienen direcciones de memoria principal de datos e instrucciones, o una parte de la dirección que se utiliza en el cálculo de la dirección efectiva o completa. Por ejemplo:
    - **Registro índice:** el direccionamiento indexado implica sumar un índice a un valor de base para obtener una dirección efectiva.
    - **Puntero de segmento:** la memoria se divide en segmentos, que son bloques de palabras de longitud variable.
    - **Puntero de pila:** si hay direccionamiento de pila visible para el usuario, hay un registro dedicado que apunta a la cima de la pila. Se puede apilar y extraer.



- **Registros de control y estado:** usados por el procesador para controlar su operación y por rutinas privilegiadas del sistema operativo para controlar la ejecución del programa. Son esenciales para la ejecución de instrucciones.

- **RDIRM.**
- **RDAM.**
- **RDIE/S.**
- **RDAE/S.**
- **Contador de programa (PC):** contiene la dirección de la próxima instrucción que se leerá de la memoria.
- **Registro de instrucción (IR):** contiene la última instrucción leída.
- **Puntero de pila (SP).**
- **Registro de estado (bits informativos).**
- **Palabra de estado del programa (PWS):** registro o conjunto de registros que contienen información del estado del programa y códigos de condición. Los **códigos de condición** también llamados indicadores son bits asignados por el hardware del procesador teniendo en cuenta el resultado de operaciones.

## 2.4 Ejecución de instrucciones. Tipos de instrucciones

Un programa consta de un conjunto de instrucciones almacenadas en memoria. Procesar una instrucción consta de dos pasos:

1. El procesador **lee** (busca) instrucciones de la memoria, una cada vez.
2. El procesador **ejecuta** cada instrucción.

Se denomina **ciclo de instrucción** al procesamiento requerido por una única instrucción.

La ejecución de un programa consiste en repetir el proceso de búsqueda y ejecución de instrucciones.



Proceso a seguir:

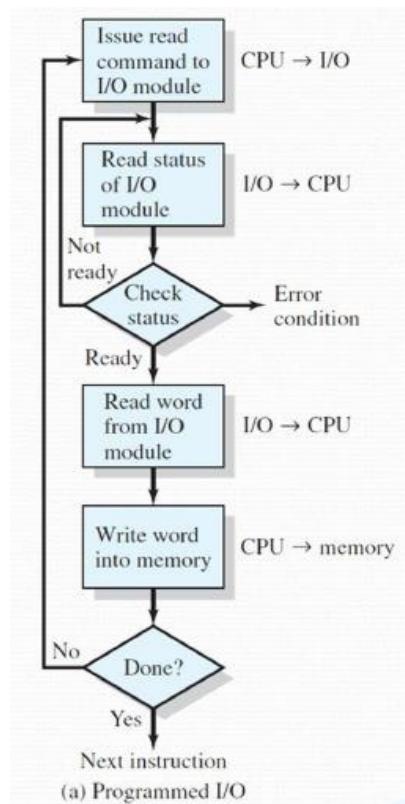
1. Se **lee** la instrucción cuya dirección está en el PC.
2. Se **incrementa** el PC en una unidad.
3. Se **ejecuta** la instrucción.

La ejecución del programa se **detiene** sólo si se apaga la máquina, se produce un error irrecuperable o se ejecuta una instrucción del programa que para el procesador.

**Tipos** de instrucciones:

- **Procesador-memoria:** transferencia de datos desde el procesador a la memoria o viceversa.
- **Procesador-E/S:** se envían datos a un dispositivo periférico o se reciben desde el mismo, transfiriéndolos entre el procesador y un módulo de E/S.
- **Procesamiento de datos:** el procesador realiza operaciones aritmético-lógicas sobre los datos.
- **Control:** una instrucción puede especificar que se va a alterar la secuencia de ejecución.

## 2.5 Técnicas de Comunicación de E/S



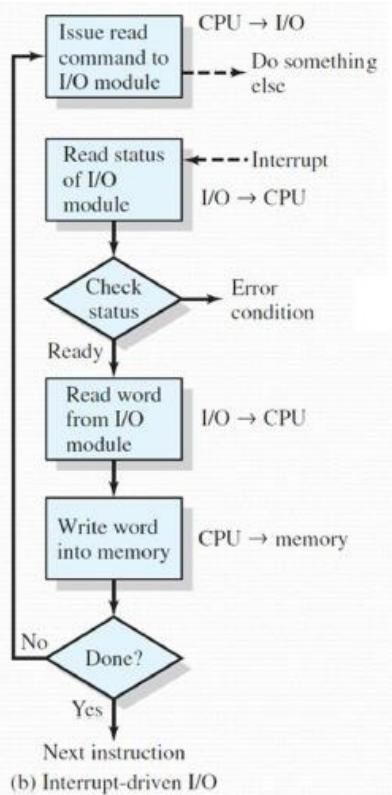
Se pueden intercambiar datos directamente entre un módulo de E/S y el procesador. Hay tres **técnicas** para llevarlo a cabo:

- **E/S programada:** el procesador encuentra una instrucción con la E/S. Se genera un mandato al módulo de E/S apropiado.

El procesador adopta un papel activo mientras se atiende la instrucción de E/S y comprueba periódicamente el estado de ejecución del módulo de E/S hasta que ha finalizado la operación.

**PROBLEMA:** el procesador pasa mucho tiempo esperando la finalización del módulo de E/S y el sistema se degrada gravemente.

**SOLUCIÓN:** mientras se atiende al módulo de E/S, se intenta que el procesador pueda continuar con trabajo útil.



(b) Interrupt-driven I/O

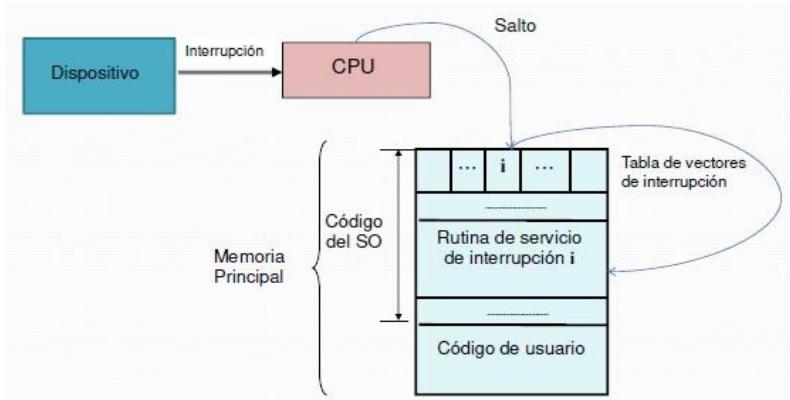
- **E/S dirigida de interrupciones:** evento que interrumpe el flujo normal de ejecución y que está producido por un elemento externo al procesador. Es un evento asíncrono.

**PROBLEMA:** en transferencias considerables de memoria a dispositivo o viceversa conlleva un uso excesivo del procesador.

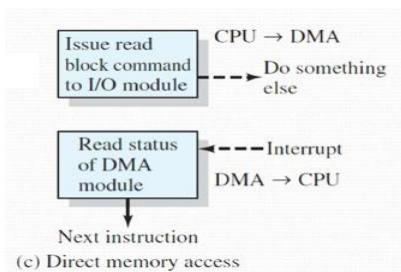
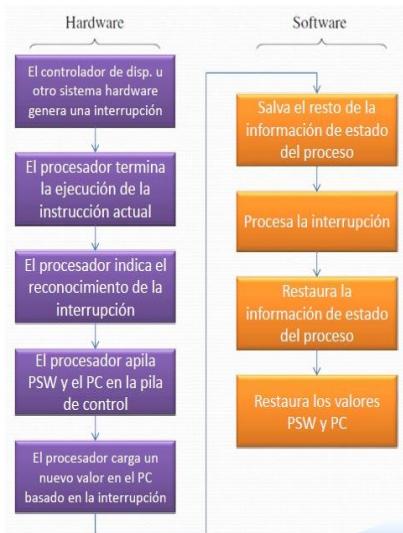
**SOLUCIÓN:** Acceso Directo a Memoria. En un solo mandato se genera todo lo necesario para realiza la transferencia de información de memoria al dispositivo o viceversa.

### Ciclo de instrucción con interrupciones:





### Tratamiento de interrupciones vectorizadas:



- **Acceso Directo a Memoria (Direct Memory Access, DMA):** realizada por un módulo separado conectado en el bus del sistema o incluida en un módulo de E/S. Útil cuando el procesador desea leer o escribir un bloque de datos.

## 2.6 Excepciones

Una excepción es un **evento inesperado** generado por alguna condición que ocurre durante la ejecución de una instrucción (ejemplo: desbordamiento aritmético, dirección inválida, instrucción privilegiada...). Es un evento **síncrono**.

## 2.7 Protección del procesador

Funcionamiento en Modo Dual. ¿Qué ocurre si un programa accede a la memoria donde se alojan los vectores de interrupciones? ¿Qué pasa si las modifica? El procesador dispone de diferentes **modos de ejecución** de instrucciones:

- **Instrucciones privilegiadas (modo supervisor/kernel)**: su ejecución puede interferir en la ejecución de un programa cualquiera o programa del SO (ejemplo, escribir en el puerto de un dispositivo).
- **Instrucciones no privilegiadas (modo usuario)**: su ejecución no presenta ningún problema de seguridad para el resto de programas (ejemplo, incrementar un contador).

## 2.8 Protección de los dispositivos de E/S

Es fundamental para evitar que unos usuarios puedan acceder indiscriminadamente a los periféricos (sobre todo los de almacenamiento).

Para conseguirlo, las instrucciones máquina para acceso a los dispositivos de E/S no pueden ejecutarse en modo usuario, son privilegiadas. Cualquier acceso a los dispositivos desde un programa de usuario se hará mediante peticiones al SO.

## 2.9 Protección de memoria

Cada programa en ejecución requiere de un espacio de memoria.

Por lo que hay que proteger la zona de memoria asignada y la memoria en la que está el código del SO (tabla de vectores de interrupción, rutinas de tratamiento de cada interrupción).

## 3. El Sistema Operativo

---

Es un programa o conjunto de programas que controla la ejecución de los programas de aplicación y que actúa como interfaz entre el usuario de una computadora y el hardware de la misma. Sus objetivos son:

- **Facilidad y comodidad** en el uso.
- **Eficiencia**: existen más programas que recursos. Hay que repartir los recursos entre los programas.
- Capacidad para **evolucionar** debido a las siguientes razones:

- Actualizaciones del hardware y nuevos tipos de hardware.
- Mejorar y/o aportar nuevos servicios.
- Resolución de fallos.

### 3.1 El SO como interfaz Usuario/Computadora

Presenta al usuario una máquina abstracta más fácil de programar que el hardware subyacente:

- Oculta la complejidad del hardware.
- Da tratamiento homogéneo a diferentes objetos de bajo nivel (archivos, procesos, dispositivos...).

Una aplicación se puede expresar en un lenguaje de programación y la desarrolla un programador de aplicaciones.

Es más fácil programar las aplicaciones en lenguajes de alto nivel que en el lenguaje máquina que entiende el hardware.

Un SO proporciona utilidades en las siguientes áreas:

- **Desarrollo de programas:** editores de texto, compiladores, depuradores de programas.
- **Ejecución de programas:** cargador de programas y ejecución de éstos.
- **Acceso a dispositivos de E/S:** cada dispositivo requiere su propio conjunto de instrucciones.
- **Acceso al sistema:** en sistemas compartidos o públicos, el SO controla el acceso y uso de los recursos del sistema: Shell, Interfaz gráfico.
- **Detección y respuesta a errores:** tratamiento a nivel software y hardware.
- **Contabilidad:** estadísticas de uso de los recursos y medida del rendimiento del sistema.

## 3.2 El SO como administrador de recursos

Un computador es un conjunto de recursos y el SO debe gestionarlos y para ello posee un mecanismo de control que cubre dos aspectos:

- Las **funciones** del SO actúan de la misma forma que el resto del software, es decir, son programas ejecutados por el procesador.
- El **SO** frecuentemente cede el control y depende del procesador para volver a retomarlo.

Por lo tanto:

- El SO dirige al **procesador** en el uso de los recursos del sistema y en la temporización de la ejecución de otros programas.
- Una parte del **código** del SO se encuentra cargado en la memoria principal (kernel y, a veces, otras partes del SO que se estén usando). El resto de la memoria está ocupada por programas y datos de usuario.
- La asignación de la **memoria principal** la realizan conjuntamente el SO y el hardware de gestión de memoria del procesador.
- El SO decide cuándo un programa en ejecución puede usar un dispositivo de E/S y también el acceso y uso de los ficheros. El procesador es también un recurso.

## 4. Utilidades del Sistema

---

### 4.1 Programas del servicio del SO

Se trata de un conjunto de programas de servicio que, en cierta medida, pueden considerarse como una ampliación del SO:

- **Compactación de discos:** para poder acceder a ellos más rápidamente.
- **Compresión de datos:** utilizando algoritmos de compresión.
- **Gestión de comunicaciones:** a través de tarjeta de red o de módem.
- **Visualizadores y navegadores de internet:** son programas que sirven para visualizar y acceder a páginas web.

- **Respaldo de seguridad.**
- **Recuperación de archivos eliminados.**
- **Antivirus.**
- **Salvapantallas:** evitan imágenes fijas durante largos períodos de tiempo que pueden deteriorar la pantalla.
- **Interfaz gráfica.**

## 4.2 Herramientas generales

Su misión es facilitar la **construcción** de las aplicaciones de los usuarios, sea cual sea la naturaleza de estas, tales como:

- Editores de texto.
- Compiladores.
- Intérpretes.
- Enlazadores.
- Cargadores/Montadores.

# TEMA 2. Introducción a los sistemas operativos

---

## 1. Componentes de un sistema multiprogramado

---

### 1.1 Sistemas multiprogramados y de tiempo compartido

Como el mejor sitio por donde empezar es el principio, comenzemos con una breve evolución de las computadoras, hasta un sistema multiprogramado:

1. **Raimundo Lulio** en el siglo XII escribió *Ars Magna*, en el que se dedicó a diseñar y construir una máquina lógica de naturaleza mecánica, donde las teorías, los sujetos y los predicados teológicos estaban organizados en figuras geométricas de las consideradas "perfectas".
2. **Leibnitz**, en 1650 construye la primera máquina que multiplica, cuando descubre *Ars magna* empieza a trabajar en la primera de algoritmo.
3. **Babbage** a principio del siglo XIX construye la primera máquina que procesa información.
4. A mediados del mismo siglo las matemáticas se empiezan a formalizar y estructurarse, considerándose un fin por sí mismas, surgiendo los axiomas de Peano, lógica... Todo gracias a personas y asociaciones como: **David Hilbert** y en 1950 **Burbaki**. Gracias a este movimiento, **Turing** crea las "máquinas de Turing" y posteriormente "la máquina universal de Turing" que podría considerarse uno de los precursores del ordenador.
5. A este ritmo los ordenadores iban surgiendo de manera natural. **A. Church** saca la máquina de cálculo **Post** y comienzan a aparecer los nuevos paradigmas: operacionales, funcionales y lógicos.
6. Surge el primer ordenador, **Colosus** para generar tablas de tiro, más tarde los ingleses lo utilizaron para manejar la información de los radares, aunque no se sabe si el primero fue alemán por los archivos encontrados tras su muerte a **Kamrao Zure** ingeniero para el ejército alemán durante la primera guerra mundial.

Aquí se acaba la historia con personajes y comienza la evolución de la arquitectura de los computadores:

## 1.2 Monoprograma o procesamiento en serie

La más arcaica, requiere de mucho tiempo, el programador tenía interacción directa con el hardware, (no existía el Sistema Operativo) si había un error el programa se detenía.

Estas máquinas eran utilizadas desde una consola que contenía luces, interruptores, algún dispositivo de entrada y una impresora. Los programas en código de máquina se cargaban a través del dispositivo de entrada. Si un error provocaba la parada del programa, las luces indicaban la condición de error. El programador podía entonces examinar los registros del procesador y la memoria principal para determinar la causa de error. Si el programa terminaba de forma normal, la salida aparecía en la impresora.

Problemas principales de estos sistemas:

- **Panificación** Las instalaciones contaban con plantillas impresas de reserva de tiempo, tiempo limitado (múltiplos de media hora). Por lo que se malgastaba el tiempo de procesamiento del computador.
- **Tiempo de configuración** Un trabajo (único programa) implicaba:
  - i. Carga en memoria compilador y lenguaje de alto nivel (programa en código fuente).
  - ii. Carga y enlace del programa objeto y funciones comunes.Estos pasos suponían montar y desmontar cintas o configurar tarjetas. Si ocurría un error, el usuario tenía que volver al comienzo de la secuencia de configuración. Lo que significa un tiempo elevado de configuración del programa que se va a ejecutar.

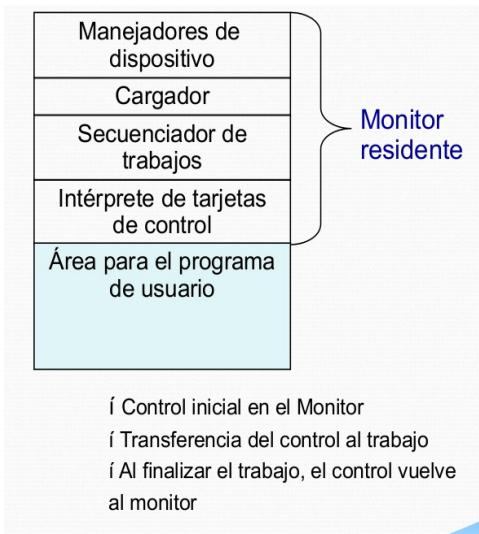
Se han desarrollado varias herramientas de software de sistemas con el fin de realizar el procesamiento serie más eficiente: bibliotecas de funciones comunes, enlazadores, cargadores, depuradores, rutinas de gestión de E/S disponibles como software común para todos los usuarios...

## 1.3 Sistemas en lotes sencillos o Sistemas Batch

El primer sistema operativo en lotes (y también el primer sistema operativo de cualquier tipo) surge en deseo de maximizar la utilización de las máquinas. La idea central se basaba en una pieza de software denominada **monitor**: el usuario no tiene que acceder directamente a la máquina, sino que introduce el trabajo por medio de una tarjeta o cinta al operador del computador, que crea un sistema de lotes con los trabajos enviados y los coloca en el dispositivo de entrada para que los utilice el monitor. Cuando un programa finaliza su procesamiento devuelve el control al monitor, que comenzará la carga del siguiente programa.

Análisis de este esquema desde distintos puntos de vista:

- **Punto de vista del monitor:** controla la secuencia de eventos desde la memoria principal, siempre disponible para su ejecución. Esta porción del monitor es denominado **monitor residente**. El resto del monitor está formado por conjunto de utilidades y funciones comunes que se encargan como subrutinas al comienzo del programa del usuario.  
El monitor lee de uno en uno los trabajos desde el dispositivo de entrada (lector de tarjetas o dispositivo de cinta magnética), coloca el trabajo en el área de programa de usuario, y se le pasa el control, que cuando ha terminado le devuelve el control al monitor, que lee el siguiente trabajo. Los resultados de cada trabajo se envían a un dispositivo de salida (impresora) para entregárselo al usuario.



- **Punto de vista del procesador:** el procesador ejecuta instrucciones de la zona de memoria principal que contiene el monitor. Por lo que se lee el siguiente trabajo y se almacena en otra zona de memoria principal. El procesador encontrará una instrucción de salto en el monitor que le indica al procesador que continúe la ejecución al inicio del programa de usuario. El procesador entonces ejecutará las instrucciones del programa usuario hasta que encuentre una condición de finalización o de error. Cualquiera de estas condiciones hace que el procesador ejecute la siguiente instrucción del programa monitor. Por tanto, la frase “se pasa el control al trabajo” significa que el procesador leerá y ejecutará instrucciones del programa de usuario, y la frase “se devuelve el control al monitor” indica que el procesador leerá y ejecutará instrucciones del programa monitor.
- Cuando el "control" lo tiene el monitor, ejecutará sus instrucciones y cuando no las del programa. El papel del monitor es de planificación, incluyendo instrucciones en algún lenguaje primitivo de **lenguaje de control de trabajos** (JCL).
- En resumen: El monitor o sistema operativo en lotes es un programa, que tiene en base, la habilidad del procesador para cargar instrucciones de memoria principal y tomar y abandonar el control.
- Necesita también un hardware con: protección de memoria, temporizador de trabajos, instrucciones privilegiadas que solo el monitor puede realizar e interrupciones. La protección de memoria y los privilegios dan lugar a los modos usuario y núcleo. El problema de la

programación en lotes era el tiempo que empleaba el ordenador en los periféricos.

## 1.4 Sistemas en lotes multiprogramados

En los trabajos automáticos de un sistema operativo en lotes simples el procesador se encuentra frecuentemente parado ya que los dispositivos de entrada y salida son mucho más lentos que este, así es como surge la **multiprogramación** o **multitarea**, se expande la memoria para que pueda albergar al sistema operativo (monitor residente) y más programas habiendo multiplexación entre ellos. Al haber varios programas a la vez podría haber un solapamiento del trabajo, para evitar esto, surgen las **interrupciones** de la mano de un avance de software y hardware, en el cual varios programas se desarrollan a la vez en sitios diferentes, esto se conoce como **S.pool**: cualquier trabajo puede suspender su actividad por la ocurrencia de un evento definido, como la finalización de una operación E/S. El procesador guardaría alguna forma de contexto (contador de programa u otros registros) y saltaría a una rutina de tratamiento de interrupciones: determinaría tipo interrupción, la procesaría y continuaría con el proceso interrumpido. Por ejemplo si varios programas requieren de una impresora, el programa que se está ejecutando escribe en un fichero lo que quiere imprimir y después lo vuelve a esta una vez que ha terminado de utilizar la impresora el programa anterior.

## 1.5 Sistemas de tiempo compartido

Teniendo en cuenta que los lotes son cerrados surgen las **colas**, sistemas de lotes abiertos, donde el SO controla los programas que esperan y los que se ejecutan, y cuándo termina estos dan paso a los siguientes en orden de prioridad. Por tanto, si un programa no utilizaba periféricos u otros recursos se podía quedar eternamente allí, o si era necesaria la interacción de varios usuarios directamente con la computadora, con esto surge la técnica **del tiempo compartido**, los programas tienen un tiempo limitado en la CPU. Estos intervalos de tiempo, también son conocidos como **cuantos de computación**.

Uno de los primeros sistemas operativos de tiempo compartido desarrollados fue el sistema CTSS (*Compatible Time-Sharing System*) desarrollado en el MIT por un grupo conocido como proyecto MAC y desarrollado para IBM.

.	Multiprogramación en lotes	Tiempo compartido
<b>Objetivo principal</b>	Maximizar el tiempo del procesador	Minimizar el tiempo de respuesta
<b>Fuente de directivas del sistema operativo</b>	Mandatos del lenguaje de control de trabajos proporcionadas por el trabajo	Mandatos introducidos al terminal.

1. **S.O. multiprogramado:** S.O. que permite que se ejecute más de un proceso simultáneamente y cuyos datos e instrucciones se encuentran en memoria principal.
2. **S.O. monousuario:** proporciona servicios a un único usuario.
3. **S.O. multiusuario:** proporciona servicios a varios usuarios simultáneamente.
4. **S.O. monoprocesador:** S.O. que gestiona un sistema de computación de un único procesador.
5. **S.O. multiprocesador:** S.O. que gestiona un sistema de computación de varios procesadores. Es decir, más de varias CPU en un mismo ordenador, pueden trabajar en distintas tareas o en **paralelo**, el mismo programa se divide en varios procesadores que trabajan simultáneamente.
6. **S.O. de tiempo compartido:** S.O. multiprogramado donde se realiza un reparto de tiempo del procesador en pequeños trozos de tal forma que todos los procesos pueden avanzar adecuadamente. Especialmente diseñado para sistemas interactivos.

Finalmente distintos ordenadores, con distintos sistemas operativos que trabajan simultáneamente, todos coordinados por un **macro sistema operativo**.

## Algunas preguntas

- La multiprogramación no tiene por qué ser de tiempo compartido. Pero para que sea posible el tiempo compartido es necesario un S.O. multiprogramado.
- ¿Un S.O. multiprogramado es un S.O. de tiempo compartido? ¿y al contrario?
- ¿Un S.O. de tiempo compartido tiene que ser multiusuario? ¿y monousuario?
- ¿Un S.O. monoprocesador tiene que ser monousuario? ¿y multiusuario?
- ¿Un S.O. multiprocesador tiene que ser monousuario? ¿y multiusuario?

## 1.6 Procesos

### 1.6.1 Concepto de proceso

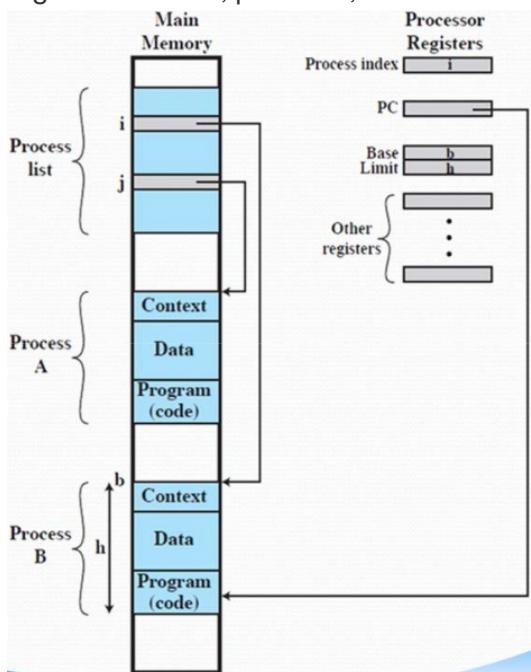
Se han dado distintas definiciones de proceso, algunas:

- Un programa en ejecución.
- Una instancia de un programa ejecutándose en un ordenador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad caracterizada por un solo flujo de ejecución, un estado actual y un conjunto de recursos del sistema asociados.

El diseño del software de la multiprogramación era sumamente complejo, los programadores acudían a métodos *ad hoc* para cooperación y coordinación de trabajos que generaba errores de sincronización, violación de la exclusión mutua, operación no determinista de un programa, interbloqueos.

Para solucionar estos problemas se necesita una forma sistemática de monitorizar y controlar la ejecución de varios programas en el procesador y aquí es donde entra el concepto de proceso, conformado por:

- Un programa ejecutable
- Los datos asociados que necesita un programa.
- El contexto de ejecución de un programa o estado del proceso, que es el conjunto de datos internos separada del proceso por el cual el sistema operativo es capaz de supervisar y controlar el proceso y el procesador para ejecutarlo. Ejemplos: contador de programa, registro de datos, prioridad, estado...



En la imagen, se muestra la manera de gestionar dos procesos, los contenidos de los registros de un programa que fue interrumpido fueron guardados en el contexto de ejecución del programa. Por esta razón un proceso puede verse como una estructura de datos, donde su **estado** se contiene en el contexto permitiendo así la cooperación y la coordinación entre procesos.

Notas de clase:

Cuando hay que leer de disco el sistema usuario no lee, los lenguajes dan sentencias, cuando se traducen son órdenes de llamadas de SO. El SO incorpora funciones propias donde corresponde al programa.

Lo mismo con lenguajes fuertemente tipado, como hay cosas que el compilador no puede resolver en tiempo de compilación por falta de información, el compilador añade instrucciones al texto (*paquete de soporte a la ejecución*), para poder ejecutarlo. Así, se puede denominar *proceso* al programa original ejecutado, es decir quitando la lógica añadida por el SO.

Ejemplo:

Cuando, con el navegador abierto, se abre otra ventana del navegador, para cada

ventanita hay una **instancia** con la sesión y la historia que guarda el navegador. Es decir, se tiene el mismo programa pero con varias instancias, es decir con distintos procesos. Podemos decir que un proceso o instancia es una ejecución particular del programa.

## 1.7 Implementación de procesos típica

Forma en la cual los procesos pueden gestionarse:

Dos procesos, A y B, se encuentran en una porción de memoria principal. Es decir, se ha asignado un bloque de memoria a cada proceso, que contiene el programa, datos e información de contexto. Se incluye a cada proceso en una lista de procesos que construye y mantiene el SO. La lista de procesos contiene una entrada por cada proceso, e incluye un puntero a la ubicación del bloque de memoria que contiene el proceso. La entrada podría también incluir parte o todo el contexto de ejecución del proceso. El resto del contexto de ejecución es almacenado en otro lugar, tal vez junto al propio proceso o frecuentemente en una región de memoria separada. El registro índice del proceso contiene el índice del proceso que le procesador está actualmente controlando en la lista de procesos. El contador de programa apunta a la siguiente instrucción del proceso que se va a ejecutar. Los registros base y límite definen la región de memoria y el registro límite le tamaño de la región (en bytes o palabras). El contador de programa y todas las referencias de datos se interpretan de forma relativa al registro base y no deben exceder el valor almacenado en el registro límite. Esto previene la interferencia entre los procesos.

En la imagen anterior (Figura 2.8), el registro índice del proceso indica que el proceso B está ejecutando. El proceso A estaba ejecutando previamente, pero fue interrumpido temporalmente. Los contenidos de todos los registros en el momento de la interrupción de A fueron guardados en su contexto de ejecución. Posteriormente, el SO puede cambiar el proceso en ejecución y continuar la ejecución del contexto de A. Cuando se carga el contador de programa con un valor que apunta al área de programa de A, el proceso A continuará la ejecución automáticamente.

- Busca proceso en lista, le da la dirección recuperar información con la que carga la información,
- traza es lo que se pasa, en la CPU

## 1.8 Bloque de control de un proceso (PCB, Process Control Block)

La memoria estaría llena de procesos o instancias. Así, el SO es el encargado de administrarlos de la forma correcta, para que todos sean ejecutados por el procesador de forma secuencial. Además, el SO tiene la capacidad de poder **bloquear un proceso**. Para que después pueda ser retomado como si nada, se necesita información sobre cada proceso, lo que se conoce como **bloque de control de un programa** (BCP), consta de:

Elemento del bloque de control	Descripción
<b>Identificador de proceso (PID Process Identifier)</b>	Identificador único que se le asocia a un proceso.
<b>Estado</b>	En qué situación se encuentra el proceso en cada momento según el modelo de los cinco estados: preparado, bloqueado, preparado...
<b>Prioridad</b>	Nivel de prioridad relativo al resto de procesos, el SO, cuenta con algoritmos para modificarla
<b>Contador de programa</b>	Dirección de la siguiente instrucción del programa que se va a ejecutar.
<b>Punteros a memoria</b>	Direcciones entre las que está un programa, dirección base y sus datos.
<b>Datos de contexto</b>	Datos del registro del procesador: registros que modifica, uso de recursos, contador de programa, registros procesador...
<b>Información del estado de E/S</b>	Incluye las peticiones pendientes de E/S, los dispositivos asignados a dichos procesos de E/S, lista de fichero en uso...
<b>Información de auditoría</b>	Incluye la cantidad de tiempo de de procesador y de reloj utilizados, límites de tiempo, registros contables...

La información de la lista anterior se almacena en una estructura de datos (PCB), que el SO crea y gestiona. El PCB contiene suficiente información de forma que es posible interrumpir el proceso cuando está corriendo y restaurar su estado de ejecución. Cuando un proceso se interrumpe, los valores de datos de contexto, se guardan en los campos correspondientes, y el estado de proceso cambia, así el SO queda libre para poner otro proceso en estado de ejecución. Por lo que un proceso está compuesto del código de programa, los datos asociados y del BCP.

## 1.9 Estados de los procesos

Se puede caracterizar el comportamiento de un determinado proceso, listando la secuencia de instrucciones que se ejecutan para dicho proceso (traza del proceso).

La figura 3.2 muestra el despliegue en memoria de tres procesos que no usan memoria virtual,

por lo que están representados por programas que residen en memoria principal. Además, existe un pequeño programa activador (dispatcher) que intercambia el procesador de un proceso a otro. La figura 3.3 muestra las trazas de cada uno de los procesos en los primeros instantes de ejecución. Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C. El proceso B ejecuta 4 instrucciones y se asume que la cuarta instrucción invoca una operación de E/S, a la cual el proceso debe esperar.

Desde el punto de vista del procesador se entremezclan las trazas de ejecución de los procesos y las trazas del código del SO.

### 1.9.1 Modelo de proceso de dos estados

La responsabilidad principal del sistema operativo es controlar la ejecución de los procesos; esto incluye determinar el patrón de entrelazado para la ejecución y asignar recursos a los procesos. El primer paso en el diseño de un sistema operativo para el control de procesos es describir el comportamiento que se desea que tengan los procesos.

Se puede construir el modelo más simple posible observando que, en un instante dado, un proceso está siendo ejecutado por el procesador o no. En este modelo, un proceso puede estar en dos estados: ejecutando ó no ejecutando. Cuando el SO crea un nuevo proceso, crea el BCP para el nuevo proceso e inserta dicho proceso en el sistema de estado no ejecutando. El proceso existe, es conocido por el SO, y está esperando su oportunidad y ejecutar. De cuando en cuando, el proceso actualmente en ejecución se interrumpirá y una parte del SO, el activador, seleccionará otro proceso.

Debe haber información correspondiente a cada proceso, incluyendo el estado actual y su localización en memoria: BCP. Los procesos que no están ejecutando deben estar en una especie de cola, esperando su turno para la ejecución. Existe una sola cola cuyas entradas son punteros al BCP de un proceso en particular. Alternativamente, la cola debe consistir en una lista enlazada de bloques de datos, en la cual cada bloque que representa un proceso. Si el proceso ha finalizado o ha sido abortado, se descarta (sale del sistema). En cualquier caso, el activador selecciona un proceso de la cola para ejecutar.

### 1.9.2 Llamadas al sistema

Por lo general, un sistema operativo está en un estado consistente y el código del servicio puede hacer uso de la funcionalidad general de un SO. Suele existir una única solicitud de interrupción de servicio, por lo que los servicios empiezan ejecutando el mismo código.

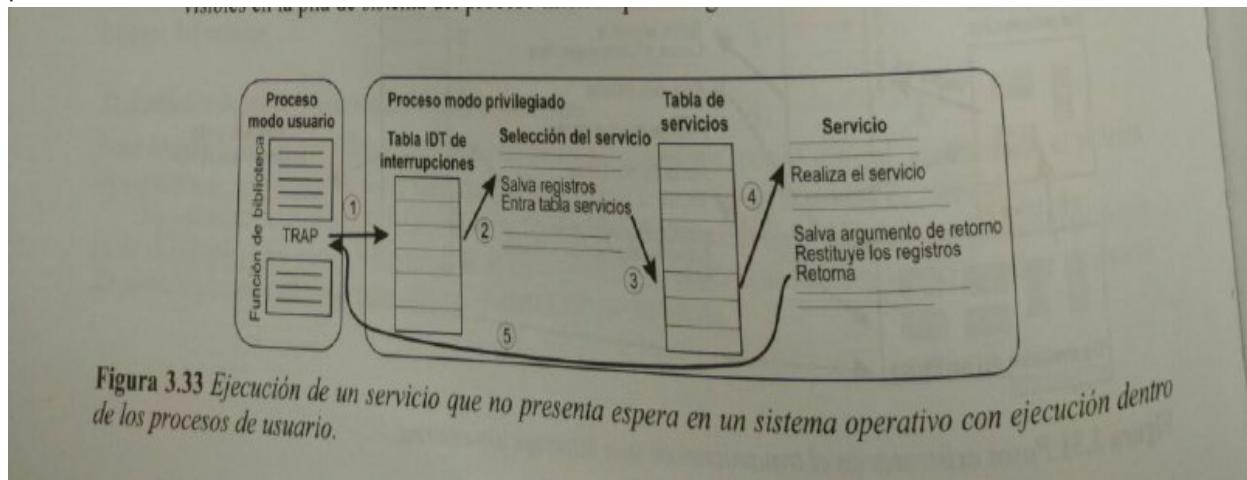
El servicio puede requerir una espera (que bloquerá al proceso) como leer un disco o no, como cerrar un proceso.

#### Cuando un servicio no requiere de espera:

- La instrucción **TRAP** genera la interrupción de petición de servicio.
- El procesador acepta la interrupción, por lo que el proceso pasa de modo usuario a modo privilegiado.

- A través de la tabla de interrupciones se ejecuta la rutina genérica que salva los registros visibles en la pila de sistema del proceso interrumpido. Seguidamente utiliza el identificador del servicio (almacenado en un registro) para entrar en la tabla de servicios y determina el punto de acceso del servicio solicitante.
- Llama al servicio y ejecuta el correspondiente código.
- Se retorna la rutina genérica que restituye los registros y en RETI, con lo que se devuelve la siguiente intrucción al TRAP.

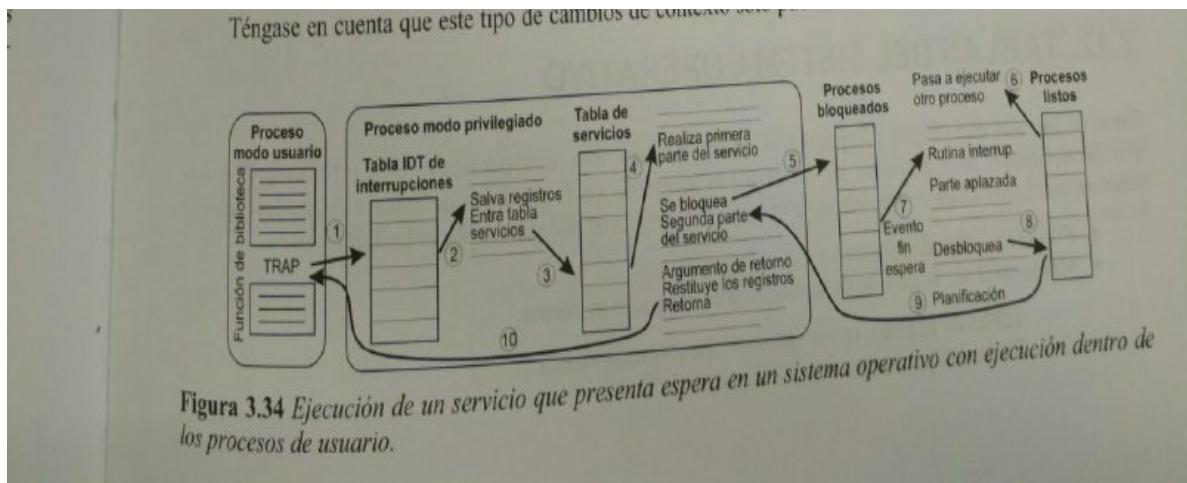
Durante la ejecución de un servicio, pudo llegar una interrupción que pondría a ejecutar otro proceso.



## Servicio que contiene espera

El tratamiento se divide en dos fases, una que inicial el servicio y otra que lo termina:

- La primera fase inicia el servicio (por ejemplo, lanza la orden de lectura de un disco), ejecuta el planificador, el proceso queda bloqueado, y se pone en ejecución el proceso seleccionado, por lo que se produce un cambio de contexto.
- Más adelante un evento indica el fin de la espera (ejemplo: un disco completa una lectura y genera una interrupción), esta interrupción ejecutará el contexto de otro proceso y podrá tener una parte aplazada.
- Si la operación se completó con éxito el proceso pasa de bloqueado a listo.
- Cuando el planificador seleccione otra vez este proceso, seguirá la ejecución completando la segunda fase del servicio, (ejemplo: copiando buffer de la información leída en el disco).
- Finalmente se genera el argumento de retorno del servicio y se restituyen los registros visibles, y se retorna al proceso que sigue su ejecución en modo usuario.



### 1.9.3 Modelo de los cinco estados

Trata de representar las actividades que le SO lleva a cabo sobre los procesos:

- **Creación de un proceso:** cuando se va a añadir un nuevo proceso a aquellos que se están gestionando en un determinado momento, el SO construye las estructuras de datos que se usan para manejar el proceso y reserva el espacio de direcciones en memoria principal para el proceso.

En un entorno por lotes, un proceso se crea como respuesta a una solicitud de trabajo. En un entorno interactivo, un proceso se crea cuando un nuevo usuario entra en el sistema. En ambos casos el SO es responsable de la creación de nuevos procesos. Razones para la creación de un proceso:

- **Nuevo proceso de lotes:** el SO dispone de un flujo de control de lotes de trabajos. Cuando el SO está listo para procesar un nuevo trabajo, leerá la siguiente secuencia de mandatos de control de trabajos.
- **Sesión interactiva:** un usuario desde un terminal entra en el sistema.
- **Creado por el sistema operativo para proporcionar un servicio:** el SO puede crear un proceso para realizar una función en representación de un programa de usuario, sin que el usuario tenga que esperar.
- **Creado por un proceso existente:** por motivos de modularidad o para explotar el paralelismo, un programa de usuario puede ordenar la creación de un número de procesos.

Cuando un SO crea un proceso a petición explícita de otro proceso, dicha acción se denomina **creación del proceso**.

Cuando un proceso lanza otro, al primero se le denomina **proceso padre**, y al proceso creado se le denomina **proceso hijo**.

- **Terminación de procesos:** todo sistema debe proporcionar los mecanismos mediante los cuales un proceso indica su finalización, o que ha completado su tarea.

Un trabajo por lotes debe incluir una instrucción HALT o una llamada a un servicio de SO específica para su terminación. Para una aplicación interactiva, las acciones del usuario indicarán cuándo el proceso ha terminado. Todas estas acciones tienen como resultado final la solicitud de un servicio al SO para terminar con el proceso solicitante.

Adicionalmente, un número de error o una condición de fallo puede llevar a la finalización de un proceso.

Razones para la terminación de un proceso:

- **Finalización normal**
- **Límite de tiempo excedido**
- **Memoria no disponible**
- **Violaciones de frontera**
- **Error de protección**
- **Error aritmético**
- **Fallo de E/S**
- **Instrucción no válida**

Por lo que para ello hace uso de cinco estados:

- **Ejecutando:** el proceso está actualmente en ejecución. Asumimos que el computador tiene un único procesador, de forma que solo un proceso puede estar en este estado en un instante determinado.
- **Listo o preparado:** un proceso que se prepara para ejecutar cuando tenga la oportunidad.
- **Bloqueado:** un proceso que no puede ejecutar hasta que se cumpla un evento determinado o se complete una operación E/S.
- **Nuevo:** un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el SO. Se trata de un nuevo proceso que no ha sido cargado en memoria principal, aunque su BCP si ha sido creado.
- **Saliente:** un proceso que ha sido liberado del grupo de procesos ejecutables por el SO, debido a que ha sido detenido o que ha sido abortado por alguna razón. Si el programa termina de ejecutarse, el SO actualiza la lista de procesos y libera esa zona de memoria, eso se convierte en basura.



Cuando se carga un programa nuevo:

- Instrucciones.
- Datos: variables sin asignar un valor, estas tienen asignado un valor, basura, el que está en memoria de otro programa. Hay compiladores que te permiten ver en la parte derecha, y puede saber que no le has asignado ningún valor, cuando la vas a utilizar te avisan (otros no te dan ningún error ni en compilación ni en ejecución). Lo que sí se recomienda es trastear con el compilador. Hay compiladores que en el for la i la ponen, otros no la hacen, calculan el número de vueltas que hace el bucle. Otros, que si haces referencia dentro del bucle a la i asocia sí lo tienen en cuenta, y si no, no. Te implementa la veces que se repite, esta variable se conoce como void variable **variable vacía**.

### 1.9.3.1 Transiciones entre estados

- **Nuevo -> Preparado**: el PCB está creado y el programa está disponible en memoria.
- **Ejecutándose -> Finalizado**: el proceso finaliza normalmente o es abortado por el SO a causa de un error no recuperable.
- **Preparado -> Ejecutándose**: el SO (planificador CPU) selecciona un proceso para que se ejecute en el procesador.
- **Ejecutándose -> Bloqueado**: el proceso solicita algo al SO por lo que debe esperar.
- **Ejecutándose -> Preparado**: un proceso ha alcanzado el máximo tiempo de ejecución ininterrumpida.
- **Bloqueado -> Preparado**: se produce el evento por el cual el SO bloqueó al proceso.
- **Preparado (o Bloqueado) -> Finalizado**: terminación de un proceso por parte de otro (en la mayoría de los SO modernos, no se permite. En modo super usuario sí se puede).

## 2. DESCRIPCIÓN Y CONTROL DE PROCESOS

---

### 2.1 Descripción de procesos: PCB

Diferencia entre una pila: el primero que se atiende el último que ha llegado cola: se atiende el primero que ha llegado

En las funciones cuando se va acumulando se forma nuevo **registro de activación** donde están los datos, se guarda la dirección en una pila, tiene una dirección que es lo que se conoce como puntero de pila, que soluciona todos estos problemas.

### Creación de un proceso: Inicialización de PCB

Una vez que el SO decide crear un proceso procederá de la siguiente manera:

1. **Asignar un identificador de proceso único al proceso:** se añade una nueva entrada a la tabla primaria de procesos, que contiene una entrada por proceso.
2. **Reservar espacio para proceso:** incluye todos los elementos de la imagen del proceso. Para ello, el SO debe conocer cuánta memoria se requiere para el espacio de direcciones privado (programas y datos) y para la pila de usuario. Estos valores se pueden asignar por defecto basándose en el tipo de proceso, o pueden fijarse en base a la solicitud de creación del trabajo remitido por el usuario. Por último, se debe reservar el espacio para el BCP.
3. **Inicialización del BCP:** la parte de identificación de proceso del BCP contiene el identificador del proceso y otros identificadores (como el indicador del proceso padre). En la información de estado de proceso del BCP, que se inicializa con la mayoría de entradas a 0, excepto el contador de programa (fijado en el punto entrada del programa) y los punteros de pila de sistema (fijados para definir los límites de la pila del proceso). La parte de información de control de procesos se inicializa en base a valores por omisión, considerando también los atributos que han sido solicitados para este proceso. La prioridad se puede fijar y el proceso no debe poseer ningún recurso (dispositivos de E/S, ficheros) a menos que exista una indicación explícita o que hayan sido heredados del padre.
4. **Establecer los enlaces apropiados:** si el SO mantiene cada cola del planificador como una lista enlazada, el nuevo proceso debe situarse en la cola de Listos o en Listos/Suspendidos.
5. **Creación o expansión de otras estructuras de datos:** el SO puede mantener un registro de auditoría por cada proceso que se puede utilizar posteriormente a efectos de facturación y/o de análisis de rendimiento de sistema.

En resumen:

1. Asignar identificador único al proceso.
2. Asignar un nuevo PCB.
3. Asignar memoria para el programa asociado.
4. Inicializar PCB:
  - 4.1 PC: dirección inicial de comienzo del programa.
  - 4.2 SP: dirección de la pila de sistema.
  - 4.3 Memoria donde reside el programa.
  - 4.4 El resto de campos se inicializan a valores por omisión.

## 2.2 Control de procesos

### 2.2.1 Modo de ejecución del procesador

- **Modo usuario:** no accede a todos los registros de memoria, contador de programa, o registro de instrucción, tampoco otras instrucciones
- **Modo núcleo, kernel, supervisor o sistema:** para pasar de uno a otro, se detecta que hay una operación que no se puede hacer en modo usuario.

## ¿Cómo utiliza el SO el modo de ejecución?

El modo de ejecución (incluido en PSW) cambia a modo kernel, automáticamente por hardware, cuando se produce:

- Una **interrupción**: se compara la prioridad.
- Una **excepción**: algo que no debería ocurrir pero ocurre, "overflow" se levanta una alerta entra el SO, si lo que pasa no tiene riesgo para el resto de los procesos se deja pasar (depende del compilador), otra que ocurre se lee de un fichero, se acaba y se pide que sigue leyendo en este caso se aborta el programa.
- Una **llamada al sistema**: leer disco, C pone un código que supone una rutina del SO. Llamo al sistema, se lee el dato que quiero, se manda interrupción, en el ciclo de instrucción mira a ver si era la siguiente.

Seguidamente se ejecuta la rutina del SO correspondiente al evento producido. Finalmente, cuando termina la rutina, el hardware restaura automáticamente el modo de ejecución a modo usuario.

### 2.2.2 Operación de cambio de modo

En la fase de interrupción, el procesador comprueba que no exista ninguna interrupción pendiente. Si no hay interrupciones pendientes, el procesador pasa a la fase de búsqueda de instrucción, siguiendo con el programa del proceso actual. Si hay una interrupción pendiente, el proceso actúa así:

1. Coloca el contador de programa en la dirección de comienzo de la rutina del programa manejador de la interrupción.
2. Cambia de modo usuario a modo núclero de forma que el código de tratamiento de la interrupción pueda incluir instrucciones privilegiadas.

El procesador pasa a la fase de búsqueda de instrucción y busca la primera instrucción del programa de manejo de interrupción, que le dará servicio. El contexto del proceso que se ha interrumpido se salvaguarda en el BCP del programa interrumpido.

Es decir, se ejecuta una rutina del SO en el contexto del proceso que se encuentra en estado "Ejecutándose".

La operación de cambio de modo se puede realizar siempre que el SO pueda ejecutarse, luego como resultado de una interrupción, excepción o llamada al sistema.

## 2.2.3 Pasos en la operación de cambio de usuario a kernel

1. El que detecta el cambio de modo es el hardware, cuando el compilador detecta una llamada al sistema, interrupción o excepción. El hardware salva del contador de programa (**PC**) y la palabra de estado del proceso (**PSW**) y cambia el bit de modo a **modo kernel**.
2. Se determina automáticamente la **rutina del SO** que debe ejecutarse y cargar el **PC** con su dirección de comienzo.
3. **Ejecutar la rutina**. Posiblemente la rutina comience **salvando** el resto de registros del procesador y termine **restaurando** en el procesador la información de registros previamente salvada.
4. Volver de la **rutina del SO** al proceso que se estaba ejecutando. El **hardware** automáticamente restaura en el procesador la información del **PC** y **PSW** previamente salvada.

## 2.2.4 Operación de cambio de contexto (cambio de proceso)

Un cambio de modo puede ocurrir sin que se cambie el estado del proceso actualmente en estado Ejecutando. Salvaguarda del estado y su posterior restauración comportan solo una ligera sobrecarga. Sin embargo, si el proceso actualmente en estado Ejecutando, se va a mover a cualquier otro estado, entonces el SO debe realizar cambios sustanciales en su entorno. Los pasos a realizar para cambiar de proceso son:

1. Salvar el estado del procesador, incluyendo el contador de programa y otros registros.
2. Actualizar el BCP que está en el estado Ejecutando (incluye cambiar el estado del proceso a otro estado). También se tienen que actualizar otros campos importantes, incluyendo la razón por la cual el proceso ha dejado el estado de Ejecutando y demás información de auditoría.
3. Mover el BCP a la cola apropiada (Listo, Bloqueado en el evento i, Listo/Suspendido).
4. Selección de un nuevo proceso a ejecutar.
5. Actualizar el BCP elegido (incluye pasarlo al estado Ejecutando).
6. Actualizar las estructuras de datos de gestión de memoria.
7. Restaurar el estado del procesador al que tenía en el momento en el que el proceso seleccionado salió del estado Ejecutando por última vez, leyendo los valores anteriores de contador de programa y registros.

Por tanto, el cambio de proceso, que implica un cambio en el estado, requiere un mayor esfuerzo que un cambio de modo.

Se puede realizar cuando el SO pueda ejecutarse y decida llevarlo acabo. Luego solamente como resultado de una interrupción, excepción o llamada al sistema.

## 2.2.5 Pasos en una operación de cambio de contexto (Dispatcher)

1. Salvar los registros del procesador en el **PCB** del proceso que actualmente está en estado "**Ejecutándose**".
2. **Actualizar** el campo **estado del proceso** al nuevo estado al que pasa e insertar el **PCB** en la **cola** correspondiente.
3. Seleccionar un **nuevo proceso del conjunto** de los que se encuentran en estado "**Preparado**" (Scheduler o Planificador de CPU).
4. **Actualizar el estado del proceso** seleccionado a "**Ejecutándose**" y sacarlo de la cola de preparados.
5. **Cargar los registros** del procesador con la información de los registros almacenada en el **PCB del proceso** seleccionado.
  - ¿Si se produce una interrupción, una excepción o una llamada al sistema, se produce necesariamente un cambio de modo? ¿y un cambio de contexto?

## 3. HEBRAS (hilos)

---

### 3.1 Concepto de Hebra (hilos)

El concepto de **proceso (tarea)** tiene dos características diferenciadas e independientes que permiten al SO:

- **Propiedad de recursos:** controla la asignación de los recursos necesarios para la ejecución de programas.
- **Planificación/ejecución:** la ejecución de un proceso sigue una ruta de ejecución (traza) a través de uno o más programas. Esta ejecución puede estar intercalada con ese u otros procesos. Por lo que un proceso tiene un estado de ejecución (Ejecutando, Listo...) y una prioridad de activación y esta es la que se planifica y activa por el SO.

Para distinguir estas características, la unidad que se activa se denomina **hilo (thread)**, o **proceso ligero**, mientras que la unidad de propiedad de recursos se denomina **proceso o tarea**.

- Proceso: programa más lo que necesita, la unidad de procesamiento, lo que el SO le asigna los recursos que necesite, puede ocurrir (cada sistema hace los que le sale de las narices). Por ejemplo se está navegando, se abren ventanas, procesos distintos, tenemos instrucciones y datos, el proceso es el mismo, lo que cambia son los datos, en vez de abrir procesos nuevos, se abren hebras, proceso el mismo, datos de cada hebra distinto, detección del bloque de control de procesos se

parten nuevos registros, direcciones, se ahorra mucha memoria de instrucciones.

navegado -> proceso asociado a ese programa, dentro de la misma ejecución (navegador) se abre una hebra. Hebra ejecución independiente del mismo proceso.

Proceso unidad de gestión de un programa al que se asocian memoria y gestión, si el código es el mismo en una hebra te ahorras el código en memoria, la hebra se paraleliza la ejecución del programa, el mismo código de programa abre las ejecuciones paralelas, así el programa avanza más rápido, hasta sincronizarse esto son hebras de un mismo proceso.

Esto se utilizar para acelerar programas, sobretodo programas de cálculo monstruosos, ventajas de la hebras, se consume menos memoria, los estados de las hebras son los mismos que los de los procesos.

**Multihilo** se refiere a la capacidad de un SO de dar soporte a múltiples hilos de ejecución en un solo proceso. Y el enfoque tradicional de un solo hilo de ejecución por proceso, en el que no se identifica con el concepto de hilo es el **monohilo**.

En un entorno multihilo, un proceso se define como la unidad de asignación de recursos y una unidad de protección. Características:

- La **tarea** se encarga de soportar todos los recursos necesarios (incluida la memoria).
- Cada una de las hebras permite la **ejecución del programa** de forma independiente del resto de hebras.

### 3.2 Modelo de cinco estados para hebras

Las hebras debido a su característica de ejecución de programas presentan cinco estados análogos al modelo de estados para procesos:

- Ejecutándose
- Preparado (listo para ejecutarse)
- Bloqueado
- Nuevo
- Finalizado

### 3.3 Ventajas de las hebras

Los mayores beneficios de los hilos provienen de las consecuencias del rendimiento:

1. Menor tiempo de **creación** de una hebra en un proceso ya creado que la creación de un nuevo proceso.
2. Menor tiempo de **finalización** de una hebra que de un proceso.

3. Menor tiempo de **cambio de contexto (hebra)** entre hebras pertenecientes al mismo proceso.
4. Facilitan la **comunicación** entre hebras pertenecientes al mismo proceso.
5. Permiten aprovechar las **técnicas** de programación concurrente y el multiprocesamiento simétrico.

## 4. GESTIÓN BÁSICA DE MEMORIA

---

### 4.1 Carga absoluta y reubicación

Los requisitos que la gestión de memoria debe satisfacer son:

- Reubicación.
- Protección.
- Compartición.
- Organización lógica.
- Organización física.

#### REUBICACIÓN

Es la capacidad de cargar y ejecutar un programa en un lugar arbitrario de la memoria.

En un sistema multiprogramado, la memoria principal disponible se comparte entre varios procesos. Por lo que no es posible saber anticipadamente qué programas residirán en memoria principal en tiempo de ejecución de su programa. Sería bueno poder intercambiar procesos en la memoria principal para maximizar la utilización del procesador, proporcionando un gran número de procesos para la ejecución. Una vez que un programa se ha llevado al disco, sería bastante limitante tener que colocarlo en la misma región de memoria principal donde se hallaba, cuando este se trae de nuevo a la memoria. Por el contrario, podría ser necesario **reubicar** el proceso a un área de memoria diferente.

Por tanto, no se puede conocer de forma anticipada dónde se va a colocar un programa y se debe permitir que los programas se puedan mover en la memoria principal, debido al *intercambio o swap*. Esto pone de manifiesto a aspectos técnicos relacionados con el direccionamiento.

Las instrucciones de salto contienen una dirección para referenciar una instrucción que se va a ejecutar a continuación. Las instrucciones de referencia de los datos contienen la dirección del byte o palabra de datos referenciados.

## CARGA

El primer paso en la creación de un proceso activo es cargar un programa en memoria principal y crear una imagen del proceso. En la carga de programa se debe satisfacer el direccionamiento siguiendo tres técnicas diferentes:

- Carga absoluta.
- Carga reubicable.
- Carga dinámica en tiempo real.

### Carga absoluta

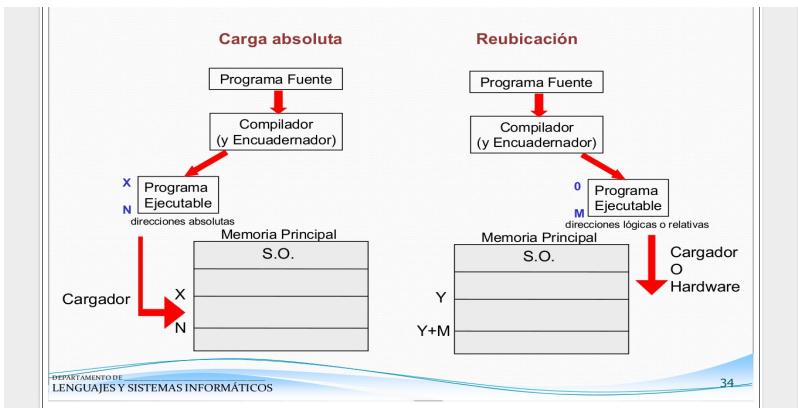
Consiste en asignar direcciones físicas (direcciones de memoria principal) al programa en tiempo de compilación. El programa no es reubicable.

Un cargador absoluto requiere que un módulo de carga dado deba cargarse siempre en la misma ubicación de la memoria principal. Por tanto, en el módulo de carga presentado al cargador, todas las referencias a direcciones deben ser direcciones de memoria principal específicas o absolutas.

Es preferible permitir que las referencias de memoria dentro de los programas se expresen simbólicamente, y entonces resolver dichas referencias simbólicas en tiempo de compilación o ensamblado. Cada referencia a una instrucción o elemento de datos se representa inicialmente como un símbolo. A la hora de preparar el módulo para su entrada a un cargador absoluto, el ensamblador o compilador convertirá todas estas referencias a direcciones específicas.

No es posible conocer previamente las direcciones de memoria de un programa, el SO es el que se encarga de posicionar y tener la información de la memoria.

Las direcciones que puede asignar un compilador son las físicas o absolutas y las que empiezan en cero, reales o lógicas, utilizar siempre absolutas implicaría que siempre se utilizase la misma dirección de memoria, en el caso de instrucciones se suma lo mismo o en el caso de bucles también se saltan.



Cuando se compila un programa se realiza la reubicación, a la carga relativa se le suma la dirección inicial en la que se inicia.

(en el núcleo, esto tiene direcciones es carga absoluta, en el se basa el grub)  
 si la máquina que tienes no tiene la memoria suficiente, ejecutar los mismos programas comparando con una de mayor, lo que tiene poca memoria, lo que tiene que hacer es coger y descargar los procesos de discos, la localización supone suma cálculo...

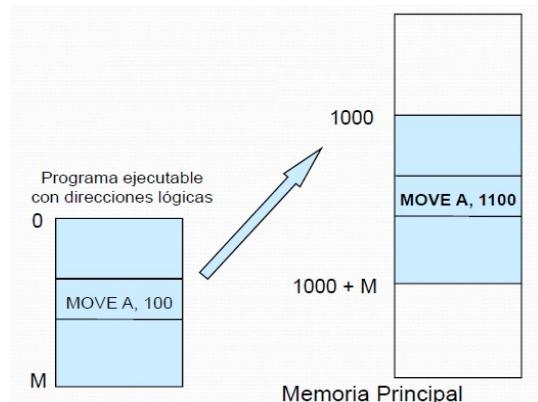
La direcciones lógicas se convierten en físicas después de la de compilación y antes de la ejecución o antes de que use la dirección, estos dos momentos son estáticas y dinámica.

## 4.2 Reubicacion estática

El **compilador** genera **direcciones lógicas** (relativas) de 0 a M. La decisión de **dónde ubicar** el programa en memoria principal se realiza en **tiempo de carga**. El **cargador** añade la dirección base de carga a todas las referencias relativas a memoria de programa.

Es decir, una vez que se conoce la posición de memoria en la que se va a colocar el programa, durante la carga a cada dirección de memoria lógica le suma la dirección base, la primera dirección donde se va a cargar, dando lugar a direcciones absolutas.

La **ventaja** que aporta es que el cambio de dirección lógica a absoluta solo se hace una vez, pero como **desventaja** ya no se puede.

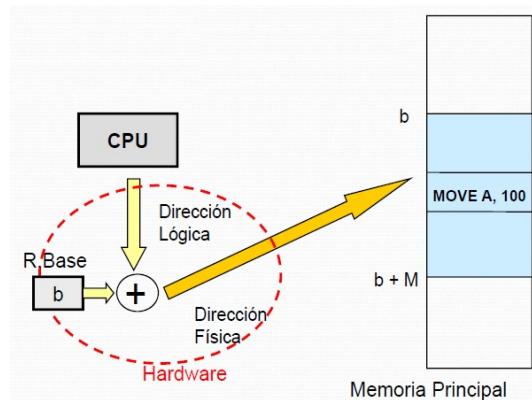


## 4.3 Reubicación dinámica

El **compilador** genera **direcciones lógicas** (relativas) de 0 a M. La **traducción** de direcciones lógicas a físicas se realizan en **tiempo de ejecución**, luego el programa está cargado con referencias relativas. Además, requiere apoyo **hardware**.

Es decir, durante la carga se escribe en memoria principal las direcciones lógicas o relativas y cada vez que se vaya a acceder a una de estas direcciones se le suma la dirección base.

Esto tiene de **ventaja** el ahorro de operaciones en caso de que haya direcciones que no se utilicen, aunque en la actualidad la reubicación absoluta es obsoleta ya que ahora se encarga la CPU de esto.



## 4.4 Espacios para las direcciones de memoria

### Espacio de direcciones lógico.

Conjunto de direcciones lógicas (o relativas) que utiliza un programa ejecutable.

### Espacio de direcciones físico.

Conjunto de direcciones físicas (memoria principal) correspondientes a las direcciones lógicas del programa en un instante dado. El sistema operativo contiene un mapa con la memoria, procesos, tabla de signos... Cuando se está realizando la compilación a cada nombre se le asigna una dirección de memoria, para los símbolos no resueltos por el compilador ( ejemplo funciones externas) se resuelven durante la encuadernación o enlace.

Como apuntes: la tabla de símbolos puede ser dinámica o estática, en caso de ser dinámica el encuadernador hará uso de ella.

Los datos a los que se le asigna un valor al comienzo del programa, en memoria estarían después del código, en argumentos de programa y pila.

Las pilas se van formando cada vez que se llama a una función, creando zonas de **registros de activación** que contienen los valores que van tomando las funciones.

Para buscar una variable se buscan en los registros que la han llamado, a diferencia de la variables globales.

Lo que se guarda en una pila de la CPU es la dirección de la dirección que es distinta a la pila de proceso, que está en la pila de la CPU y otra en memoria principal, en el registro de activación de la variables globales.

**Montículo o módulo:** en él se van almacenando las variables dinámicas, las listas y las pilas, al final del espacio de memoria reservado. Para evitar un desbordamiento del montículo se utiliza el **recolector de basura** compacta el espacio no utilizado en memoria.

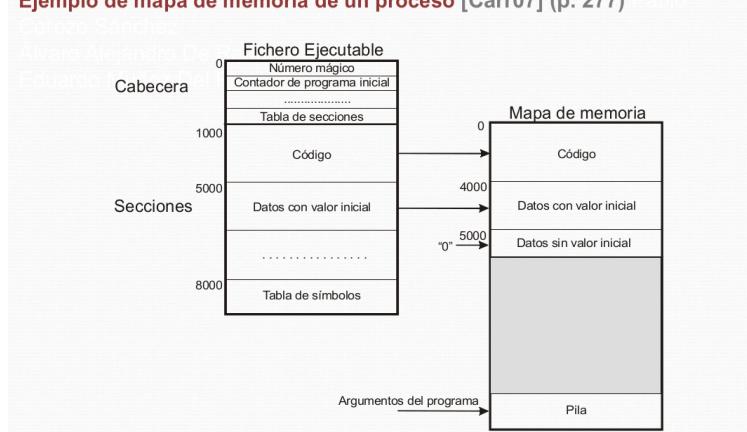
## Mapa de memoria de un ordenador.

Todo el espacio de memoria direccionable por el ordenador. Normalmente depende del tamaño del bus de direcciones.

## Mapa de memoria de un proceso.

Se almacena en una estructura de datos (que reside en memoria) donde se guarda el tamaño total del espacio de direcciones lógico y la correspondencia entre las direcciones lógicas y las físicas.

Ejemplo de mapa de memoria de un proceso [Carr07] (p. 277)



## 4.5 Problema de la fragmentación de memoria

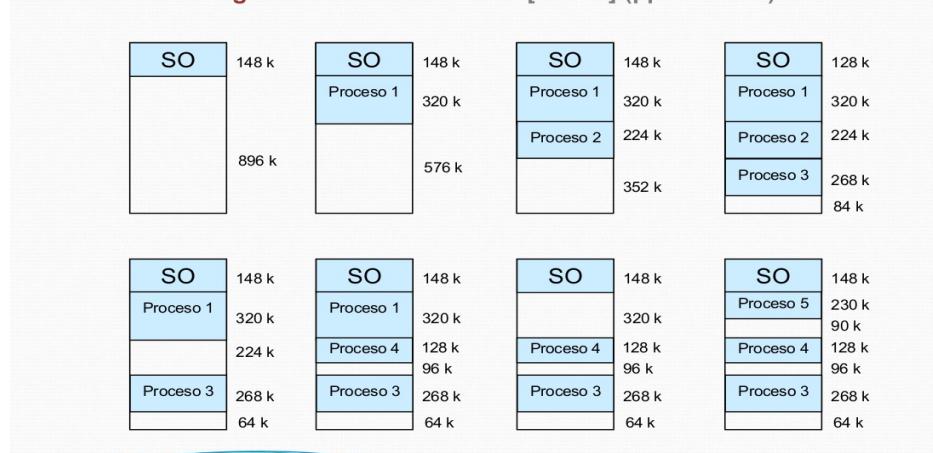
Con particiones del mismo tamaño, la ubicación de los procesos en memoria es trivial. En cuanto haya una partición disponible, un proceso se carga en dicha partición. Si todas las particiones se encuentran ocupadas por procesos que no están listos para ejecutar, entonces uno de dichos procesos debe llevarse a disco para dejar espacio para un nuevo proceso. Cuál de los procesos se lleva a disco es una decisión de **planificación**.

Con particiones de diferente tamaño, hay dos formas posibles de asignar los procesos a las particiones. La forma más sencilla consiste en asignar cada proceso a la partición más pequeña dentro de la cual cabe. Se necesita una cola de planificación para cada partición que mantenga procesos en disco destinados a dicha partición. La ventaja es que los procesos siempre se asignan de tal forma que se minimiza la memoria malgastada dentro de una partición (fragmentación interna). Pero no es óptima. Una técnica óptima sería emplear una única cola para todos los procesos. En el momento de cargar un proceso en la memoria principal, se selecciona la partición más pequeña disponible que puede albergar dicho proceso. Si todas las particiones están ocupadas, se debe llevar a cabo una decisión para enviar a *swap* a algún proceso.

El uso de particiones de distinto tamaño proporciona un grado de flexibilidad frente a las particiones fijas. Además, los esquemas de particiones fijas son relativamente sencillos y requieren un soporte mínimo por parte del SO y una sobrecarga de procesamiento mínimo. Sin embargo, tiene una serie de **desventajas**:

- El número de particiones especificadas en tiempo de generación del sistema limita el número de procesos activos (no suspendidos) del sistema.
- Debido a que los tamaños de las particiones son preestablecidos en tiempo de generación del sistema, los trabajos pequeños no utilizan el espacio de las particiones eficientemente. En un entorno donde el requisito de almacenamiento principal de todos los trabajos se conoce de antemano. Se trata de una técnica ineficiente.

**Problema de la fragmentación de memoria [Stal05] (pp. 314-316)**



## 4.6 Solución a la fragmentación de memoria

Con el particionamiento dinámico, las particiones son de longitud y número variable. Cuando se lleva un proceso a la memoria principal, se le asigna exactamente tanta memoria como requiera y no más.

Es decir, la solución es trocear el espacio lógico en unidades más pequeñas: **páginas** (elementos de longitud física), o **segmentos** (elementos de longitud variable). Los trozos no tienen por qué ubicarse consecutivamente en el espacio físico.

Los esquemas de organización del espacio lógico de direcciones y de traducción de una dirección del espacio lógico al espacio físico son:

- Paginación
- Segmentación

Por lo que, para solucionar ese problema, está el Buffer de traducción adelantada TLB, lo que se tiene en CPU, es un trocito de la tablas de página en marcos, se lee ese trocito, si aparece el marco, solo habría un acceso, si no se esta memoria se llama Caché (memoria muy rápida). La forma de acceder por hardware, intenta buscar esa página el paralelo, esta tabla tiene número pagina número de marco y bits de protección, en cache no necesita el número de páginas

### 4.6.1 Paginación

Consiste en trocear los procesos de manera homogénea, es decir, todo se parte de la misma manera: la memoria física se divide en bloques iguales, el tamaño es potencia de dos, de 512 B a 8 KB estas porciones se denominan **marcos de página**.

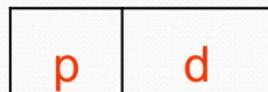
El espacio lógico de un proceso se divide conceptualmente en bloques del mismo tamaño, salvo el último que puede ser más pequeño, denominados **páginas**.

Los marcos de página contendrán páginas de los procesos.

Normalmente en la paginación por una parte va el código y por otra los datos, cuando se ejecuta el proceso los marcos libres empiezan a ocuparse, no necesariamente de manera ordenada. El estado de un marco se recoge en el **mapa de marcos libres**.

Para acceder a una dirección, se debe saber la página en que está, y dentro de la página el desplazamiento, es decir, desde el principio de su ejecución necesito saber en qué marco se está desarrollando y línea concreta.

Las **direcciones lógicas**, que son las que genera la CPU, se dividen en **número de página (p)** y **desplazamiento (d)**.



Las **direcciones físicas** se dividen en **número de marco (m**, marco donde está almacenada la página) y **desplazamiento (d)**.



Cuando la CPU genere una dirección lógica será necesario traducirla a la dirección física correspondiente, la **tabla de páginas** mantiene información necesaria para realizar dicha traducción. Existe una tabla de páginas por proceso.

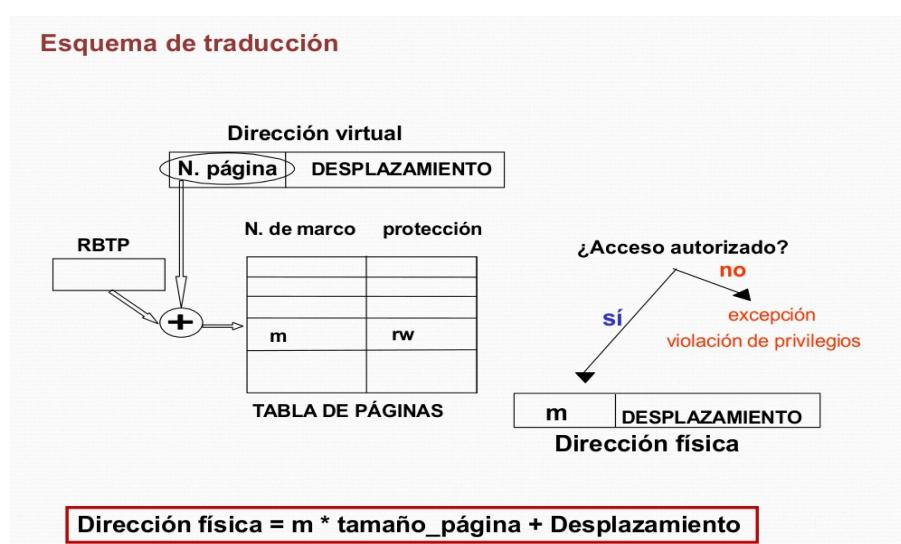
**Tabla de marcos de página**, usada por el SO y contiene información sobre cada marco de página.

#### 4.6.1.1 Contenido de la tabla de páginas

Una entrada por cada página del proceso:

- **Número de marco** en el que está almacenada la página si está en MP.
- **Modo de acceso** autorizado a la página (bits de protección).

#### 4.6.1.2 Esquema de traducción

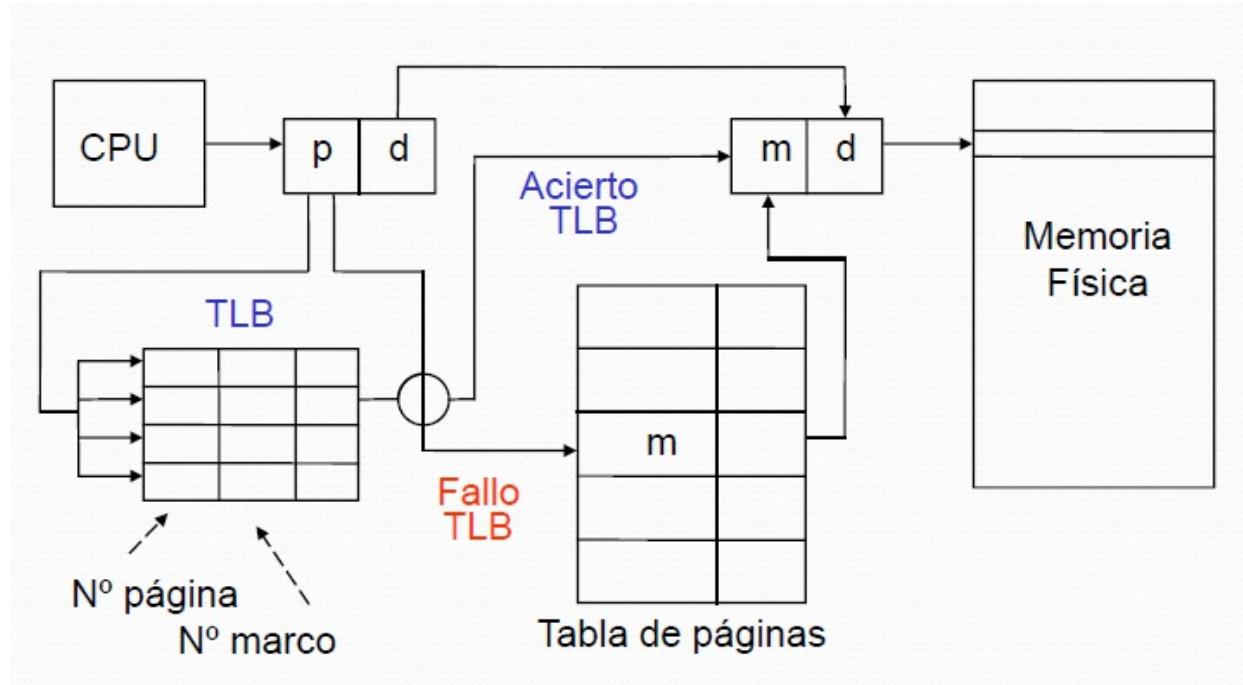


#### 4.6.1.3 Implementación de la Tabla de Páginas

- La tabla de páginas se mantiene en memoria principal.
- El *registro base de la tabla de páginas* (**RBTP**) apunta a la tabla de páginas (suele almacenarse en el PCB del proceso).
- En este esquema cada acceso a una instrucción o dato requiere dos accesos a memoria, uno a la tabla de páginas y otro a memoria.

#### 4.6.1.4 Búfer de Traducción Adelantada (TLB)

Toda referencia a la memoria virtual puede causar dos accesos a la memoria física: uno para buscar la entrada la tabla de páginas apropiada y otro para buscar los datos solicitados. De esa forma, un esquema de memoria virtual básico causaría el efecto de duplicar el tiempo de acceso a memoria. Para solventar este problema, la mayoría de esquemas de la memoria virtual utilizan un *cache* especial de alta velocidad para las entradas de la tabla de página (**TLB**, Translation Look-aside- Buffer). Esta *cache* funciona de forma similar a una memoria cache general y contiene aquellas entradas de la tabla de páginas que han sido usadas de forma más reciente. La organización del hardware de paginación resultante es:



Dada una dirección virtual, el procesador primero examina la TLB, si la entrada de la tabla de páginas solicitada está presente (acierto en TLB), entonces se recupera el número de marco y se construye la dirección real. Si la entrada de la tabla de páginas solicitada no se encuentra (fallo en la TLB), el procesador utiliza el número de página para indexar la tabla de páginas del proceso y examinar la correspondiente entrada de la tabla de páginas. Si el bit de presente está puesto a 1, entonces la página se encuentra en memoria principal, y el procesador puede recuperar el número de marco desde la entrada de la tabla de páginas para construir la dirección real. El procesador también autorizará la TLB para incluir esta nueva entrada de tabla de páginas.

Finalmente, si el bit presente no está puesto a 1, entonces la página solicitada no se encuentra en la memoria principal y se produce un fallo de acceso de memoria, llamado **fallo de página**. En este punto, abandonamos el dominio del hardware para invocar al SO, el cual cargará la página necesaria y actualizada de la tabla de páginas.

Cada entrada de la TLB debe incluir un número de página así como la entrada de la tabla de páginas completa. El procesador proporciona un hardware que permite consultar simultáneamente varias entradas para determinar si hay una conciencia sobre un número de página. Esta técnica se denomina **resolución asociativa (associative mapping)** que contrasta con la resolución directa, o indexación, utilizada para buscar en la tabla de páginas. El mecanismo de memoria virtual debe interactuar con el sistema de cache (no la cache de TLB, sino la cache de la memoria principal).

En resumen, el problema de los dos accesos a memoria se resuelve con una caché hardware de consulta rápida denominada búfer de traducción adelantada o **TLB** (*Translation Look-aside Buffer*).

El TLB se implementa como un conjunto de **registros asociativos** que permiten una búsqueda en paralelo. De esta forma, para traducir una dirección:

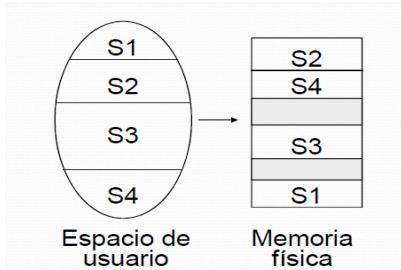
- 1 Si existe ya en el registro asociativo, obtenemos el marco.
- 2 Si no, la buscamos en la tabla de páginas y se actualiza el TLB con esta nueva entrada.

#### 4.6.1.5 Ventajas y desventajas

- **Ventaja:** se gestiona mejor la memoria.
  - **Desventajas:** los accesos a memoria, sin fragmentación son más rápidos, de la otra manera hay que hacer varios accesos, registros
1. Cpu memoria principal--> lee marco
  2. Memoria al marco 25 --> coge desplazamiento  
En total hace dos accesos a memoria.

#### 4.6.2 Segmentación

Esquema de organización de memoria que soporta mejor la visión de memoria del usuario: un programa es una colección de unidades lógicas (segmentos). Por ejemplo: procedimientos, funciones, pila, tabla de símbolos, matrices, etc.



#### 4.6.2.1 Tabla de Segmentos

Una dirección lógica es una tupla:

La **Tabla de Segmentos** aplica *direcciones bidimensionales* definidas por el usuario en direcciones físicas de una dimensión. Cada entrada de la tabla tiene los siguientes elementos (aparte de presencia, modificación y protección):

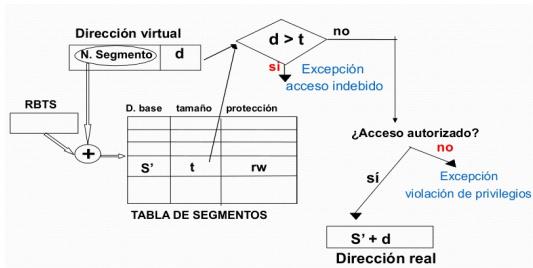
- **base**: dirección física donde reside el inicio del segmento en memoria.
- **tamaño**: longitud del segmento.

#### 4.6.2.2 Implementación de la Tabla de Segmentos

La tabla de segmentos se mantiene en memoria principal.

- El **Registro Base de la Tabla de Segmentos (RBTS)** apunta a la tabla de segmentos (suele almacenarse en el PCB del proceso).
- El **Registro Longitud de la Tabla de Segmentos (STLR)** indica el número de segmentos del proceso; el no de segmento  $s$ , generado en una dirección lógica, es legal si  $s < STLR$  (suele almacenarse en el PCB del proceso).

#### 4.6.2.3 Esquema de traducción



## 5. COMPILADORES

Ejemplo de sentencia de un programa  $a1 = b + c + 13.77$

El compilador lee de izquierda a derecha e intenta identificar cada elemento, construyendo una **tabla de símbolos**.

Signo	Datos	Dirección memoria asociada
a1	666	0xab0f20x53 (En hexadecimal)

De esta manera, cada vez que se haga referencia a a se redirecciona a la dirección asignada por el compilador.

En los primeros compiladores el número de direcciones era menor y se podía hablar de dirección de memoria absoluta. Sin embargo, en la actualidad, a alto nivel, el uso de direcciones es más complejo, abarcando incluso direcciones de direcciones, como es el caso de las sentencias de control (if, while...).

Para facilitar el trabajo se hacen asignaciones de **memoria relativa** en lo que respecta a la zona de memoria que se le otorga. Estas direcciones relativas comienzan en 0 y continúan dependiendo del tamaño en bytes del tipo de dato almacenado.

Los pasos que comprenden un programa antes de ser ejecutado son:  
compilación, encuadernación y la formación del ejecutable:

- Durante la carga del programa, se lee del disco y se copia esa información en memoria principal por el SO. El SO contiene un *mapa* con las direcciones relativas de memoria. Estas se suman a la llamada **dirección base**, para obtener las direcciones reales (cada vez que se quiera acceder a una variable durante la ejecución).

El SO también protege sobre problemas del programa. Por ejemplo, el acceso a los dispositivos lo hacen rutinas del SO, que comprueban si la dirección de memoria a la que accede el programa es correcta. En caso contrario, el SO se encargará de abortar el programa.

Volviendo a cómo se almacenan las variables, en lo concerniente a las constantes, estas se tratan exactamente igual que las simbólicas, con la única diferencia que como identificador, el nombre de la variable es el propio símbolo, el resto es lo mismo, como valor se guarda a ellos y tiene también una dirección de memoria asociada.

- **Punteros:** Es una variable que almacena una dirección de memoria. Son la base de las estructuras dinámicas de datos. El riesgo de su uso radica en la posibilidad de sobreescribir los datos de la memoria por error.
- **Arrays:** son realmente punteros encadenados en bloques de memoria. Así, es posible recorrer el array mediante aritmética de punteros, es decir, aumentando en 1 la dirección de

memoria asociada.

- Existen dos formas de pasar argumentos a una función:
  - *Por valor*: en la función se trabaja sobre una copia del valor pasado. Se puede considerar más seguro ya que un cambio en la variable dentro de la función no influye de ninguna forma al valor original.
  - *Por referencia*: a la función se le pasa la dirección de memoria de una variable. Por tanto, la función puede modificar el valor original, lo que constituye un riesgo en caso de error.

En el caso de C++ existen dos subtipos de paso de referencias: por punteros, o por referencias. El paso por punteros es heredado de C, mientras que las referencias son específicas de C++, y facilitan el paso de referencias evitando errores debidos al trabajo con punteros.

## 5.1 Resumen

El compilador conforme traduce, asigna direcciones relativas a variables y controladores de flujo. Cuando el programa se carga en memoria el SO es el encargado de gestionar el acceso del programa en memoria.

# Tema 3. Compilación y enlazado de programas

---

## 1 Lenguajes de programación

---

### 1.1 Concepto de lenguaje de programación

Un **lenguaje de programación** es un conjunto de símbolos y de reglas para combinarlos, que se usan para expresar algoritmos. Poseen un léxico (vocabulario o conjunto de símbolos permitidos), una sintaxis (indica cómo realizar construcciones del lenguaje) y una semántica (determina el significado de cada construcción correcta).

#### Lenguajes de primera generación:

- Lenguaje máquina: está escrito en binario, es lo que entiende el ordenador de forma directa.
- Ensamblador: símbolos que son traducidos al binario.

#### Lenguajes de segunda generación:

- Macro ensamblador: son secuencias de instrucciones que se empaquetaban todas juntas formándose una macro. Consta del repertorio y de las distintas macros. Se traduce de igual manera que el ensamblador. Tiene una serie de instrucciones y cierta estructura.

#### Lenguajes de tercera generación:

Los primeros lenguajes que intentaban alejarse del ensamblador eran extremadamente ineficientes en el proceso de traducción, se generaba mucho código basura que hacía que fuesen un desastre. Este paso al lenguaje de tercera generación es un desarrollo un poco mayor del macro ensamblador.

- **FORTRAN**: desarrollado por IBM fue el primer lenguaje con un compilador eficiente que hacía igual de eficaz el código máquina al código escrito.

Los lenguajes de alto nivel son lenguajes que combinan símbolos y estructura. La estructura a su vez está compuesta por datos e instrucciones. Aparecen los tipos de datos, formalizados matemáticamente.

Un tipo de dato está compuesto por una terna (G, O, P). Donde G representa del género del tipo conjunto de valores que puede tomar dentro de ese tipo (es la representación), La O que indica las operaciones que soporta el tipo. La P representa las propiedades de esas operaciones.

Los lenguajes de programación, o lenguajes de alto nivel, están específicamente diseñados para programar computadores. Sus características son:

- Son independientes de la arquitectura física del computador. Por tanto, no obligan al programador a conocer los detalles del computador que utiliza, y permiten utilizar los mismos programas en computadores diferentes, con distinto lenguaje de máquina (portabilidad).
- Una sentencia en un lenguaje de alto nivel da lugar, tras el proceso de traducción, a varias instrucciones en lenguaje máquina.

- Algo expresado en un lenguaje de alto nivel utiliza notaciones más cercanas a las habituales en el ámbito en el que se usan.

Ejemplo de como una sentencia de un lenguaje de alto nivel da lugar a varias instrucciones ensamblador y máquina:

Lenguaje de Alto Nivel	Lenguaje Ensamblador	Lenguaje Máquina
$A = B + C$	LDA 0, 4, 3	021404
	LDA 2, 3, 3	031403
	ADD 2, 0	143000
	STA 0, 5, 3	041405

La utilización de conceptos habituales suele implicar las siguientes cualidades:

- Las instrucciones se expresan por medio de **texto**, conteniendo caracteres alfanuméricos y especiales (+, =, / ...).
- Se puede asignar un **nombre simbólico** a determinados componentes del programa, es decir, se pueden definir **variables** con casi cualquier nombre. La asignación de memoria para variables y constantes las hace el traductor.
- Contiene **operadores y funciones**: aritméticas (seno, coseno...), especiales (cambiar un dato de tipo real a entero...), lógicas (comparar...), de tratamiento de caracteres (buscar una subcadena en una cadena de caracteres...), etc.
- Pueden incluirse **comentarios** en las líneas de instrucciones o específicas.

Aunque los programas escritos en lenguajes de programación no pueden ser directamente interpretados por el computador, siendo necesario realizar previamente su *traducción* a lenguaje máquina. Hay dos tipos de traductores de lenguajes de programación: compiladores e intérpretes.

Ejemplo. INT:

G = { -231, ..., 231}

O = Operaciones;

P = Propiedades

O y P conforman un grupo conmutativo.

Existen tres tipos de datos:

- **Primitivos:** no tienen estructura son los enteros, reales, caracteres etc.
- **Estructurados:** los datos estructurados necesitan tener definido de forma anterior el elemento G, por ejemplo en un array necesitamos decir en número filas y el número columnas y lo que hay

dentro.

- **Tipos de datos abstractos:** Necesita ser definido desde 0, es decir nosotros definimos la terna GOP.

Las **instrucciones** son las distintas órdenes que se le dan al ordenador desde el lenguaje de programación. Con el proceso de desarrollo de los lenguajes de programación se incorporan junto con las expresiones el control de flujo el cual regula el orden en el que se ejecutan las instrucciones.

Las **expresiones** son las operaciones como : a + b + c, d /e...

Las **sentencias** son más complejas entre ellas están las rupturas de flujo, las lecturas , escrituras, asignaciones...

Desde el punto de vista de la traducción con las estructuras de alto nivel aparecen dos tipos de sentencias:

- **Sentencias semánticamente simples:** son macros entre ellas están las asignaciones y las operaciones de E/S de una sola variable. No se tiene que expresar respecto a las demás.

- **Sentencias semánticamente compuestas:** las demás sentencias.

## 2 Construcción de traductores

---

### 2.1 Definición de traductor

**Traductor** es un programa que recibe como entrada un texto en un lenguaje de programación concreto y produce, como salida, un texto en lenguaje máquina equivalente.

Entrada -> lenguaje fuente, que define a una máquina virtual.

Salida --> lenguaje objeto, que define a una máquina real.

La forma en la que un programa escrito para una máquina virtual es posible ejecutarlo en una máquina real puede ser:

- Compilador.
- Intérprete.

### 2.2 Definición de compilador

**Compilador** traduce la especificación de entrada a lenguaje máquina incompleto y con instrucciones máquina incompletas -> necesidad de un complemento llamado enlazador.

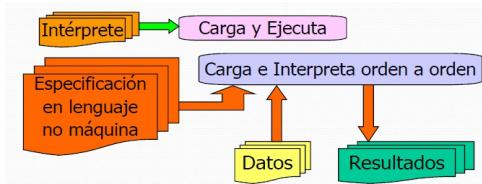
**Enlazador** (linker) realiza el enlazado de los programas completando las instrucciones máquina necesarias (añade rutinas binarias de funcionalidades no programadas directamente en el programa fuente) y generando un programa ejecutable para la máquina real.



## 2.3 Definición de intérprete

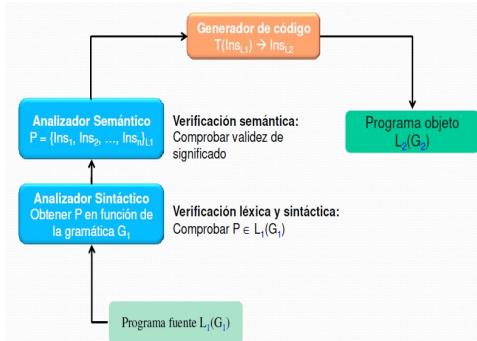
**Intérprete** lee un programa fuente escrito para una máquina virtual realiza la traducción de manera interna y ejecuta una a una las instrucciones obtenidas para la máquina real.

No se genera ningún programa objeto equivalente al descrito en el programa fuente, si no que da lugar a la ejecución inmediata del programa.



## 2.4 Esquema de traducción

Fases en la traducción de un programa fuente a un programa objeto, por un compilador:



## 2.5 Definición de gramática

La complejidad de la verificación sintáctica depende del tipo de gramática que define el lenguaje.

Una gramática definida como  $G = (\$V\_N\$, \$V\_T\$, P, S)$ , donde:

- $V_N$  es el conjunto de símbolos no terminales.
- $V_T$  es el conjunto de símbolos terminales.
- $P$  es el conjunto de producciones.
- $S$  es el símbolo inicial.

## EJEMPLO.

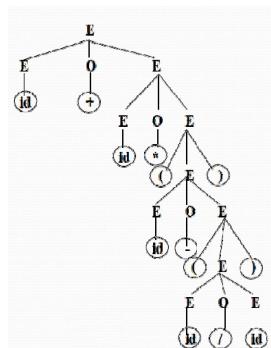
Dada la gramática siguiente:

$$\begin{aligned} P &= \{ E \rightarrow E O E \\ &\quad | (E) \\ &\quad | id \\ &\quad O \rightarrow + | - | * | / \\ &\quad \} \\ V_N &= \{E, O\} \\ V_T &= \{(,), id, +, -, *, /\} \\ S &= E \end{aligned}$$

Y el texto de entrada:

id+id\*(id-(id/id))

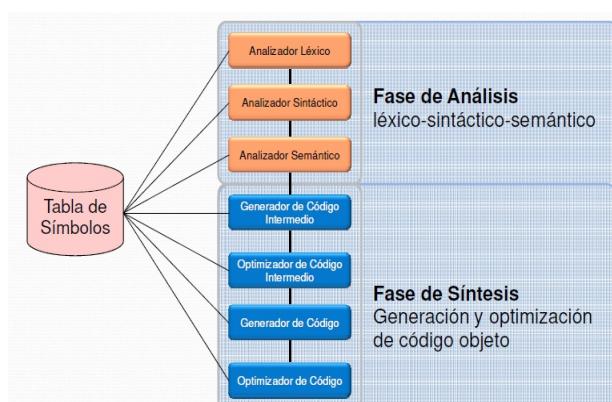
Usando las reglas de formación gramatical, se obtendría una representación que valida la construcción del texto de entrada -> verificación sintáctica correcta.



Nota: en los exámenes suele caer un ejercicio para hacer lo de las dos últimas imágenes.

## 3. Fases de traducción

### 3.1 Fases en la construcción de un traductor



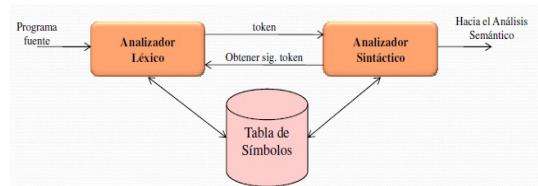
### 3.2 Análisis léxico

### 3.2.1 Función principal y conceptos

**Función:** Leer los caracteres de la entrada del programa fuente, agruparlos en lexemas (palabras) y producir como salida una secuencia de tokens para cada lexema en el programa fuente.

Conceptos que surgen del Analizador Léxico:

- **Lexema o Palabra:** Secuencia de caracteres del alfabeto con significado propio.
- **Token:** Concepto asociado a un conjunto de lexemas que, según la gramática del lenguaje fuente, tienen la misma misión sintáctica.
- **Patrón:** Descripción de la forma que pueden tomar los lexemas de un token.



El analizador léxico realiza otras funciones secundarias:

- Identifica y salta comentarios.
- Identifica y pasa espacios en blanco y tabulaciones.
- Informa de posibles errores de léxico.

El analizador léxico aísla los símbolos, identifica su tipo y almacena en las tablas de símbolos la información que pueda ser necesaria durante el proceso de traducción. Como resultado del análisis léxico se obtiene una representación del programa formada por la descripción de símbolos en las tablas, y una secuencia de símbolos junto con la referencia a la ubicación del símbolo en la tabla. Cuando abstraemos el término concreto, un lexema (pepe = 10) por ejemplo obtenemos un token.

#### EJEMPLO 1

Token	Descripción informal	Lexemas de ejemplo
IF	Caracteres 'i' y 'f'	if
ELSE	Caracteres 'e', 'l', 's' y 'e'	else
OP_COMP	Operadores <, >, <=, >=, !=, ==	<=, ==, !=, ...
IDENT	Letra seguida por letras y dígitos	pi, dato1, dato3, D3
NUMERO	Cualquier constante numérica	0, 210, 23.45, 0.899, ...

## EJEMPLO 2

S → A   C
A → id = E
C → if E then S
E → E O E   (E)   id
O → +   -   *   /
id → letra   id digito   id letra
letra → a   b   ...   z
digito → 0   1   ...   9

Token	Patrón
ID	letra(letra digito)*
ASIGN	"= "
IF	"if"
THEN	"then"
PAR_IZQ	" ("
PAR_DER	") "
OP_BIN	"+ "   "- "   "* "   "/ "

## EJEMPLO 3

Al realizar el análisis léxico de la sentencia Pascal

```
Intensidad := (Magnitud + D) * 17
```

se generará la siguiente secuencia de símbolos:

```
Identificador [1]
Asignación
(
Identificador [2]
Operador +
Identificador [3]
)
Operador *
Número [1]
```

### 3.2.2 Error léxico

En muchos lenguajes de programación, se cubren la mayoría de los siguientes tokens:

- Un token para cada palabra reservada (if, do, while, else, ...).
- Los tokens para los operadores (individuales o agrupados).
- Un token que representa a todos los identificadores tanto de variables como de subprogramas.
- Uno o más tokens que representan a las constantes (números y cadenas de literales).

- Tokens para cada signo de puntuación (paréntesis, llaves, coma, punto, punto y coma, corchetes, ...).

Esta representación contiene la misma información que el programa fuente, pero en una forma más compacta, no estando el código ya como una secuencia de caracteres, sino de símbolos.

**Error léxico:** Se producirá cuando el carácter de la entrada no tenga asociado a ninguno de los patrones disponibles en nuestra lista de tokens (ej: carácter extraño en la formación de una palabra reservada: **whi~~le~~le**).

El compilador comprueba cada símbolo según la gramática hasta que termina la expresión, se lee de izquierda a derecha hasta que se localiza un separador.

Conforme el analizador va leyendo el programa construye una tabla de símbolos con las declaraciones de variable. Esta tabla incluye el lexema, el tipo, la longitud y la dirección de memoria asociada. Lo primero que aparece en un programa es la zona de declaraciones y las bibliotecas, estas son apuntadas en la tabla de símbolos con las correspondientes direcciones asociadas.

En la parte procedural detecta si existen los distintos símbolos y traduce el identificador a la dirección de memoria donde está esa variable.

Todo lo descrito anteriormente es parte del análisis léxico. Este analizador léxico coge la gramática y analiza carácter a carácter comprobando si es correcto. Crea la tabla de símbolos que será utilizada posteriormente por el analizador sintáctico.

### 3.2.3 Especificación de los Tokens usando expresiones regulares

Se pueden usar expresiones regulares para identificar un patrón de símbolos del alfabeto como pertenecientes a un token determinado:

1. Cero o mas veces, operador \*
2. Uno o más veces, operador +:  $r^* = \$r^+\$|\$\lambda\$$
3. Cero o una vez, operador ?
4. Una forma cómoda de definir clases de caracteres es de la siguiente forma:  
 $a|b|c|\dots|z = [a-z]$

### 3.2.4 Especificación de los Tokens

Dada la gramática mostrada anteriormente, los patrones que van a definir a los tokens serían los que muestra la tabla de la derecha:

Token	Patrón
ID	letra(letra digito)*
ASIGN	=
IF	if
THEN	then
PAR_IZQ	(
PAR_DER	)
OP_BIN	++   --   **   //

$S \rightarrow A \mid C$   
 $A \rightarrow id = E$   
 $C \rightarrow if E \text{ then } S$   
 $E \rightarrow E \circ E \mid (E) \mid id$   
 $\circ \rightarrow + \mid - \mid * \mid /$   
 $id \rightarrow letra \mid id \text{ digito} \mid id \text{ letra}$   
 $letra \rightarrow a \mid b \mid \dots \mid z$   
 $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$

### 3.3 Análisis sintáctico

Un programa es sintácticamente correcto cuando sus estructuras (expresiones, sentencias declarativas, asignaciones...) aparecen en un orden correcto.

Las gramáticas ofrecen beneficios considerables tanto para los que diseñan lenguajes como para los que diseñan los traductores. Destacamos:

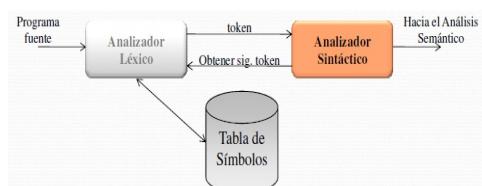
- Una gramática proporciona una especificación sintáctica precisa de un lenguaje de programación.
- A partir de ciertas clases gramaticales, es posible construir de manera automática un analizador sintáctico eficiente.
- Permite revelar ambigüedades sintácticas y puntos problemáticos en el diseño del lenguaje.
- Una gramática permite que el lenguaje pueda evolucionar o se desarrolle de forma iterativa agregando nuevas construcciones.

#### 3.3.1 Función del analizador sintáctico

**Objetivo:** Analizar las secuencias de tokens y comprobar que son correctas sintácticamente.

A partir de una secuencia de tokens, el analizador sintáctico nos devuelve:

- Si la secuencia es correcta o no sintácticamente (existe un conjunto de reglas gramaticales aplicables para poder estructurar la secuencia de tokens).
- El orden en el que hay que aplicar las producciones de la gramática para obtener la secuencia de entrada (**árbol sintáctico**).



Si no se encuentra un árbol sintáctico para una secuencia de entrada, entonces la secuencia de entrada es incorrecta sintácticamente (tiene errores sintácticos).

### 3.3.2 Gramáticas libres de contexto

Si utilizamos una variable sintáctica *instr* para denotar las instrucciones, y una variable *expr* para denotar las expresiones, la siguiente producción:

```
instr -> if (expr) instr else instr
```

especifica la estructura de esta forma de instrucción condicional.

La gramática libre de contexto consiste en terminales, no terminales, un símbolo inicial y producciones.

1. Los **terminales** son los símbolos básicas a partir de los cuales se forman las cadenas. El término *nombre de token* es un sinónimo de *terminal*. Las terminales son los primeros componentes de los tokens que produce el analizador léxico. En el ejemplo, los terminales son las palabras reservadas *if* y *else*, y los símbolos (*y*).
2. Los **no terminales** son variables sintácticas que denotan conjuntos de cadenas. En el ejemplo *instr* y *expr* son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.
3. En una gramática, un no terminal se distingue como el **símbolo inicial**, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Las producciones para el símbolo inicial se listan primero.
4. Las **producciones** de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada producción consiste en:
  - 4.1 Un no terminal, conocido como *encabezado* o *lado izquierdo* de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
  - 4.2 El símbolo  $\rightarrow$  o  $:=$ .
  - 4.3 Un *cuerpo* o *lado derecho*, que consiste en cero más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

*Ejemplo.* En esta gramática, los símbolos de los terminales son:

```
id + - * / ( )
```

Los símbolos de los no terminales son *expresión*, *term* y *factor*, y *expresión* es el símbolo inicial.

Gramática para las expresiones aritméticas simples:

*expresión* -> *expresión* + *term*

*expresión* -> *expresión* - *term*

*expresión* -> *term*

*term* -> *term* \* *factor*

*term* -> *term* / *factor*

*term* -> *factor*

*factor* -> (*expresión*)

*factor* -> *id*

## CONVENCIONES DE NOTACIÓN

### 1. Estos símbolos son **terminales**:

- 1.1 Las primeras letras minúsculas del alfabeto, como *a*, *b*, *c*.
- 1.2 Los símbolos de operadores como +, \*.
- 1.3 Los símbolos de puntuación como paréntesis, coma...
- 1.4 Los dígitos 0, 1, ..., 9.
- 1.5 Las cadenas en negrita como **\*id\*** o **if**, cada una de las cuales representa un solo símbolo terminal.

### 2. Estos símbolos son **no terminales**:

- 2.1 Las primeras letras mayúsculas del alfabeto, como *A*, *B*, *C*.
- 2.2 La letra *S* que es el símbolo inicial.
- 2.3 Los nombres en cursiva y minúsculas, como *expr* o *instr*.
- 2.4 Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para expresiones, término y factores se representan a menudo *E*, *T*, *F*, respectivamente.

## DIAPOSITIVAS:

Una gramática definida como  $G=(V_N, V_T, P, S)$ , donde:

- $V_N$  es el conjunto de símbolos no terminales.
- $V_T$  es el conjunto de símbolos terminales.

- P es el conjunto de producciones.
- S es el símbolo inicial.

se dice que es una gramática libre de contexto cuando el conjunto de producciones P obedece al formato:

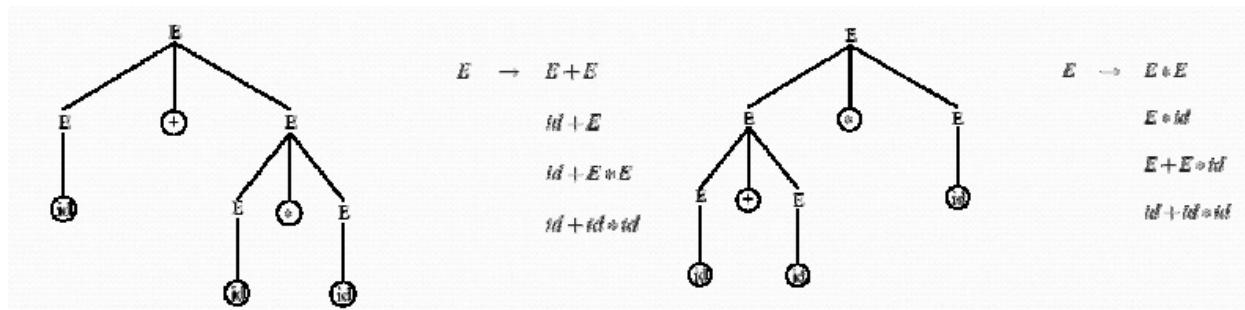
$$P = \{A \rightarrow \alpha / A \in V, \alpha \subset (V_N \cup V_T)^*\}$$

es decir, solo admiten tener un símbolo no terminal en su parte izquierda. La denominación *libre de contexto* se debe a que se puede cambiar A por \$/alfa\$, independientemente del contexto en el que aparezca A.

### 3.3.3 Gramáticas ambiguas

Una gramática es ambigua cuando admite más de un árbol sintáctico para una misma secuencia de símbolos de entrada.

**Ejemplo 3.2:** Dadas las producciones de la gramática del ejemplo 4.1 y dada la misma secuencia de entrada **id+id\*id**, se puede apreciar que le pueden corresponder dos árboles sintácticos.



Cuando programamos en un determinado lenguaje:

¿A qué nos referimos cuando hablamos de “precedencia de operadores”?

¿Por qué hay que utilizar los paréntesis para evitar la precedencia de operador?

### 3.4 Tipos de gramáticas

- **Gramáticas regulares/expresiones regulares:** Son las gramáticas más básicas que solo leen expresiones /ejemplo anterior.
- **Gramática independiente del contexto:** Así son la de la mayoría de los lenguajes de programación. Consisten en terminales, no terminales, un símbolo inicial y producciones
- **Gramática sensible al contexto:** Entre estas se incluyen las gramática de dos niveles y la

gramática con atributos.

- **Gramáticas libres.**

Todas estas gramáticas se definen igual en lo que varían es un sus reglas de producción.

Una gramática con atributos es una cuádrupla compuesta por una gramática independiente del contexto, un conjunto de atributos( son asociados a los símbolos de la gramática inicial) , un conjunto de reglas de valor y condiciones sobre los valores de los atributos.

Una gramática ambigua es aquella que admite más de un árbol sintáctico para una misma secuencia de símbolos de entrada.

Estas gramáticas de atributos especifican la semántica de los lenguajes de programación. Esta especifica la sintaxis de un lenguaje con más sensibilidad al contexto.

Ejemplo:

a|b|c|d|e|f|g ..... |z

1|2|3 .... |9

|

→ |

Genera un lenguaje en este caso los nombres de las variables.

P1a2b3c es un lexema de este lenguaje. El lexema es la unidad mínima de significado, es un símbolo terminal. Todos los caracteres que pongamos o vayamos a poner deben de estar en la gramática.

El análisis sintáctico coge los identificadores (símbolos terminales) recogidos por el análisis léxico u analiza la expresión de los diversos identificadores.

En otras palabras el sintáctico recibe el token, si es un identificador de variable es una asignación.

Las gramáticas ofrecen diferentes beneficios para los diseñadores de lenguajes y también para los traductores. Se destacan:

- Una gramática proporciona una especificación sintáctica precisa de un lenguaje de programación
- A partir de ciertas clases gramaticales es posible construir de manera automática un analizador sintáctico eficiente
- Permite revelar ambigüedades sintácticas y puntos problemáticos en el diseño de lenguajes.
- Una gramática permite que el lenguaje pueda evolucionar o se desarrolle de forma iterativa agregando nuevas construcciones.

Los lenguajes están hechos de tal manera que una vez localizado el primer token ya se sabe cual va a ser su expresión. Dependiendo del tipo de gramática se va construyendo las expresiones de un modo u otro.

- **Las gramáticas de tipo LL (left, left)** construyen de arriba a abajo y de izquierda a derecha.

Este árbol es compuesto por el análisis sintáctico.

- **Las familias de gramáticas de tipo LR(left, right)** (son más complejas y detectan más elementos) El ejemplo anterior es un ejemplo de Gramática LR. El árbol es ascendente y va de derecha a izquierda.

Cualquier sentencia analiza carácter a carácter de izquierda a derecha reconoce los identificadores y crea la tabla de símbolos.

## 3.4 Análisis semántico

Semántica de un lenguaje de programación es el significado dado a las distintas construcciones sintácticas.

El proceso de traducción es la generación de un código en lenguaje máquina con el mismo significado que el código fuente.

En los lenguajes de programación, el significado está ligado a la estructura sintáctica de las sentencias.

En el proceso de traducción, el significado de las sentencias se obtiene de la identificación sintáctica de las construcciones sintácticas y de la información almacenada en las tablas de símbolos.

*Ejemplo:* En una sentencia de asignación, según la sintaxis del lenguaje C, expresada mediante la producción siguiente:

```
sent_asignación -> IDENTIFICADOR OP_ASIG expresion PYC
```

Donde \*IDENTIFICADOR\*, \*OP\_ASIG\* y \*PYC\* son símbolos terminales (tokens) que representan a una **variable**, el *operador de asignación* "=" y al *delimitador de sentencia* ";" respectivamente, deben cumplirse las siguientes reglas semánticas:

- \*IDENTIFICADOR\* debe estar previamente declarado.
- El tipo de la \*expresión\* debe ser acorde al tipo del \*IDENTIFICADOR\*.

- Durante la fase de análisis semántico se producen errores cuando se detectan construcciones **sin un significado correcto** (p.e. variable no declarada, tipos incompatibles en una asignación, llamada a un procedimiento incorrecto o con número de argumentos incorrectos, ...).
- En lenguaje C es posible realizar asignaciones entre variables de distintos tipos, aunque algunos compiladores devuelven **warnings** o avisos de que algo puede realizarse mal a posteriori.

- Otros lenguajes impiden la asignación de datos de diferente tipo (lenguaje Pascal).

**Coacción:** Operación de dos tipos distintos en la que hay que definir la salida( de que tipo será).

**Polimorfismo:** Admite las variables de tipo, depende de la ejecución.

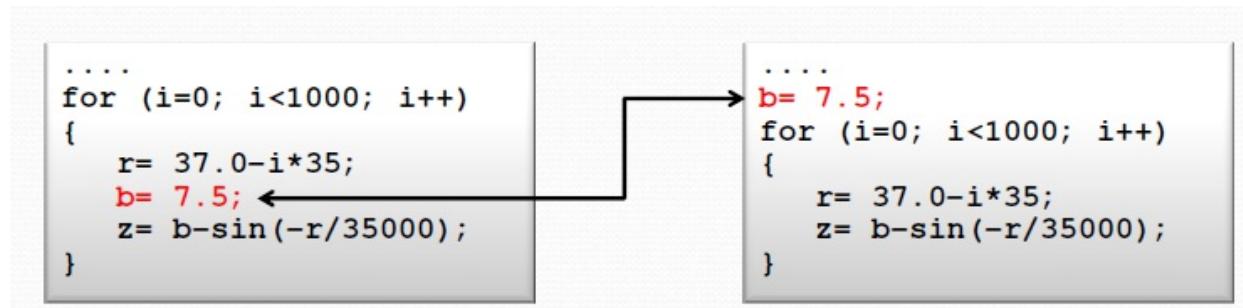
## 3.5 Generación de código

- En esta fase se genera un archivo con un código en lenguaje objeto (generalmente lenguaje máquina) con el mismo significado que el texto fuente.
- El archivo-objeto generado puede ser (dependiendo del compilador) directamente ejecutable, o necesitar otros pasos previos a la ejecución (ensamblado, encadenado y carga).
- En algunos, se intercala una fase de generación de código intermedio para proporcionar independencia de las fases de análisis con respecto al lenguaje máquina (portabilidad del compilador).
- En la generación de código intermedio se completan y consultan las tablas generadas en fases anteriores (tablas de símbolos, de constantes...). También se realiza la asignación de memoria a los datos definidos en el programa.

## 3.6 Optimización de código

- Se mejora el código mediante comprobaciones locales a un grupo de instrucciones (bloque básico) o a nivel global.
- Se pueden realizar optimizaciones de código tanto al código intermedio (si existe) como al código objeto final. Generalmente, las optimizaciones se aplican a códigos intermedios.

*Ejemplo:* Una asignación dentro de un bucle for en lenguaje C:



## 3.7 Fases de traducción

Un traductor es un programa que recibe como entrada un texto en un lenguaje de programación concreto y produce como salida un texto en lenguaje maquina equivalente. El programa inicial se denomina programa fuente y el programa obtenido programa objeto.

La traducción por un compilador consta de dos etapas fundamentales, la etapa de análisis del

programa fuente y la etapa de síntesis del programa objeto. Cada una de estas etapas conlleva la realización de varias fases como hemos visto en la parte del análisis.

El orden de construcción de un programa que entienda el ordenador es:

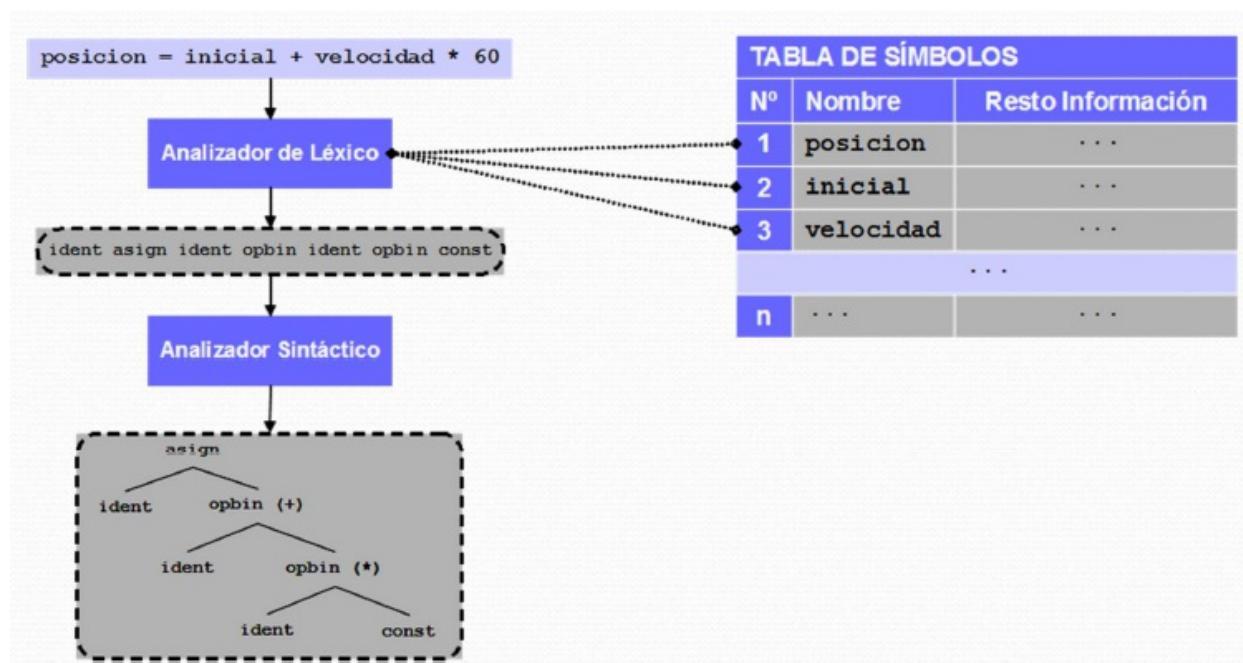
Lenguaje de programación → Lenguaje ensamblador → Lenguaje máquina

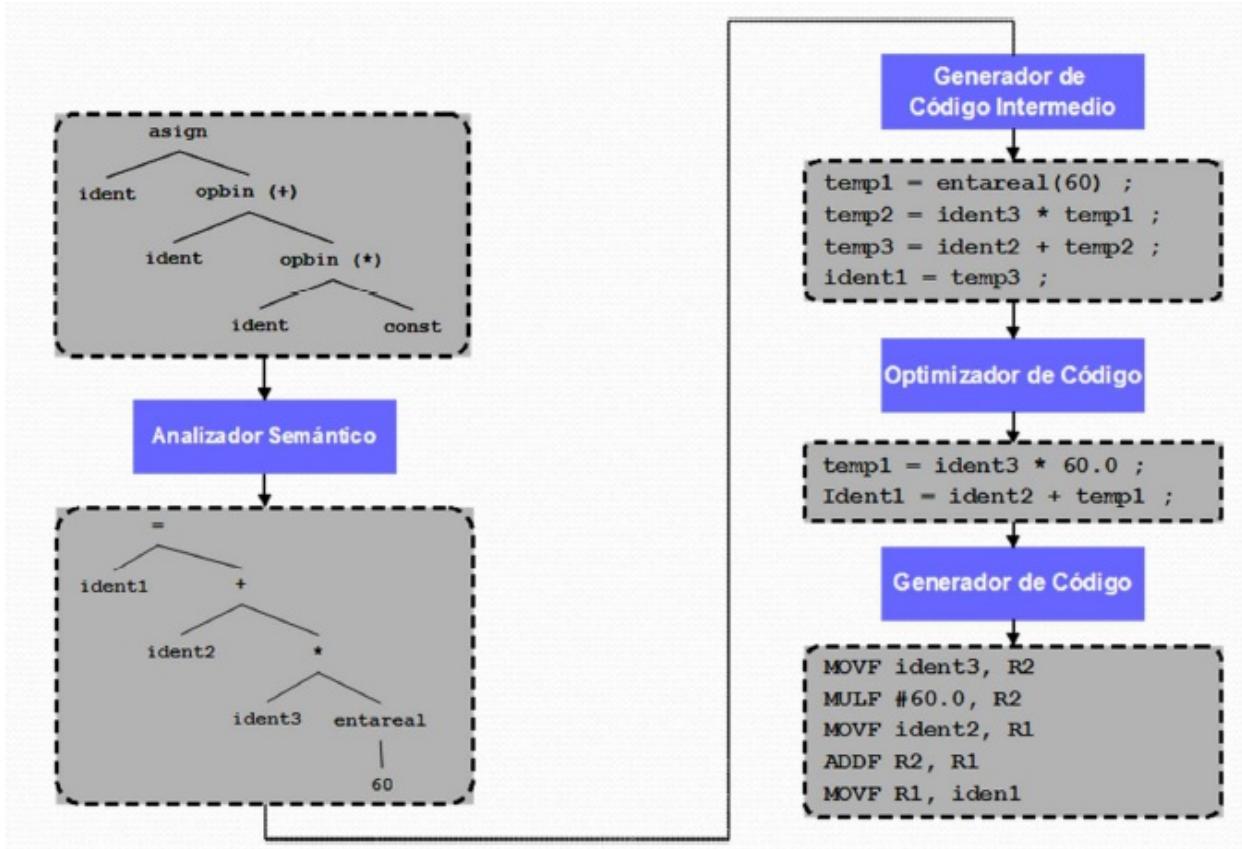
El traductor que es el programa que recibe como entrada un texto en un lenguaje y reproduce como salida un texto en lenguaje máquina equivalente. Para traducir de ensamblador a lenguaje máquina utilizamos una tabla que relacione cada símbolo con el código binario. Esta tabla contiene las direcciones de memoria, los operandos etc.

La traducción de ensamblador se basa en dos tablas, la tabla de instrucción y las tablas de símbolos.

Los traductores cruzados son aquellos que pueden efectuar la traducción a dos computadores distintos. También en un ordenador se puede simular el comportamiento de otro, a estos les llamamos emuladores.

### Ejemplo





Completar con Aho08

## 4. Intérpretes

**Intérprete:** hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. El intérprete capta una sentencia fuente, la analiza e interpreta dando lugar a su ejecución inmediata.

Consecuencias inmediatas:

- No se crea un archivo o programa objeto almacenable en memoria para posteriores ejecuciones.
- La ejecución del programa escrito en lenguaje fuente está supervisada por el intérprete.
- Ejemplo: Bash

En la práctica, el usuario crea un archivo con el programa fuente. Esto suele realizarse con un editor específico del propio intérprete del lenguaje. Según se van almacenando las instrucciones simbólicas, se analizan y se producen los mensajes de error correspondientes; así el usuario puede dar la orden de ejecución y el intérprete lo ejecuta línea a línea. Siempre el análisis antecede inmediatamente a la ejecución.

¿Cuándo es útil un intérprete?

- El programador trabaja en un entorno interactivo y se desean obtener los resultados de la ejecución de una instrucción antes de ejecutar la siguiente.
- El programador lo ejecuta escasas ocasiones y el tiempo de ejecución no es importante.
- Las instrucciones del lenguaje tienen una estructura simple y pueden ser analizadas fácilmente.
- Cada instrucción será ejecutada una sola vez.

¿Cuándo **no** es útil un intérprete?

- Si las instrucciones del lenguaje son complejas.
- Los programas van a trabajar en modo de producción y la velocidad es importante.
- Las instrucciones serán ejecutadas con frecuencia.

## 5. Modelo de memoria de un proceso

---

El SO gestiona el mapa de memoria de un proceso durante la vida del mismo. Además, el mapa inicial de un proceso está muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo.

Los programadores desarrollan sus aplicaciones utilizando lenguajes de alto nivel. Por lo que una aplicación estará compuesta por un conjunto de módulos de código fuente que deberán ser procesados para obtener el ejecutable de la aplicación. Este procesamiento consta de dos fases:

- **Compilación:** se genera el código máquina correspondiente a cada módulo fuente de la aplicación asignando direcciones a los símbolos definidos en el módulo y resolviendo las referencias a los mismos. Como resultado de esta fase se genera un módulo objeto por cada archivo fuente.
- **Montaje o enlace:** se genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos.

En resumen, los elementos responsables de la gestión de memoria son:

- Lenguaje de programación.
- Compilador.
- Enlazador.
- SO.
- MMU (Memory Management Unit).

### 5.1 Niveles de la gestión de memoria

- **Nivel de procesos** - reparto de memoria entre los procesos. Responsabilidad del SO. Se realizan operaciones vinculadas con la gestión del mapa de memoria del proceso:

- **Crear el mapa de memoria del proceso** (*crear\_mapa*): antes de comenzar la ejecución de un programa, hay que iniciar su imagen de memoria tomando como base el fichero ejecutable que lo contiene. En UNIX se trata del servicio *exec*.
- **Eliminar el mapa de memoria del proceso** (*eliminar\_mapa*): cuando termina la ejecución de un proceso, hay que liberar todos sus recursos, entre los que está su mapa de memoria. En UNIX también hay que liberar el mapa actual del proceso en el servicio *exec*, antes de crear el nuevo mapa basado en el ejecutable especificado.
- **Duplicar el mapa de memoria del proceso** (*duplicar\_mapa*): el servicio *fork* de UNIX crea un nuevo proceso cuyo mapa de memoria es un duplicado del mapa del padre.
- **Cambiar de mapa de memoria de proceso** (*cambiar\_mapa*): cuando se produce un cambio de contexto, habrá que activar de alguna forma el nuevo map y desactivar el anterior.
- **Nivel de regiones** - distribución del espacio asignado a un proceso a las regiones del mismo. Gestionado por el SO. Las operaciones están relacionadas con la gestión de las regiones:
  - **Crear una región dentro del mapa de un proceso** (*crear\_region*): será activada al crear el mapa del proceso, para crear las regiones iniciales del mismo. Además, se utilizará para ir creando las nuevas regiones que aparezcan según se va ejecutando el proceso.
  - **Eliminar una región del mapa de un proceso** (*eliminar\_region*): al terminar un proceso hay que eliminar todas sus regiones. Además, hay regiones que desaparecen durante la ejecución del proceso.
  - **Cambiar el tamaño de una región** (*redimensionar\_region*): algunas regiones, como la pila y el *heap*, tienen un tamaño dinámico que evoluciona según lo vaya requiriendo el proceso.
  - **Duplicar una región del mapa de un proceso en el mapa de otro** (*duplicar\_region*): el duplicado del mapa sociado al servicio *fork* requiere duplicar cada una de las regiones del proceso padre.
- **Nivel de zonas** - reparto de una región entre las diferentes zonas (nivel estático, dinámico basado en pila y dinámico basado en heap) de ésta. Gestión del lenguaje de programación con soporte del SO. Las operaciones identificadas solo son aplicables a las regiones que almacenan internamente múltiples zonas independientes, como *heap* o la pila.
  - **Reservar una zona** (*reservar\_zona*): en el caso de *heap* y tomando como ejemplo el lenguaje C++, se trataría del operador *new*.
  - **Liberar una zona reservada** (*liberar\_zona*): en C++ se corresponde a la operación *delete*.
  - **Cambiar el tamaño de una zona reservada** (*redimensionar\_zona*).

## 5.2 Necesidades de memoria de un proceso

- Tener un espacio lógico propio e independiente.
- Espacio protegido del resto de procesos.

- Posibilidad de compartir memoria.
- Soporte a diferentes regiones.
- Facilidades de depuración.
- Uso de un mapa amplio de memoria.
- Uso de diferentes tipos de objetos de memoria.
- Persistencia de datos.
- Desarrollo modular.
- Carga dinámica de módulos.

## 5.3 Modelo de memoria de un proceso

Estudiaremos aspectos relacionados con la gestión del mapa de memoria de un proceso, desde la generación del ejecutable a su carga en memoria:

- Nivel de región.
- Nivel de zona.

Para ello veremos:

- Implementación de tipos de objetos necesarios por un programa.
- Ciclo de vida de un programa.
- Estructura de un ejecutable.
- Bibliotecas.

## 5.4 Tipos de datos

- **Datos estáticos:** datos que existen durante toda la vida del programa. Cada dato tiene asociada una posición fija en el mapa de memoria del proceso durante toda la ejecución del programa. Tipos:
  - **Globales:** a todo el programa, a todo un módulo o locales a una función, dependiendo del ámbito de visibilidad de la variable correspondiente. Es muy importante para el compilador y montador.
  - **Constantes o variables:** las constantes no se modifican. El compilador puede hacer ciertas comprobaciones y generar un error en caso de que se intente modificar. Pero, es necesario asegurarse en tiempo de ejecución de que no se puede modificar, para lo cual habrá que asociar este tipo de dato a una región que no pueda modificarse.
  - **Con o sin valor inicial** – direccionamiento relativo: PIC. En caso de que tenga un valor asociado, habrá que asegurarse de que este esté almacenado en la memoria cuando el proceso intente acceder al mismo. Dado que el mapa inicial del proceso se construye a partir del ejecutable, habrá que almacenar ese valor en el mismo. El código PIC se trata de una región privada ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo.

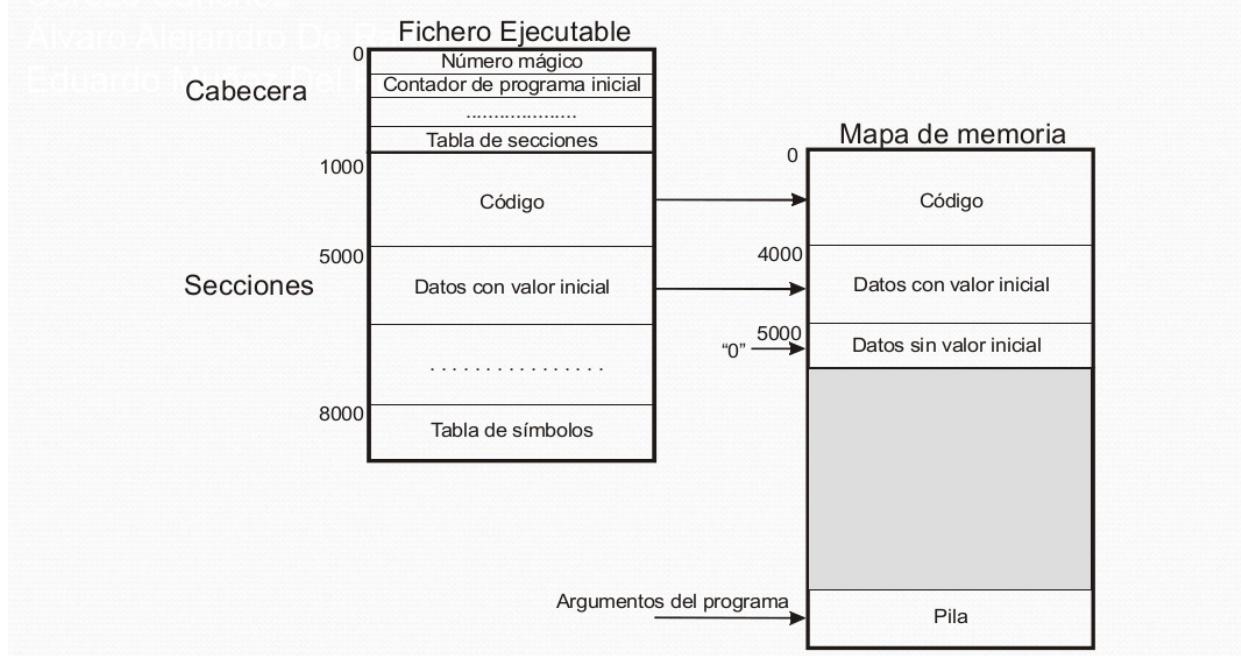
- **Datos dinámicos asociados a la ejecución de una función:** la vida de este tipo de objetos de memoria está asociada a la ejecución de una función y se corresponden con las variables locales (las no estáticas) y los parámetros de la función. Se crean cuando se invoca la función correspondiente y se destruyen cuando termina la llamada. En consecuencia, estas variables no tienen asignado espacio en el mapa inicial del proceso ni en el ejecutable. Se almacenan dinámicamente en la pila del proceso, en una estructura de datos denominada **registro de activación**, donde se guardan las variables locales, parámetros y la información necesaria para retornar al punto de llamada cuando termina la ejecución de la función.

La dirección que corresponde a una variable de este tipo se determina en tiempo de ejecución. Para acceder a esta variable, hay que consultar el último registro de activación apilado. Este tipo de variables, al igual que las estáticas, pueden tener asignado un valor inicial.

- **Datos dinámicos controlados por el programa – heap.** Se trata de datos dinámicos que el programa crea cuando considera oportuno, usando los mecanismos proporcionados por el lenguaje de programación correspondiente. Los datos se guardan en la región *heap*, donde se van almacenando todos los datos dinámicos usados por el programa. El espacio asignado se liberará cuando ya no sea necesario, o bien porque así lo indique el programa usando un mecanismo de detección automática, denominado **recolección de basura**.

La dirección asociada a un dato de este tipo solo se conoce en tiempo de ejecución, cuando el entorno de ejecución del lenguaje le asigna una zona del *heap*. Las referencias a este tipo de objetos no necesitan reubicarse y cumplen la propiedad PIC. Por tanto, el compilador y las bibliotecas del lenguaje resuelven todos los aspectos de implementación requeridos.

### Ejemplo de mapa de memoria de un proceso [Carr07] (p. 277)



## 5.5 Ejemplo de evolución de la pila (stack) en la ejecución de un programa

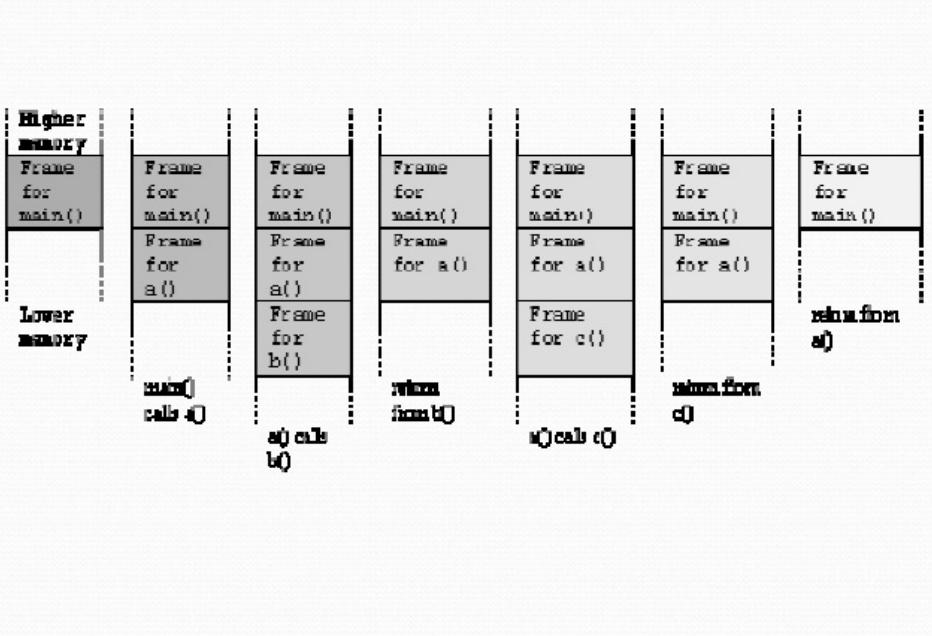
```
#include <stdio.h>
int a();
int b();
int c();

int a()
{
    b();
    c();
    return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

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

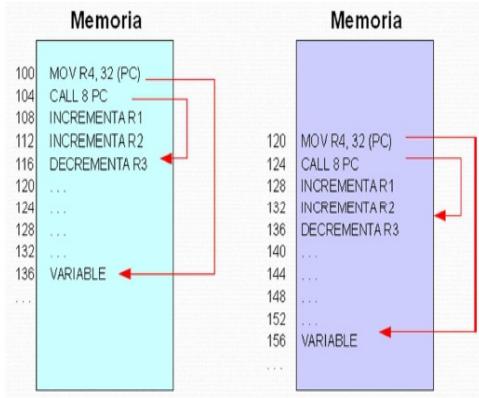


### GESTIÓN DE LA PILA

La pila es una región que está organizada internamente como zonas independientes (los diversos registros de activación apilados en un momento dado) por lo que se trata de un caso de lo que hemos denominado nivel de gestión de zonas. Las gestión de la pila es por tanto un problema general de la asignación de memoria, donde la petición de reserva se corresponde con la creación de un registro de activación al invocar una función y la de liberar está asociada a la eliminación de un registro de activación cuando concluye la ejecución de la función. Sin embargo, se trata de un caso trivial de este problema, ya que las reservas tienen un comportamiento LIFO, es decir la última zona reservada es la primera que se libera, no generándose nunca huevos entre las zonas reservadas.

## 5.6 Código independiente de la posición (PIC, Position Independent Code)

- Un fragmento de código cumple esta propiedad si puede ejecutarse en cualquier parte de la memoria.
- Es necesario que todas sus referencias a instrucciones o datos no sean absolutas sino relativas a un registro, por ejemplo, contador de programa.



## 5.7 Ejemplos de tipos de objetos de memoria

```

int a;           /* variable estática global sin valor inicial */
int b= 8;        /* variable estática global con valor inicial */
static int c;    /* variable estática de módulo sin valor inicial */
static int d= 8; /* variable estática de módulo con valor inicial */
const int e= 8;  /* constante estática global */
static const int f= 8; /* constante estática de módulo */
extern int g;   /* referencia a variable global de otro módulo */

void funcion (int h) /* parámetro: variable dinámica de función */
{
    int i;          /* variable dinámica de función sin valor inicial */
    int j= 8;        /* variable dinámica de función con valor inicial */
    static int k;   /* variable estática local sin valor inicial */
    static int l= 8; /* variable estática local con valor inicial */
    {
        int m;      /* variable dinámica de bloque sin valor inicial */
        int n= 8;    /* variable dinámica de bloque con valor inicial */
    }
    . . .
}

```

## 5.8 Programa que usa los tres tipos de objetos de memoria básicos

```

struct tipo {
    int a, b;
};

int main (int argc, char *argv[])
{
    static struct tipo var_estatica;
    struct tipo var_dinamica;
    struct tipo *var_heap= malloc (sizeof (struct tipo));

    var_estatica.b= 12;      /* 1 acceso con direccionamiento absoluto */
    var_dinamica.b= 14;      /* 1 acceso con direccionamiento relativo a SP */
    var_heap->b= 22;        /* 2 accesos con direccionamiento indirecto */

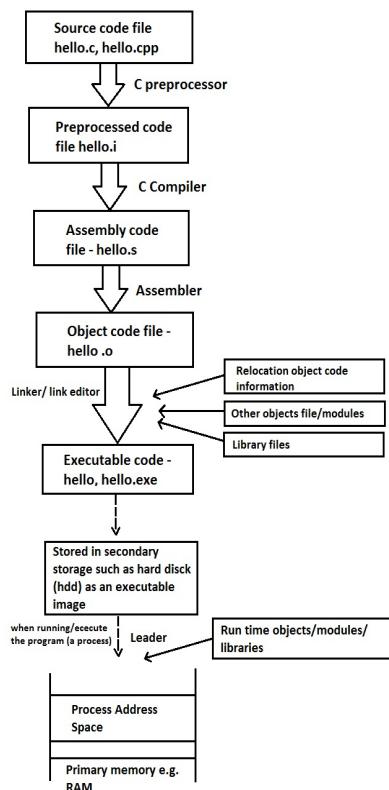
    return 0;
}

```

# 6. Ciclo de vida de un programa

A partir de un código fuente, un programa debe pasar por varias fases antes de poder ejecutarse:

1. Preprocesado
2. Compilación
3. Ensamblado
4. Enlazado
5. Carga y Ejecución



## 6.1 Ejemplo de compilación

**gcc o g++** es un wrapper (envoltorio) que invoca a:

```
$ gcc -v ejemplo.c
cpp1 ... // preprocesador
cc ... // compilador
as ... // ensamblador
collect2 ... // wrapper que invoca al enlazador ld
```

Podemos salvar los archivos temporales con

```
$ gcc -fprofile-arcs -fcoverage-mapping
```

Podemos generar el archivo ensamblador con

```
$ gcc -S
```

El archivo objeto con

```
$ gcc -c
```

Enlazar un objeto para generar el ejecutable con:

```
ld objeto.o -o eje
```

## 6.2 Compilación

El compilador procesa cada uno de los archivos de código fuente para generar el correspondiente archivo objeto.

Realiza las siguientes acciones:

- Genera **código objeto** y **calcula cuánto espacio** ocupan los diferentes tipos de datos
- Asigna **direcciones a los símbolos estáticos** (instrucciones o datos) y **resuelve las referencias** bien de forma **absoluta** o **relativa** (necesita reubicación).
- Las referencias a símbolos dinámicos se resuelven usando direccionamiento relativo a pila para datos relacionados a la invocación de una función, o con direccionamiento indirecto para el heap. No necesitan reubicación al no aparecer en el archivo objeto.
- Genera la **Tabla de símbolos e información de depuración**.

Una vez traducido el programa, su ejecución es independiente del compilador.

Compiladores dirigidos por sintaxis: El primer analizador que se ejecuta es el sintáctico. El léxico es llamado por el sintáctico para mandarle los tokens a este, cuando se tiene una expresión el léxico manda al semántico la expresión. El compilador solo genera el código máquina.

*Ejemplo*

Programa ejemplo:

```

#include <stdio.h>
int x=42;

int main()
{
    printf("Hola Mundo, x=%d\n", x);

```

Tabla de símbolos:

```

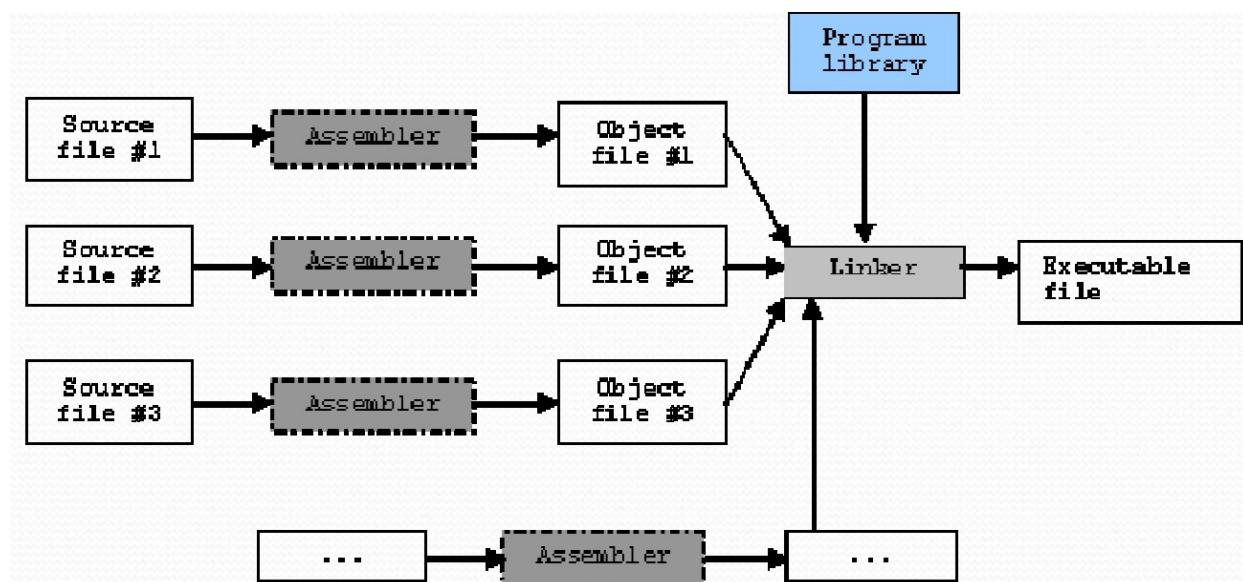
$ gcc -c hola.c
$ nm hola.o
00000000 T main
          U printf
00000000 D x

```

## 6.3 Enlazado

El **enlazador** (linker) debe agrupar los archivos objetos de la aplicación y las bibliotecas, y resolver las referencias entre ellos.

En ocasiones, debe realizar reubicaciones dependiendo del esquema de gestión de memoria utilizado.

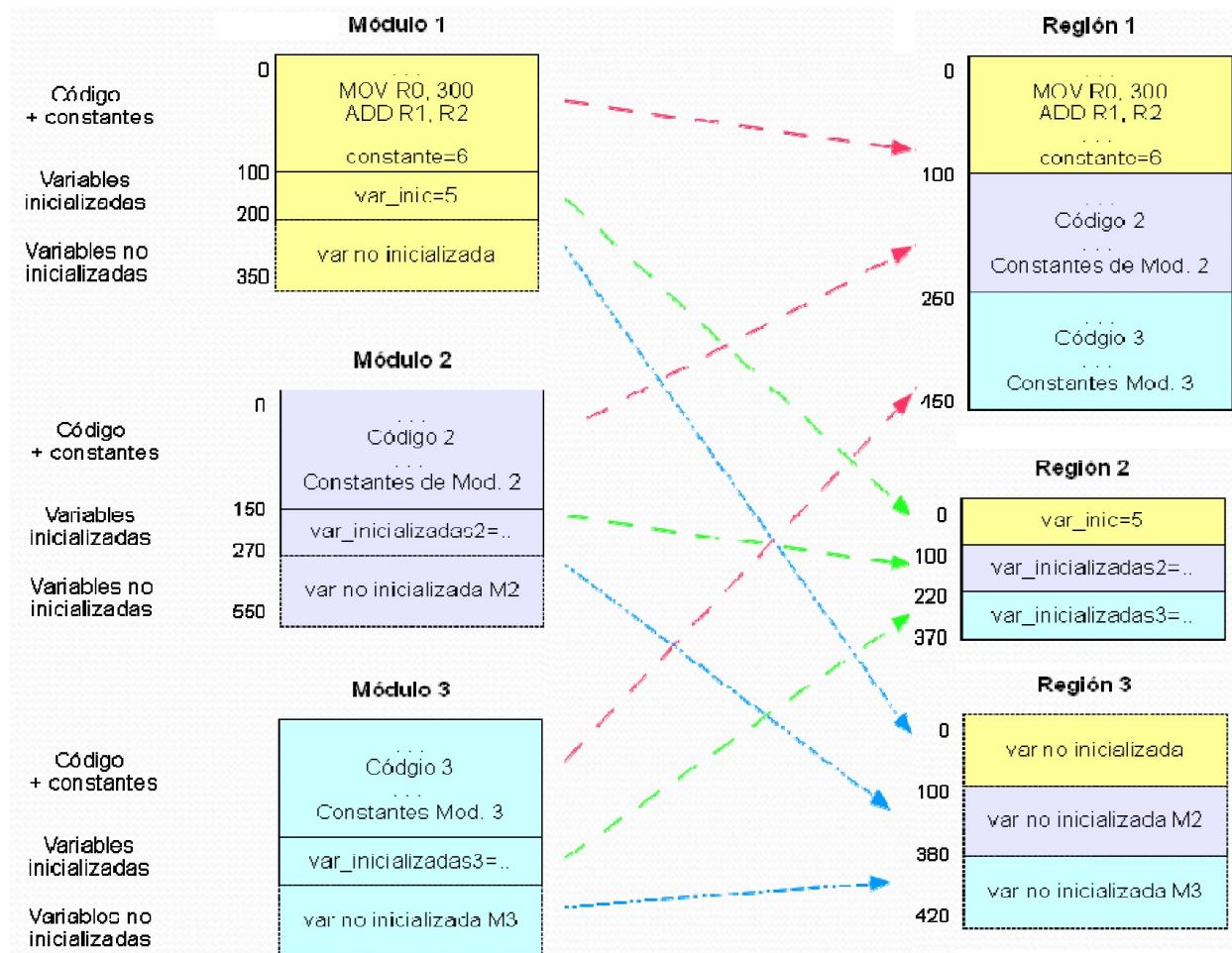


### 6.3.1 Funciones del enlazador

- Se completa la etapa de resolución de símbolos externos utilizando la tabla de símbolos.
- Se agrupan las regiones de similares características de los diferentes módulos en **regiones (código, datos inicializados o no, etc.)**

- Se realiza la **reubicación de módulos** – hay que transformar las referencias dentro de un módulo a referencias dentro de las regiones. Tras esta fase cada archivo objeto tiene una lista de reubicación que contiene los nombres de los símbolos y los desplazamientos dentro del archivo que deben aún parchearse.
- En sistemas paginados, se realiza la **reubicación de regiones**, es decir, transformar direcciones de una región en direcciones del mapa del proceso.

### 6.3.2 Agrupamiento de módulos en regiones



### 6.3.3 Tipos de enlazado y ámbito

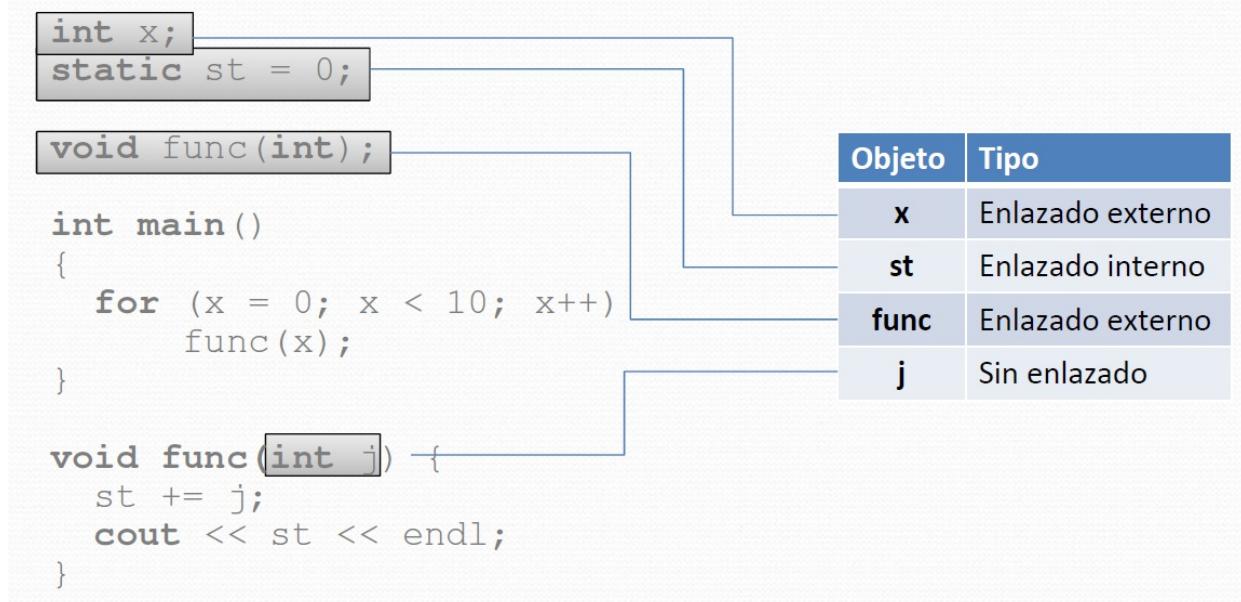
- **Atributos de enlazado:** externo, interno o sin enlazado
- Los tipos de enlazado definen una especie de **ámbito**:
  - Enlazado externo -> visibilidad global
  - Enlazado interno -> visibilidad de fichero
  - Sin enlazado -> visibilidad de bloque

### 6.3.4 Reglas de enlazado

1. Cualquier objeto/identificador que tenga ámbito global deberá tener enlazado interno si su

- declaración contiene el especificador **static**.
2. Si el mismo identificador aparece con enlazados externo e interno, dentro del mismo fichero, tendrá enlazado externo.
  3. Si en la declaración de un objeto o función aparece el especificador de tipo de almacenamiento **extern**, el identificador tiene el mismo enlazado que cualquier declaración visible del identificador con ámbito global. Si no existiera tal declaración visible, el identificador tiene enlazado externo.
  4. Si una función es declarada sin especificador de tipo de almacenamiento, su enlazado es el que correspondería si se hubiese utilizado **extern** (es decir, **extern** se supone por defecto en los prototipos de funciones).
  5. Si un objeto (que no sea una función) de ámbito global a un fichero es declarado sin especificar un tipo de almacenamiento, dicho identificador tendrá enlazado externo (ámbito de todo el programa). Como excepción, los objetos declarados **const** que no hayan sido declarados explícitamente **extern** tienen enlazado interno.
  6. Los identificadores que respondan a alguna de las condiciones que siguen tienen un atributo sin enlazado:
    - Cualquier identificador distinto de un objeto o una función (por ejemplo, un identificador **typedef**).
    - Parámetros de funciones.
    - Identificadores para objetos de ámbito de bloque, entre corchetes {}, que sean declarados sin el especificador de clase **extern**.

#### Ejemplo



## 6.4 Carga y ejecución

La reubicación del proceso se realiza en la **carga o ejecución**. Tres tipos, según el esquema de gestión de memoria:

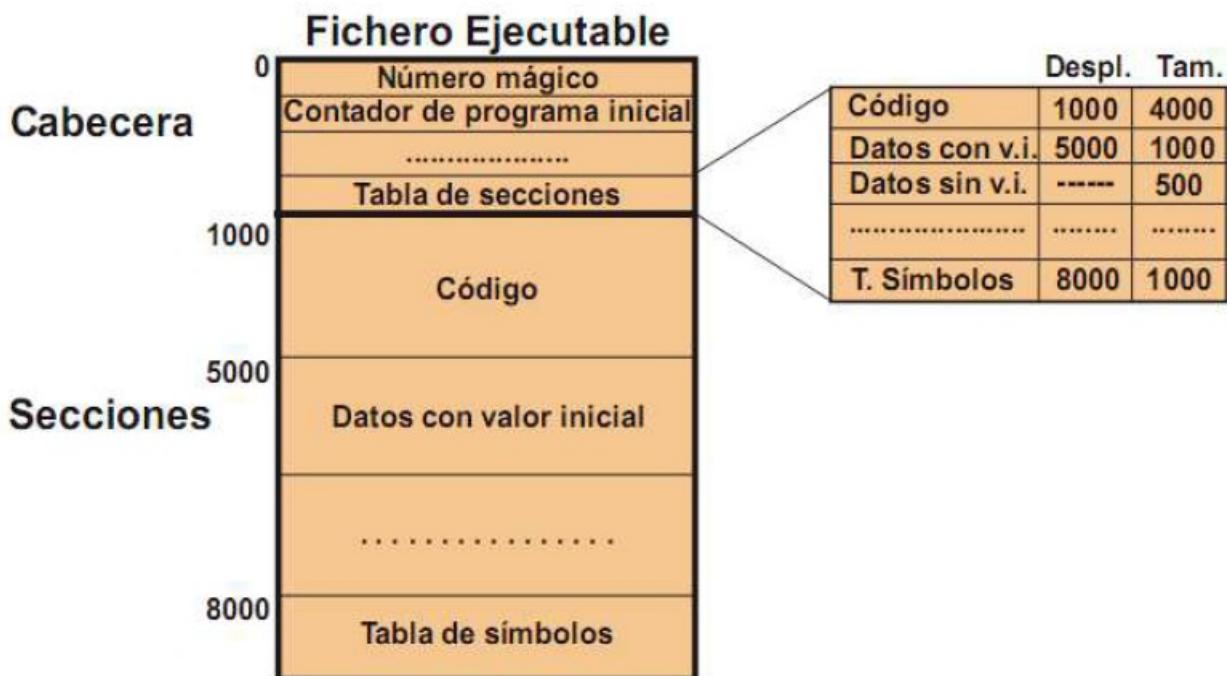
- El cargador copia el programa en memoria sin modificarlo. Es la **MMU** la encargada de realizar la reubicación en ejecución.
- En paginación, el **hardware** es capaz de reubicar los procesos en ejecución por lo que el cargador lo carga sin modificación.
- Si no usamos hardware de reubicación, ésta se realiza en la **carga**.

## 6.5 Diferencias entre archivos objeto y archivos ejecutables

Los archivos objeto (resultado de la compilación) y ejecutable (resultado del enlazado) son muy similares en cuanto a contenidos.

- Su principales diferencias son:
  - En el ejecutable la cabecera del archivo contiene el punto de inicio del mismo, es decir, la primera instrucción que se cargará en el PC.
  - En cuanto a las regiones, sólo hay información de reubicación si ésta se ha de realizar en la carga.

### 6.5.1 Formato de archivo ejecutable



### 6.5.2 Formatos de archivo objeto y ejecutables

	Descripción
a.out	Es el formato original de los sistemas Unix. Consta de tres secciones: text, data y bss que se corresponden con el código, datos inicializados y sin inicializar. No tiene información para depuración.
COFF	El <i>Common Object File Format</i> posee múltiples secciones cada una con su cabecera pero están limitadas en número. Aunque permite información de depuración, ésta es limitada. Es el formato utilizado por Windows.
ELF	<i>Executable and Linking Format</i> es similar al COFF pero elimina algunas de sus restricciones. Se utiliza en los sistemas Unix modernos, incluido GNU/Linux y Solaris.

## 6.6 Secciones de un archivo

- **.text – Instrucciones.** Compartida por todos los procesos que ejecutan el mismo binario. Permisos: r y w. Es de las regiones más afectada por la optimización realizada por parte del compilador.
- **.bss – Block Started by Symbol:** datos no inicializados y variables estáticas. El archivo objeto almacena su tamaño pero no los bytes necesarios para su contenido.
- **.data – Variables globales y estáticas inicializadas.** Permisos: r y w
- **.rdata – Constantes o cadenas literales**
- **.reloc**– Información de **reubicación** para la **carga**.
- **Tabla de símbolos – Información** necesaria (**nombre y dirección**) para localizar y reubicar **definiciones** y referencias simbólicas del programa. Cada entrada representa un **símbolo**.
- **Registros de reubicación** – información utilizada por el **enlazador** para ajustar los contenidos de las **secciones** a reubicar.

## 7. Bibliotecas

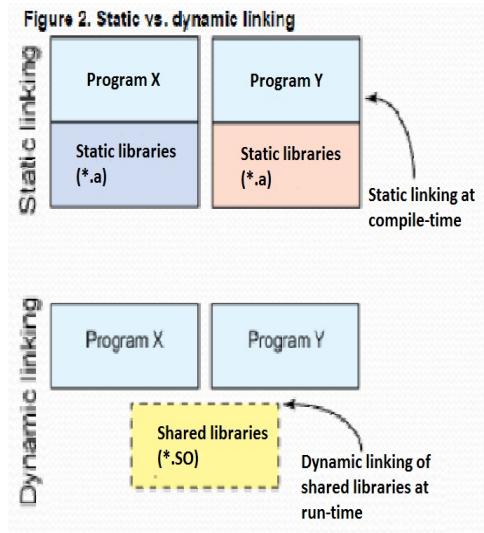
---

### 7.1 Definiciones

Una **biblioteca** es una colección de objetos normalmente relacionados entre sí.

Las bibliotecas favorecen modularidad y reusabilidad del código. Podemos clasificarlas según la forma de enlazarlas:

- **Bibliotecas estáticas:** se enlazan con el programa en la compilación (.a).
- **Bibliotecas dinámicas:** se enlazan en ejecución (.so).



## 7.2 Bibliotecas estáticas

Una biblioteca estática es básicamente un conjunto de archivos objeto que se copian en un único archivo.

Pasos para su creación:

- Construimos el código fuente:

```
double media(double a, double b)
{
    return (a+b) / 2;
}
```

- Generamos el objeto:

```
gcc -c calc_mean.c -o calc_mean.o
```

- Archivamos el objeto (creamos la biblioteca):

```
ar rcs libmean.a calc_mean.o
```

- Utilizamos la biblioteca:

```
gcc -static prueba.c -L. -lmean -o statically_linked
```

## 7.3 Bibliotecas dinámicas

Las bibliotecas estáticas tiene algunos inconvenientes:

- El archivo ejecutable puede ser bastante grande ya que incluye, además del código propio de la aplicación, todo el código de las funciones *externas* que usa el programa.
- Todo programa en el sistema que use una determinada función de biblioteca tendrá una copia del código de la misma.
- El código de la biblioteca está en todos los ejecutables que la usan, lo que desperdicia disco y memoria.
- Si actualizamos las bibliotecas, debemos recompilar el programa para que se beneficie de la nueva versión.

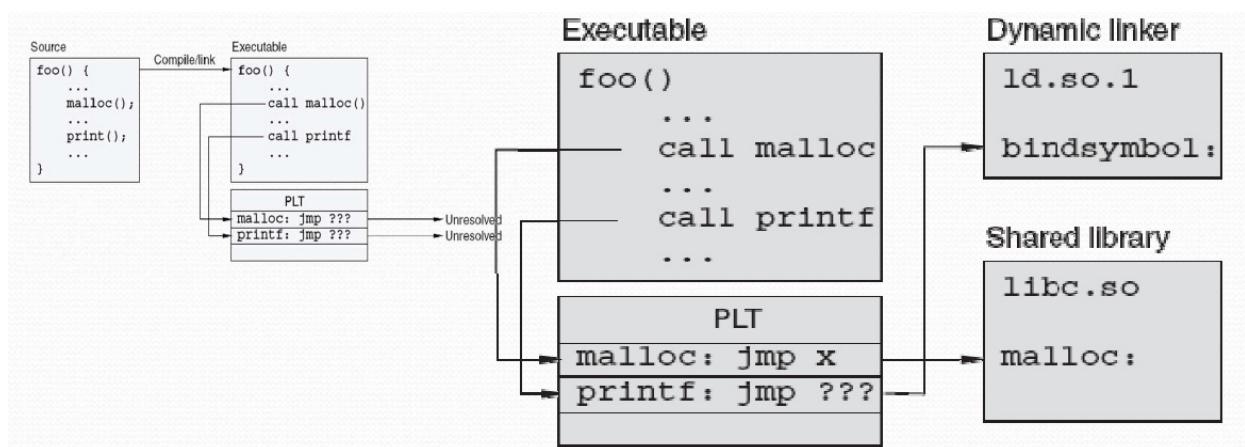
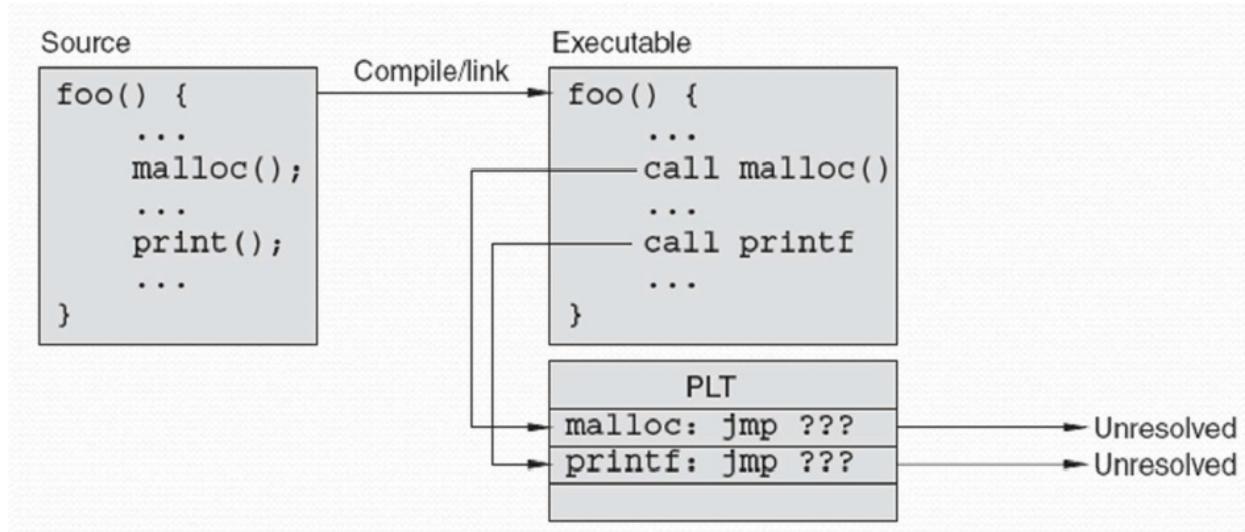
Para resolver estas deficiencias se usan las **bibliotecas dinámicamente enlazadas o bibliotecas dinámicas**. Las bibliotecas dinámicas se integran **en ejecución**, para ello se ha realizado la reubicación de módulos. Su diferencia con un ejecutable: tienen tabla de símbolos, información de reubicación y no tiene punto de entrada.

Cuando en la fase de montaje el montador procesa una biblioteca dinámica, no incluye en el ejecutable código extraído de la misma, sino que simplemente anota en el ejecutable el nombre de la biblioteca para que ésta sea cargada y enlazada en tiempo de ejecución.

Pueden ser:

- **Bibliotecas compartidas de carga dinámica** – la reubicación se realiza en tiempo de enlazado.
- **Bibliotecas compartidas enlazadas dinámicamente** – el enlazado se realiza en ejecución.

## 7.4 Estructura de un ejecutable tras el proceso de compilación y enlazado



## 7.5 Creación y uso de bibliotecas dinámicas

- Generamos el objeto de la biblioteca:

```
gcc -c -fPIC calc_mean.c -o calc_mean.o
```

- Creamos la biblioteca:

```
gcc -shared -Wl,-soname,libmean.so.1 -o libmean.so.1.0.1 calc_mean.o
```

- Usamos la biblioteca:

```
gcc main.c -o dynamically_linked -L. -lmean
```

- Podemos ver las bibliotecas enlazadas con un programa:

```
ldd hola
```

## 8. Automatización del proceso de compilación y enlazado. Herramientas y entornos

### 8.1 Automatización de la construcción de software

Automatizar la construcción es la técnica utilizada durante el ciclo de vida de desarrollo de software donde la transformación del código fuente en el ejecutable se realiza mediante un guión (script).

La automatización mejora la calidad del resultado final y permite el control de versiones.

Varias formas:

- Herramienta make
- IDE (Integrated Development Environment), que embebe los guiones y el proceso de compilación y enlazado.

# Tema 4. Sistemas de archivos. Introducción a las bases de datos

---

## 1. Concepto de archivo y directorio

---

### 1.1 Concepto de archivo

Los datos que se encuentran en **memoria masiva** suelen organizarse en archivos.

**Archivo** (o fichero) es conjunto de información sobre un mismo tema, tratada como una unidad de almacenamiento en memoria secundaria y organizada de forma estructurada para la búsqueda de un dato individual.

Un archivo está compuesto de **registros homogéneos** que contienen información sobre el tema.

Un **registro** es una estructura o unidad que forma el archivo y que contiene la información correspondiente a un elemento individual.

Un registro se puede dividir en campos. Un **campo** es un dato que forma parte de un registro y representa una información unitaria o independiente.

### 1.1.1 Operaciones sobre archivos

La vida de todo archivo comienza cuando se crea, y acaba cuando se borra. Durante la vida del archivo se suelen realizar sobre él determinadas operaciones de recuperación o consulta y de mantenimiento o actualización. Estas operaciones las realizan programas específicos, que actúan al nivel de registro.

El SO permite que el usuario pueda aludir al archivo mediante un **nombre**, independientemente de la forma en que se almacene en el dispositivo (p.e., un disco).

Todo archivo tiene asociados unos **atributos**, además del nombre, como: tamaño, fecha de creación y de modificación, propietario, permisos de acceso, etc. (dependerá del sistema operativo).

El SO permite que el usuario pueda aludir al archivo mediante un nombre, independientemente de la forma en que se almacene en el dispositivo, y suministra órdenes que realizan operaciones como:

- Crear/copiar/borrar/renombrar un archivo.
- Establecer/obtener atributos de un archivo.
- Abrir/cerrar un archivo para el procesamiento de su contenido.
- Leer/escribir un registro de un determinado archivo.

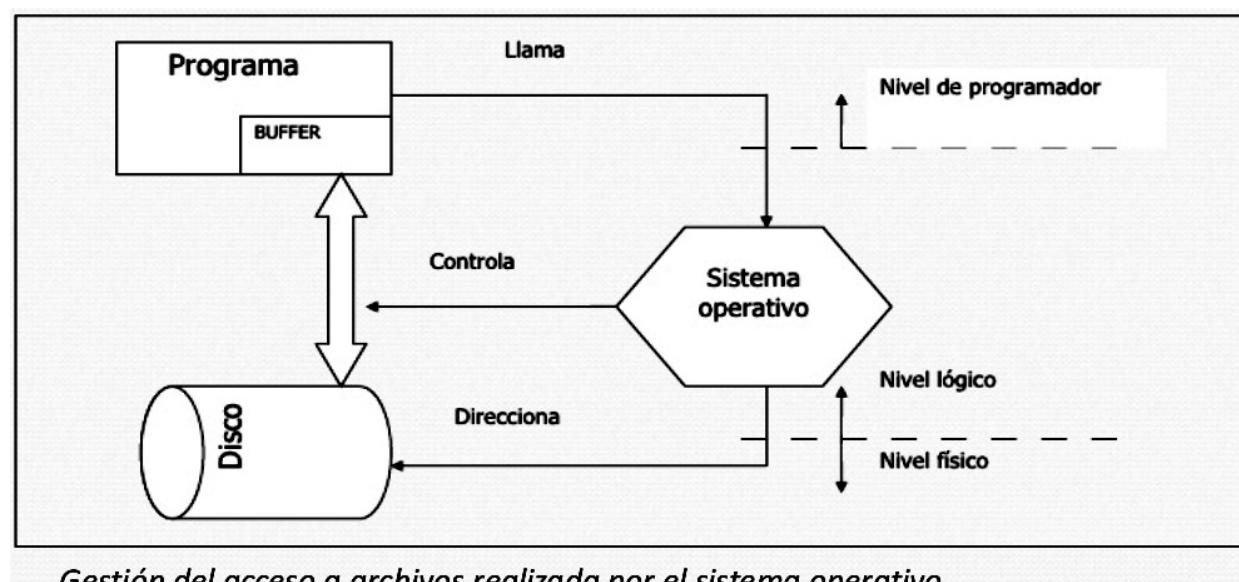
La mayor parte de las operaciones de recuperación y actualización implican la realización de una localización o búsqueda de un registro concreto para luego actuar sobre él.

Además, existen paquetes de programas específicos denominados sistemas de gestión de archivos que permiten al usuario diseñar archivos con determinadas estructuras y realizar recuperaciones y actualizaciones eficazmente.

## 1.1.2 Gestión de archivos

Cuando un usuario utiliza archivos desde un programa escrito en un lenguaje de alto nivel, el archivo no es manejado directamente por el propio programa, sino por el SO o por el software específico del computador para gestión de archivos.

Este software se encarga de efectuar los accesos necesarios al dispositivo donde se encuentra el archivo y transfiere la información solicitada del archivo al programa, o a la inversa. Esto facilita que los programas sean trasladables, pues no tienen que hacer referencia a la forma específica de gestionar la información sobre el soporte, que puede ser diferente de un sistema a otro.



Nos dice cómo interviene el SO para manipular los archivos del sistema

### 1.1.2.1 Gestión de archivos: comentarios

- El SO transporta, cada vez que se accede al dispositivo, una cantidad fija de información que depende de las características físicas de éste: **bloque o registro físico**, de longitud distinta al tamaño del registro.
- En el diseño de archivos es importante la longitud de bloque o el factor de blocale, que es el número de registros del archivo que entran en un bloque. Cuantomayor sea, menor será el número de accesos al dispositivo necesarios para procesar el archivo.
- El SO transforma la **dirección lógica** usada en los programas de usuario en la **dirección física** con la que se direcciona en el dispositivo.
  - La dirección lógica es la posición relativa que ocupa el registro en el archivo, tal y como se ve este desde el programa del usuario.
  - La dirección física es la posición real o efectiva donde se encuentra el registro en el soporte de información.
- Un archivo es una estructura de datos externa al programa que lo usa; en las operaciones de lectura/escritura se transfiere la información a/desde un buffer en memoria principal asociado a las operaciones de entrada/salida sobre el archivo.
- Los archivos se guardan o almacenan en dispositivos de memoria masiva, estando limitados en tamaño tan solo por el de los dispositivos que los albergan. Los dispositivos o soportes de memoria auxiliar, pueden ser de tipo secuencial o no direccionables o de acceso directo o direccionables.
  - En los soportes no direccionables si el último acceso se ha efectuado al registro físico  $i$ , para acceder al registro  $j > i$ , hay que leer o pasar por los registros situados entre el  $i$  y el  $j$ .
  - En los soportes direccionables, se puede leer o escribir directamente un registro físico sin más que dar su dirección física, sin necesidad de recorrer o leer otros registros.

### 1.1.3 Clasificación de archivos según el tipo de registros

1. **Longitud fija:** los registros tienen la misma longitud.
2. **Longitud variable:** los registros no tienen la misma longitud. Sin embargo, si se sabe la longitud que tiene cada registro, ya que el sistema reserva una palabra al comienzo de cada registro para anotar su longitud.
  - **Con delimitador:** un determinado carácter llamado delimitador marca el fin de un registro (suelen usarse como delimitadores el salto de línea, nulo...).
  - **Con cabecera:** cada registro contiene un campo inicial que almacena el número de bytes del registro.
3. **Longitud indefinida:** los registros no tienen la misma longitud y no sabemos la longitud que van a tener. El SO no realiza ninguna gestión sobre la longitud de los registros ya que el archivo no tiene realmente ninguna estructura interna. En cada operación de lectura/escritura se transfiere una determinada subcadena del archivo y será el programa de usuario quien indique al SO el principio y final de cada registro.

Una posibilidad añadida: disponer de un **campo clave** (o llave) que permita localizar rápidamente un registro.

Cuando una clave se utiliza como campo de localización en el archivo la denominaremos llave.

## 1.2 Tipos de archivos

- **Archivos permanentes:** contiene información relevante para una aplicación, es decir, los datos necesarios para el funcionamiento de la misma. Su vida es larga y no puede generarse de una forma inmediata a partir de otros archivos. Se pueden clasificar en:
  - **Archivos maestros.** Contiene el estado actual de los datos susceptibles de ser modificados en la aplicación. Es el núcleo central de la aplicación. Todos los procesos están orientados a actualizar el archivo maestro o a obtener resultados de él. Un ejemplo es el archivo de clientes de un banco, en el que los registros contienen información de identificación de clientes, su saldo en cuenta...
  - **Archivos constantes.**
  - **Archivos históricos.**
- **Archivos temporales:** contiene información relevante para un determinado proceso programa, pero no para el conjunto de la aplicación. Se genera a partir de los datos de archivos permanentes o para actualizar estos, y su vida es muy corta.

## 1.3 Concepto de directorio

**Directorio** es un archivo especial que permite agrupar archivos según las necesidades de los usuarios. En cada dispositivo existe una estructura jerárquica donde se localizan todos los archivos de los distintos usuarios que lo usan.

En la visión que el usuario tiene de la estructura jerárquica de archivos se utilizan los siguientes conceptos:

- Directorio actual o de trabajo.
- Directorio inicial o *home*.
- Rutas (*pathname*) absoluta y relativas.
- Lista de búsqueda.
- Enlace duro, enlace simbólico.

El SO proporciona operaciones para manejar los conceptos anteriores.

## 2. Organización de archivos

---

Se debe optar por una u otra organización atendiendo a la forma en que se va a usar el archivo. Las principales organizaciones de archivos son:

- **Secuencial.** Los registros se encuentran en cierto orden y yuxtapuestos consecutivamente.

Los registros han de ser leídos necesariamente según este orden.

- **Indexada.** Se utiliza un índice para obtener la ubicación de la zona del archivo donde se encuentra del registro buscado. Esto permite localizar un registro sin leer previamente todos los que le preceden (solo los de su zona).
- **Encadenada.** Cada registro contiene un puntero que permite localizar el siguiente registro. El archivo tiene la misma estructura que una lista lineal.
- **Directa.** La ubicación de registro en el soporte se obtiene a partir del valor de la llave (mediante un algoritmo de transformación de esta).

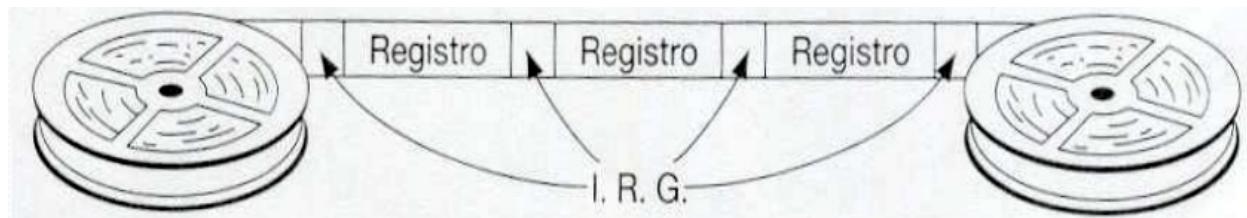
## 2.1 Organización secuencial

Los registros están almacenados físicamente de forma contigua siguiendo la secuencia lógica del archivo.

Todas las operaciones que se realizan sobre el archivo se hacen según esta secuencia. Esta es la única organización de archivos susceptible de ser gestionada en un dispositivo no direccional (soportes secuenciales tales como cinta magnética, cinta de papel o tarjeta perforada). La secuencia en que aparecen los registros en el archivo puede estar determinada por el valor de algún campo ser simplemente temporal.

organización secuencial física: (son...) se da en aquellos dispositivos que son físicamente secuenciales (p.e. CD, cinta magnética...)

Ejemplo de archivo secuencial almacenado en una cinta magnética. (IRG: inter record gap)



La organización secuencial es **adecuada**:

- Cuando se quiere leer los registros en la misma secuencia en que están almacenados.
- Cuando se necesita leer la mayoría de los registros del archivo.

### Ventajas:

- Buen aprovechamiento del espacio.
- Sencilla de utilizar.
- Se puede utilizar con dispositivos secuenciales que son de bajo precio.

## 2.1.2 Organización secuencial: operaciones sobre archivos

- **Añadir.** Solo es posible escribir al final del archivo, después del último registro escrito. Es decir, la información se graba en el archivo escribiendo los registros en secuencia, en el orden en que se desea que estén en el archivo.
- **Consulta o recuperación.** Se realiza en orden secuencial, es decir, el orden en el que se hubieran escrito determina el orden en el que se leen los registros. Para leer el registro que ocupa la posición **n** en el archivo es necesario leer previamente los **n-1** registros que hay antes que él. Por ejemplo, si queremos leer el 4º registro, tenemos que leer los registros 1, 2, 3 y 4.
- **Inserción, modificación y eliminación.** No es fácil realizar estas operaciones sobre un archivo secuencial, siendo posible que sea necesario crear otro archivo que incorpore las actualizaciones.
  - La **modificación** de un registro solo es posible si no se aumenta su longitud.
  - No es posible **eliminar** un registro del archivo, pero se puede realizar un borrado lógico, es decir, marcarlo de tal forma que al leer se identifique como no válido.
  - En otros casos es necesario crear un archivo nuevo con las actualizaciones que se quieran realizar.

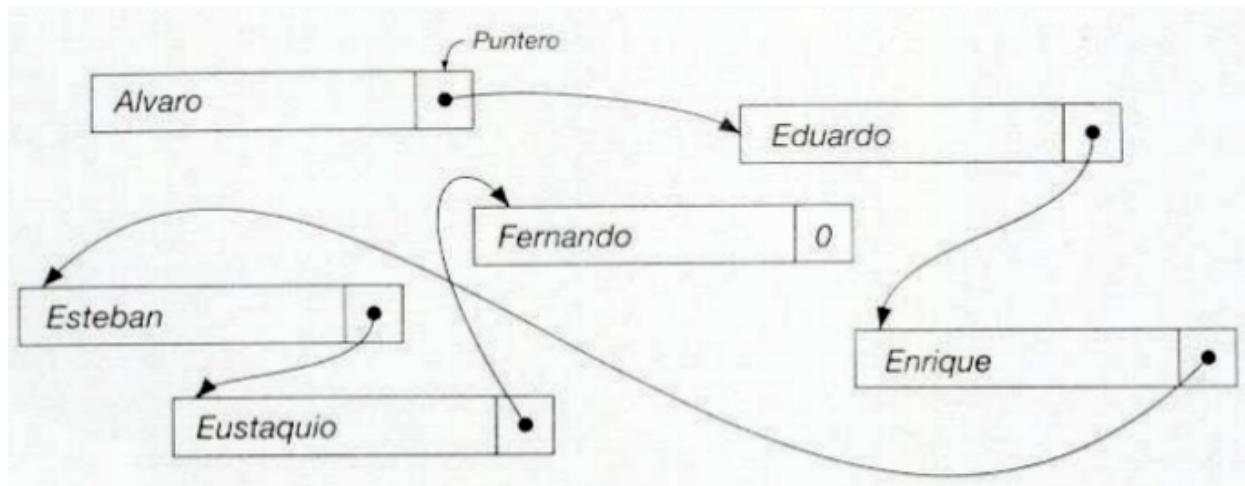
No se puede insertar los registros en el medio

## 2.2 Organización secuencial encadenada o lógica

Junto a cada registro se almacena un puntero con la dirección física del registro siguiente, dando lugar a una cadena de registros.

El último registro de la cadena contiene una marca especial en el lugar del puntero indicando que ya no hay más registros en el archivo.

Los archivos con esta organización solo pueden ser gestionados en soportes direccionables. Estructuralmente un archivo secuencial encadenado es equivalente a una lista lineal de registros.



**Principal ventaja:** facilidad de inserción y borrado de registros.

**Principal inconveniente:** limita las consultas de forma secuencial.

## 2.2.1 Organización secuencial encadenada: operaciones sobre archivos

- **Consulta o recuperación.** La consulta es secuencial, al igual que en un archivo con organización secuencial pura. Cada vez que se lee un registro se lee la posición del siguiente, lo que permite seguir la secuencia lógica del archivo.
- **Inserción.** Para insertar un registro hay que seguir los siguientes pasos:
  - i. Localizar la posición donde se desea insertar, es decir, entre qué dos registros debe estar el nuevo registro del archivo.
  - ii. Escribir el registro en una zona libre de memoria.
  - iii. Asignar al nuevo registro como puntero la dirección física del registro siguiente.
  - iv. Modificar el registro anterior para actualizar el valor de su puntero de forma que contenga la dirección del nuevo registro.
- **Borrado.** Para borrar un registro se asigna al puntero del registro anterior la dirección del registro siguiente al que se desea borrar.
- **Modificación.** Si el cambio no implica un aumento de longitud del registro, éste puede reescribirse en el mismo espacio. En caso contrario, se debe insertar el registro y luego borrar la versión anterior al cambio.

La principal **ventaja** de la organización secuencial encadenada es la facilidad de inserción y borrado de registros.

Su principal **inconveniente** es, al igual que en la organización secuencial pura, su limitación a consulta es secuencial.

## 2.2.2 Organización secuencial encadenada: ejemplo de inserción

Se inserta un registro con llave “Gato”.

Dir. Física	Llave	Puntero
1	Alba	4
2	Elefante	3
3	León	8
4	Coral	2
5	Mosca	6
6	Ñu	7
7	Pantera	0
8	Morsa	5
9	Gato	3
10		

Paso 1

Dir. Física	Llave	Puntero
1	Alba	4
2	Elefante	3
3	León	8
4	Coral	2
5	Mosca	6
6	Ñu	7
7	Pantera	0
8	Morsa	5
9	Gato	3
10		

Pasos 2 y 3

Dir. Física	Llave	Puntero
1	Alba	4
2	Elefante	9
3	León	8
4	Coral	2
5	Mosca	6
6	Ñu	7
7	Pantera	0
8	Morsa	5
9	Gato	3
10		

Paso 4

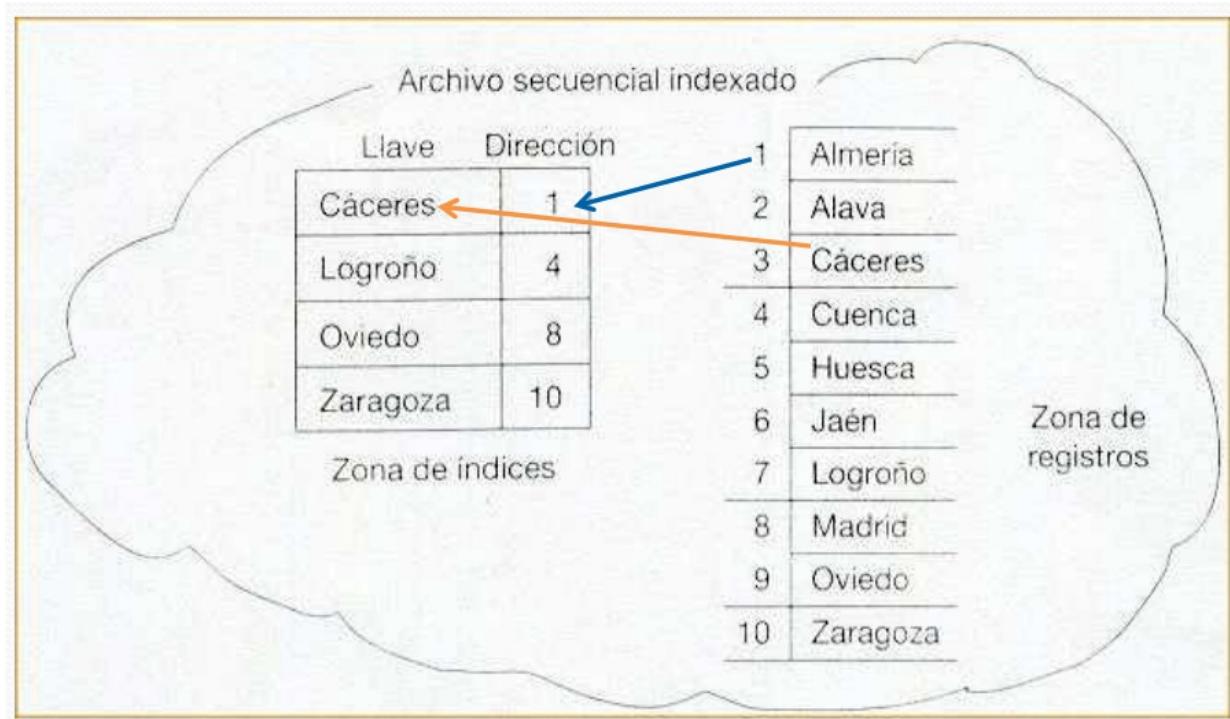
## 2.3 Organización secuencial indexada

Un archivo con organización secuencial indexada está formado por dos estructuras: zona de registros y zona de índices.

- **Zona de registros:** se encuentran los registros en sí, ordenados según el valor de una llave. Es una zona donde se direccionan los registros del archivo y está dividida en **tramos** (conjunto de registros consecutivos). Por cada tramo hay un registro en la zona de índices.
- **Zona de índices:** es una zona en la que por cada tramo de la zona de registros hay un registro que contiene:
  - El mayor valor de la **llave** del tramo (valor de llave del último registro del tramo).
  - La **dirección** del primer registro del tramo.

La gestión de la estructura la realiza el SO o un software especial, por lo que el usuario de esta estructura no necesita conocer la existencia de ambas zonas, pudiendo contemplar ambas como un todo.

**Ejemplo.**



El índice tendría 4 tramos.

P.e. Logroño está en el segundo tramo.

Para buscar Cáceres, leo Almería pero no me dice nada. Ahora tendría que leer cuenca y entonces me habría pasado.

### 2.3.1 Organización secuencial indexada: operaciones sobre archivos

- **Consulta.** Se realiza por llave (esto es, localizar un registro conocida su llave) sin necesidad de leer los registros que no se encuentran en su mismo tramo. El procedimiento a seguir para realizar una consulta por llave es:
  - i. Leer secuencialmente las llaves en la zona de índices hasta encontrar una mayor o igual a la del registro buscado.
  - ii. Obtener la dirección de comienzo del tramo donde está el registro.
  - iii. Leer secuencialmente el tramo de la zona de registros a partir de la dirección obtenida en la zona de índices hasta encontrar el registro buscado o uno con valor de llave mayor que el buscado (en este último caso el registro no se encuentra en el archivo).
- **Inserción.** Dado que ambas zonas son secuenciales, no es posible insertar un registro en archivos con esta organización. En algunos casos se permite la escritura de nuevos registros al final de la zona de registros. Estos registros, como es lógico, no podrán ser consultados por llave con el procedimiento descrito anteriormente.
- **Borrado.** Al estar los registros escritos en secuencia no es posible borrar un registro. La única forma de eliminar la información contenida en un registro es marcándolo, lo que se conoce como **borrado lógico**.

- **Modificación.** Las modificaciones son posibles tan sólo si el registro no aumenta de longitud al ser modificado y no se altera el valor de la llave del mismo.

Esta organización resulta **útil** cuando se debe combinar consultas a registros concretos y el procesamiento secuencial de todo archivo.

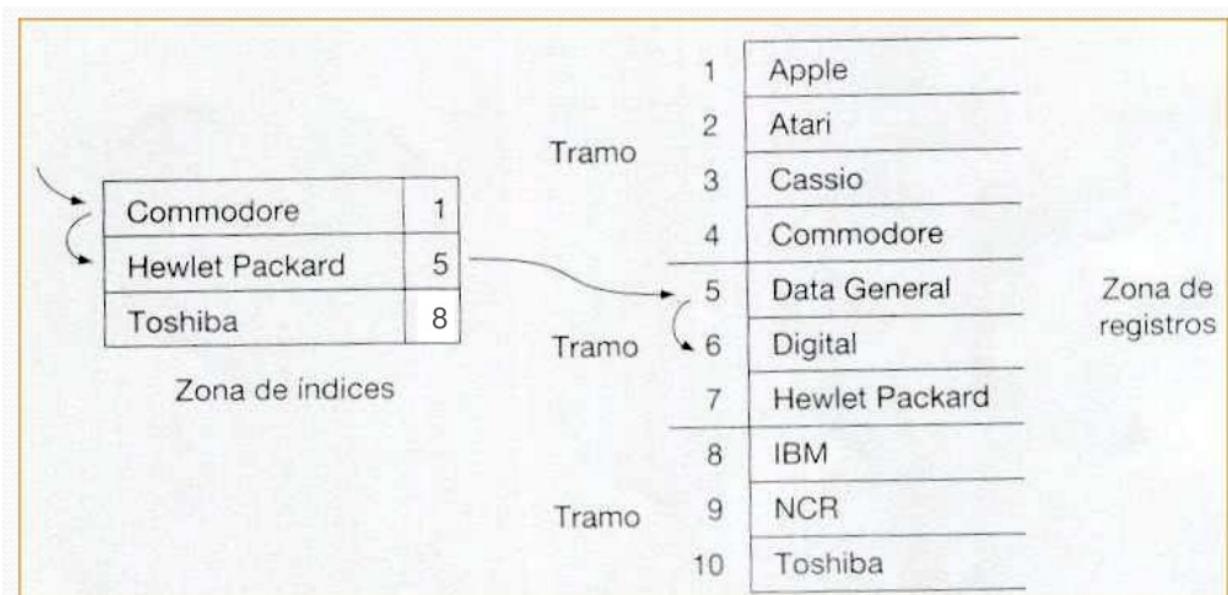
Su principal **inconveniente** es la imposibilidad de realizar actualizaciones.

Para permitir actualizaciones es necesario incluir en la estructura una zona de **desbordamiento** (zona de *overflow*). En esta zona los registros están desordenados, cada nuevo registro se escriba el final de esta. Esto complica la búsqueda por llave, pues, si el registro no es encontrado en la zona de registros, es necesario buscarlo secuencialmente en la zona de desbordamientos. Además se imposibilita la consulta secuencial del archivo, ya que los registros no aparecen ordenados por llave.

Otra mejora de la estructura anterior es incluir punteros entre los registros de forma que estos mantengan el orden lógico de los registros. A esta organización se le llama **secuencial indexada encadenada**. Cuando se crea un archivo con esta organización, su estructura es igual a la de un archivo secuencial indexado, salvo que se ha previsto un campo en cada registro para albergar un puntero.

### 2.3.2 Organización secuencial indexada: ejemplo

Consulta de un registro con llave "Digital".



## 2.4 Organización directa o aleatoria

Un archivo con **organización directa o aleatoria** (*random*) es un archivo escrito sobre un

soporte de acceso directo (o direccionable) para el cual existe una función de transformación que genera la dirección de cada registro en el archivo a partir de un campo que se usa como llave.

El nombre de “aleatorio” se debe a que normalmente no existe ninguna vinculación aparente entre el orden lógico de los registros y su orden físico. No resulta adecuado realizar una consulta secuencial en un archivo aleatorio, ya que los registros no serían leídos según el orden de la llave.

La organización directa es útil para archivos donde los accesos deben realizarse por llave, accediéndose siempre a registros concretos.

Si la información se va a procesar en conjunto, con frecuencia puede ser más rentable una organización secuencial indexada.

#### 2.4.1 Organización directa o aleatoria: problemas de los sinónimos

Un problema fundamental de esta organización es elegir adecuadamente la **función de transformación o método de direccionamiento** que se va a utilizar ya que pueden darse las siguientes situaciones no deseadas:

- Que haya direcciones que no se corresponden con ninguna llave y, por tanto, habrá zonas de disco sin utilizar.
- Que haya direcciones que se correspondan con más de una llave. En este caso se dice que las llaves son sinónimas para esa transformación o que se produce una colisión.

Hay **dos formas de resolver el problema de los sinónimos**, siempre a costa de complicar la estructura del archivo:

1. Cuando se asocia a una llave **una dirección ya ocupada** por un registro distinto (esto es, por un sinónimo de esta llave), se busca en el archivo hasta encontrar una posición libre donde escribir el registro.

La búsqueda de una posición libre puede realizarse secuencialmente a partir de la posición asignada, o aplicando a la dirección obtenida un segundo método de direccionamiento. En cualquier caso ambos son lentos y degradan el archivo.

2. Se reserva una **zona de desbordamiento** donde se escribirán los registros que no se pueden escribir en la posición que les corresponde según la transformación. Esta zona se puede gestionar secuencialmente o encadenada a la zona principal de registros.

Se puede gestionar la zona de desbordamiento secuencialmente o encadenada a la zona principal. Esto último presenta el inconveniente de tener que reservar un puntero en cada registro, pero permite un acceso más rápido a los sinónimos, lo que será más importante que el espacio.

## 2.4.2 Organización directa o aleatoria: métodos de direccionamiento

1. **Direccionamiento directo.** Se utiliza como dirección la propia llave y solo es factible cuando la llave es numérica y su rango de valores no es mayor que el rango de direcciones en el archivo. Se utiliza como dirección la propia llave.

Por ejemplo, el archivo de habitaciones de un hotel puede organizarse en forma aleatoria con direccionamiento directo haciendo coincidir la dirección con el número de habitación.

**Inconveniente:** en algunos casos pueden quedar lagunas de direcciones sin utilizar, en lugares conocidos de antemano. En este caso se pueden ocupar dichas direcciones desplazando las direcciones superiores. Un archivo aleatorio con direccionamiento directo está siempre ordenado.

2. **Direccionamiento asociado.** Se puede utilizar cualquier tipo de llave. Si se utiliza este método debe construirse una tabla en la que se almacena para cada llave la dirección donde se encuentra el registro correspondiente. Dicha tabla se debe guardar mientras exista el archivo.

Al añadir nuevos registros las llaves se colocan al final de la tabla. Esta se encontrará desordenada. Por tanto, habrá que localizar las llaves en ella por lectura secuencial, lo que ralentiza el acceso, al menos que la tabla esté residiendo en la memoria principal, se indexe, o se ordene y se busque en ella la llave con un procedimiento rápido ("búsqueda dicotómica").

3. **Direccionamiento calculado (*hashing*).** La dirección de cada registro se obtiene evaluando una expresión que utiliza como dato la llave o realizando una transformación sobre la llave. Se utiliza cuando:

- La llave no es numérica, en cuyo caso se necesita una conversión previa para obtener un número a partir de ella. Por ejemplo se usa el equivalente decimal al propio código binario del carácter (al carácter A le correspondería el 65, ...).
- La llave es numérica pero toma valores en un rango inadecuado para usarse directamente como dirección.

Con estas transformaciones aparecen sinónimos.

Métodos de cálculo de dirección:

- **División.** La dirección es el resto de dividir la llave por una constante (es el mayor de los números primos menores que el número de posiciones del archivo).
- **Extracción.** Consiste en utilizar como dirección un grupo de cifras de llave, contiguas o no.

- **Elevación al cuadrado.** Se utiliza cuando la representación numérica de la llave no es muy larga. Consiste en elevar esta al cuadrado y tomar los dígitos centrales. Así, si hay 1000 registros en el archivo y la llave es 3489: primero, se calcula  $3489^2 = 123121$ ; segundo, se toma como dirección las tres cifras centrales 3.
- **Plegamiento.** Se utiliza para llaves muy largas. Se descompone la llave en trozos de cifras contiguas del mismo tamaño y se suman estos. Sobre el número generado se puede aplicar otro método si es necesario.

Siempre que deba accederse a una posición, ya sea para introducir por primera vez un registro o para su consulta o modificación, debe aplicarse a la llave el algoritmo de transformación.

#### **2.4.3 Organización directa o aleatoria: operaciones sobre archivos**

- **Consulta.** La consulta se realiza por llave. Para leer un registro debe aplicarse a la llave el algoritmo de transformación, este algoritmo devuelve un número que es la dirección del registro que se quiere leer. Si el registro con la llave buscada no se encuentra allí, se procederá según se haya resuelto la gestión de sinónimos o colisiones.
- **Borrado.** Siempre se realiza un borrado lógico, pudiéndose reutilizar el espacio del registro eliminado.
- **Modificación e inserción.** Siempre se puede modificar o insertar un nuevo registro, realizando la transformación de la llave correspondiente.

La organización directa es **útil** para archivos donde los accesos deben realizarse por llave, accediéndose siempre a registros concretos. Si la información se va a procesar en conjunto con frecuencia puede ser más rentable una organización secuencial indexada.

### **3. Bases de datos**

---

Las bases de datos son estructuras de datos externas a los programas que están almacenados en disco.

En una aplicación convencional con archivos, estos se diseñan de acuerdo con los programas. Esto es, una vez planteado el problema, se decide si debe haber archivos, cuántos deben ser, qué organización tendrán, qué información contendrá cada uno, qué programas actuarán sobre ellos y cómo lo harán.

Esto tiene la ventaja de que los programas son bastante eficientes, ya que la estructura de un archivo está pensada "para el programa" que lo va a usar.

Sin embargo, conlleva graves inconvenientes:

- Los programas que se realizan con posterioridad a la creación de un archivo pueden ser muy

lentos, al tener que usar una organización pensada y creada "a la medida" de otro programa previo que realiza procesos diferentes.

- Si se crean nuevos archivos para los programas que se han de realizar, se puede entrar en un proceso de degeneración de la aplicación, ya que:
  - Gran parte de la información aparecerá duplicada en más de un archivo (redundancias) ocupando la aplicación más espacio del necesario.
  - Al existir la misma información en varios archivos, los procesos de actualización se complican.
  - Se corre el riesgo de tener datos incongruentes entre los distintos archivos. Por ejemplo, tener dos domicilios diferentes de un mismo individuo en dos archivos distintos (por estar uno actualizado y el otro no).

## 3.1 Problemática

En una aplicación convencional con archivos aparecen los siguientes **problemas**:

1. **Dificultad de mantenimiento.** Si hay archivos con información parcialmente duplicada, realizar las actualizaciones necesarias es un problema complejo y costoso. Normalmente, es necesario actualizar varios archivos con diferentes organizaciones. Si la actualización no se realiza correctamente se tendrá información incoherente.
2. **Redundancia.** Se dice que hay redundancia si un dato se puede deducir a partir de otros datos (se dan los problemas explicados en el caso anterior).
  - **Redundancia directa:**
  - **Redundancia indirecta:** la información que se puede obtener se duplica. Por ejemplo: la letra del DNI (porque se puede calcular).
3. **Rigidez de búsqueda.** El archivo se concibe para acceder a los datos de un modo determinado. Sin embargo, en la mayoría de los casos es necesario (o al menos deseable) combinar acceso secuencial y directo por varias claves.
4. **Dependencia con los programas.** En un archivo no están reflejadas las relaciones existentes entre campos y registros. El programa que trabaja con el archivo es quien determina en cada caso dichas relaciones. En consecuencia, cualquier modificación de la estructura de un archivo obliga a modificar todos los programas que lo usen. Esto ocurre aun en el caso de que la alteración sea ajena al programa. Así por ejemplo, si se aumenta la longitud de un campo habrá que modificar incluso los programas que no lo usan.
5. **Seguridad.** Uno de los mayores problemas de cualquier sistema de información es mantener la seguridad necesaria sobre los datos que contiene. Si se está trabajando con archivos, el control deberá realizarlo el propio programa. El aspecto de la seguridad es particularmente deficitario en los sistemas de archivos.

## 3.2 Concepto de Base de Datos

Las bases de datos surgen como alternativa a los sistemas de archivos, intentando eliminar o al

menos reducir sus inconvenientes.

Una **base de datos** es un sistema formado por un conjunto de datos y un paquete software para gestión de dicho conjunto de datos de tal modo que:

- se controla el almacenamiento de datos redundantes;
- los datos resultan independientes de los programas que los usan;
- las relaciones entre los datos se almacenan junto a ellos;
- se puede acceder a los datos de diversas formas.

En una base de datos se almacenan las relaciones entre datos junto a los datos.

La forma en la que se almacenan las relaciones entre datos y el utilizar como unidad de almacenamiento el campo además del registro, es el fundamento de la independencia respecto a los programas de aplicación.

En cualquier base de datos se puede tolerar un cierto nivel de redundancia. Se utilizan las redundancias para hacer más rápido el acceso y para asegurar la integridad de los datos. Por lo que la gestión de la información redundante es interna a la base de datos.

### 3.3 Requisitos que deben cumplir las bases de datos

1. **Acceso múltiple.** Diversos usuarios pueden acceder a la base de datos, sin que se produzcan conflictos, ni visiones incoherentes.
2. **Utilización múltiple.** Cada usuario podrá tener una imagen o visión particular de la estructura de la base de datos.
3. **Flexibilidad.** Se podrán usar distintos métodos de acceso, con tiempos de respuesta razonablemente pequeños.
4. **Seguridad.** Se controlará el acceso a los datos (a nivel de campo), impidiéndoselo a los usuarios no autorizados.
5. **Protección contra fallos.** Deben existir mecanismos concretos de recuperación en caso de fallo de la computadora.
6. **Independencia física.** Se puede cambiar el soporte físico de la base de datos sin que esto repercuta en la base de datos ni en los programas que la utilizan.
7. **Independencia lógica.** Se pueden modificar los datos contenidos en la base, las relaciones existentes entre ellos o incluir nuevos datos, sin afectar a los programas que la usan.
8. **Redundancia controlada.** Los datos se almacenan una sola vez. (¿La redundancia es indirecta???)
9. **Interfaz de alto nivel.** Existe una forma sencilla y cómoda de utilizar la base al menos desde un lenguaje de programación de alto nivel.
10. **Interrogación directa (“query”).** Existe una utilidad que permite el acceso a los datos de forma interactiva o conversacional.

## 3.4 Estructura de una base de datos

En una base de datos se almacena información de una serie de objetos o elementos. Estos objetos reciben el nombre de **entidades**, siendo una entidad cualquier ente sobre el que se almacena información.

De cada entidad se almacenan una serie de datos que se denominan **atributos** de la entidad. Puede ser atributo de una entidad cualquier característica o propiedad de esta.

Las entidades y los atributos son conceptos abstractos. En una base de datos la información de cada entidad se almacena en **registros**, y cada atributo en **campos** de dicho registro, de forma análoga al almacenamiento en archivos. Sin embargo, en una base de datos **hay diferentes tipos de registros**, uno por cada entidad.

Normalmente se reserva el nombre "registro" para especificar un "tipo de registro", utilizándose otros nombres para especificar cada una de las apariciones de ese registro en la base de datos, tales como: **elemento, valor actual de registro, u ocurrencia de registro**.

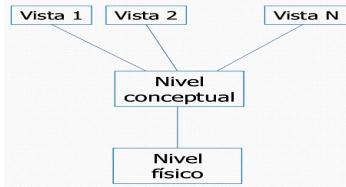
Se dice que uno o más atributos de una entidad es un **identificador o clave primaria** si el valor de dichos atributos determina de forma única cada uno de los elementos de dicha entidad, y no existe ningún subconjunto de él que permita identificar a la entidad de manera única.

En una base de datos se almacenan además de las entidades, las **relaciones** existentes entre ellas.

En la implementación de la base de datos, estas relaciones se almacenan con punteros que inserta automáticamente el software que la maneja y esto es "transparente" al usuario.

## 3.5 Niveles de abstracción de la información: vistas y esquemas

- **Nivel de vista:** permite describir diferentes **vistas** o **subesquemas**, cada una de las cuales se corresponde con la parte de la base de datos que interesa a un determinado grupo de usuarios. Además limita el acceso solo a la información de la vista.
- **Nivel conceptual.** Describe el **esquema de la base de datos**. En éste se especifica qué información se guarda en la base de datos, incluyendo todos los datos almacenados en ella y las relaciones entre ellos. Este nivel se utiliza en la administración de la base de datos.
- **Nivel físico.** Describe cómo se almacenan los datos, con las estructuras de datos necesarias para ello.



## 3.6 Modelos de datos

**Modelo de datos:** grupo de herramientas conceptuales que permite describir los datos, sus relaciones, su semántica y sus limitaciones. Ayuda a describir la estructura de una base de datos.

Clasificación de los modelos de datos:

- **Modelos lógicos basados en objetos.** Describen los datos a nivel conceptual y a nivel de vista, permiten una estructuración flexible y especificar limitaciones de los datos. Caso a tratar: Modelo entidad-relación.
- **Modelos lógicos basados en registros.** Describen los datos a nivel conceptual y a nivel de vista, permitiendo especificar la estructura lógica pero no las limitaciones de los datos. Casos a tratar: Modelo jerárquico, Modelo en red y Modelo relacional.
- **Modelos físicos de los datos.** Describen los datos en el nivel de implementación de los sistemas de base de datos.

## 3.7 Modelo entidad-relación

### 3.7.1 Conceptos básicos

- **Entidad:** objeto que tiene existencia propia, que puede distinguirse de otros y del cual se quiere almacenar información de ciertas características. Ejemplo: Pepe con DNI 24324450 y que vive en Granada.
- **Conjunto de entidades:** grupo de entidades del mismo tipo que representa la estructura genérica de una entidad de interés. Las entidades pueden pertenecer a más de un conjunto de entidades. Ejemplo: Cliente.
- **Relación:** asociación entre varias entidades. Ejemplo: El cliente con DNI 24324450 compra un coche con matrícula 6670 BBC el 7/11/2005.
- **Conjunto de relaciones:** grupo de relaciones del mismo tipo que representa la estructura genérica de las relaciones entre conjuntos de entidades. Ejemplo: Compra.
- **Grado de una relación:** número de tipos de entidad que intervienen en un tipo de relación. Ejemplo: En el caso de la relación en la que un cliente compra un coche, el grado de la relación es 2.
- **Atributo:** unidad básica de información sobre un tipo de entidad o un tipo de relación. Ejemplos: DNI, Nombre, Dirección, Fecha de compra.

### 3.7.2 Tipos de correspondencias (cardinalidad)

**Cardinalidad:** expresa el número de entidades con las que puede asociarse o corresponderse una determinada entidad mediante una relación.

En el caso de un conjunto binario de relaciones R entre los conjuntos de relaciones A y B, los tipos de correspondencias o cardinalidades pueden ser:

- **Uno a uno (1:1):** a cada entidad de A le puede corresponder una única entidad de B y viceversa.
- **Uno a muchos (1:N):** a cada entidad de A le puede corresponder más de una entidad de B, pero cada entidad de B solo puede asociarse con una única entidad de A.
- **Muchos a uno (N:1):** a cada entidad de A le puede corresponder una única entidad de B, pero cada entidad de B puede asociarse con varias entidades de A.
- **Muchos a muchos (N:M):** a cada entidad de A le puede corresponder más de una entidad de B, y a cada entidad de B le puede corresponder más de una entidad de A.

#### Ejemplo.

alumno <-> expediente ----- 1 a 1

alumno <->> libro ----- 1 a N

profesor <<->>alumno ----- N a N

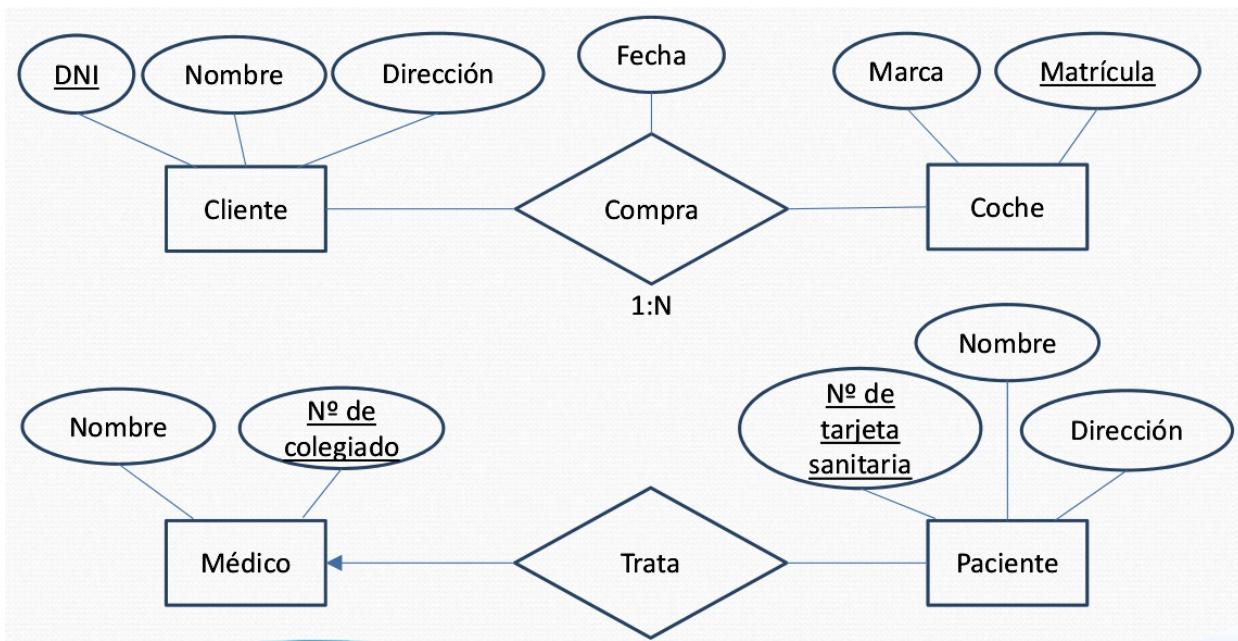
### 3.7.3 Diagrama entidad-relación

**Diagrama entidad-relación:** representación gráfica de la estructura de una base de datos organizada según el Modelo entidad-relación.

Sus componentes principales son:

- **Rectángulos:** representan conjuntos de entidades.
- **Rombos:** representan conjuntos de relaciones.
- **Elipses:** representan atributos (de conjuntos de entidades o de conjuntos de relaciones).
- **Representación de la cardinalidad.** Se puede hacer de varias maneras:
  - Con una etiqueta asociada al conjunto de relaciones (solo para relaciones binarias).
  - Poniendo una punta de flecha que señale hacia el conjunto de relaciones que participa de forma “uno” y usando una línea sin punta en el caso de conjuntos de entidades que participan de forma “muchos”.

#### Ejemplo.



El subrayado significa que es una llave.

## 3.8 Modelo lógicos basados en registros

### 3.8.1 Tipos de bases de datos

1. **Modelo de datos jerárquico.** Permite especificar una **base de datos jerárquica**, donde se establece una relación jerárquica entre los datos en forma de árbol, y no es posible definir relaciones muchos a muchos.



2. **Modelo de datos en red.** Permite especificar una **base de datos en red**, donde pueden darse relaciones binarias con cualquier cardinalidad (1:1, 1:N, N:1, N:N) y no es necesario que la estructura tenga forma de árbol.
3. **Modelo de datos relacional.** Permite especificar una **base de datos relacional**, que estará formada por tablas. Una **tabla** es una estructura bidimensional formada por una sucesión de registros del mismo tipo. Si se imponen ciertas condiciones a las tablas, se

pueden tratar como **relaciones** matemáticas. Las tablas deben cumplir las siguientes condiciones:

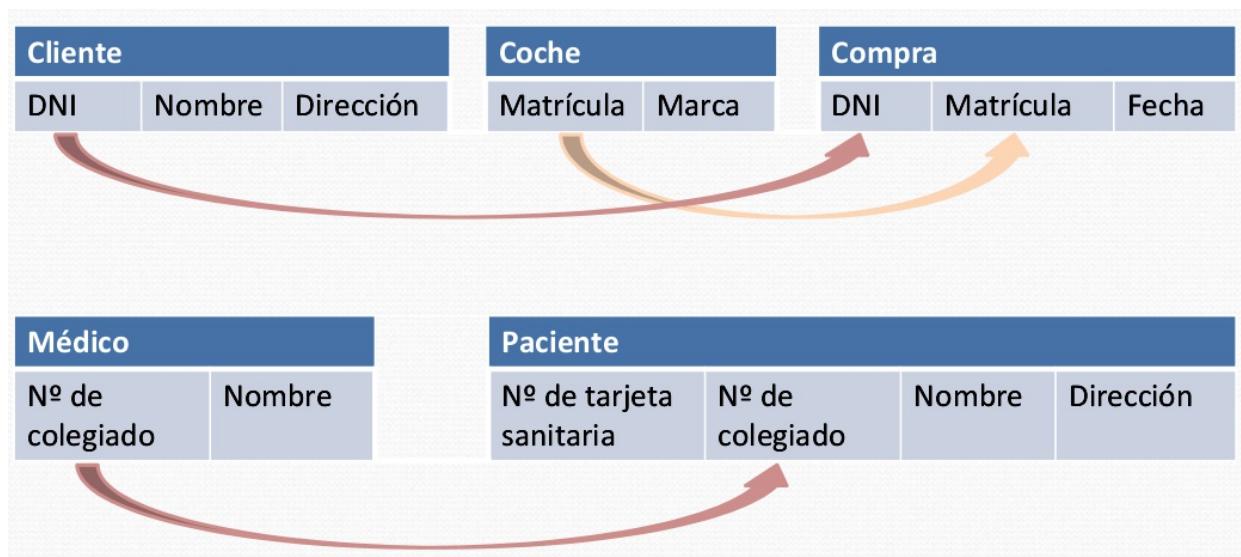
- i. Todos los registros de una tabla son del mismo tipo. Para almacenar registros de tipos distintos se usan tablas distintas.
- ii. En ninguna tabla aparecen campos repetidos.
- iii. En ninguna tabla existen registros duplicados.
- iv. El orden de los registros en la tabla es indiferente. En cada momento se pueden recuperar los registros en un orden particular.
- v. En cada tabla hay una llave, formada por uno o varios campos.

## 3.9 Transformación del modelo entidad-relación al modelo de datos relacional

Dado un Diagrama entidad-relación, el paso a tablas o relaciones del Modelo de datos relacional se efectúa como sigue:

- **Conjuntos de entidades:**
  - Se define una tabla para cada conjunto de entidades.
  - Para cada atributo se define una columna en la tabla.
- **Conjuntos de relaciones sin atributos propios:**
  - **Cardinalidad 1:1:** En la tabla de uno de los conjuntos de entidades (el que se considere principal) se añaden las columnas necesarias para albergar los atributos que forman la clave del otro conjunto de entidades.
  - **Cardinalidad 1:N o N:1:** En la tabla del conjunto de entidades que participa de forma “muchos” se añaden las columnas necesarias para albergar los atributos que forman la clave del otro conjunto de entidades.
  - **Cardinalidad N:M:** Se define una tabla propia para el conjunto de relaciones y, en ella, se definen las columnas necesarias para albergar los atributos que forman la clave de cada uno de los conjuntos de entidades que relaciona ese conjunto de relaciones.
- **Conjuntos de relaciones con atributos propios:**
  - **Cardinalidad 1:1, 1:N o N:1.** Se puede seguir el mismo enfoque dado para conjuntos de relaciones sin atributos propios, pero añadiendo también las columnas necesarias para albergar los atributos del conjunto de relaciones en la misma tabla del conjunto de entidades donde se añada la clave del otro conjunto de entidades. Esta forma está desaconsejada desde el punto de vista conceptual y por cuestiones de mantenimiento del software .
  - **Cardinalidad N:M, 1:1, 1:N o N:1.** Se define una tabla propia para el conjunto de relaciones y, en ella, se definen las columnas necesarias para albergar los atributos que forman la clave de cada uno de los conjuntos de entidades que relaciona ese conjunto de relaciones; en esa misma tabla, además, se definen las columnas necesarias para albergar los atributos propios del conjunto de relaciones.

## Ejemplos.



## 4. Sistema de gestión de la bases de datos

### 4.1 Definición de sistema de gestión de base de datos

**Sistema de gestión de la base de datos - SGBD** (*Data Base Management System - DBMS*): conjunto de software destinado a la creación, control y manipulación de la información de una base de datos.

Un SGBD debe permitir la realización de las siguientes **tareas**:

1. **Definición** del esquema de la base de datos y de los distintos subesquemas.
2. **Acceso** a los datos desde algún lenguaje de alto nivel.
3. **Interrogación** (o recuperación de la información) directa en modo conversacional.
4. **Organización** física de la base de datos y recuperación tras fallos del sistema.

### 4.2 Lenguajes específicos en un SGBD

Las tres primeras tareas se realizan mediante dos lenguajes específicos:

- **Lenguaje de descripción de datos (DDL, Data Description Language)**. Se usa para la descripción del esquema y de los subesquemas.
- **Lenguaje de manipulación de datos (DML, Data Manipulation Language)**. Se utiliza para el acceso a la base de datos desde lenguajes de alto nivel o en modo conversacional.

**El sistema de gestión de la base de datos** actúa como intermediario entre los programas de aplicación y el sistema operativo, lo que permite que los programas sean independientes de la estructura física de los datos.

# Tema 5. Generación y depuración de aplicaciones

---

## 1. Plataforma

---

### 1.1 Concepto de plataforma

**Plataforma:** combinación de hardware y/o software utilizada para ejecutar aplicaciones software.

La versión más simple de plataforma puede ser una arquitectura de computadora o SO.

Ejemplos:

Sistema Operativo	Arquitectura Hardware
Microsoft Windows	x86
Linux/Unix	X86, RISC, SPARC
Mac OS X	X86, PowerPC
Android	Dispositivos móviles basados en arquitecturas ARM, MIPS y x86
Java	Múltiples SOs para los que existen implementaciones de <i>Java Virtual Machine</i> , y por tanto múltiples arquitecturas

Notas:

Un programa normal va a depender de una plataforma (hardware (máquina) y software (SO)).

Aunque algunos

programas no necesitan SO (tienen un chip que hace la función de SO).

Normalmente, las plataformas nos ofrecen una máquina virtual.

### 1.2 Clasificación del software

En cuanto a plataformas, el software se puede clasificar como:

1. Dependiente de una plataforma particular para la cual se desarrolla y ejecuta, bien sea esta hardware, SO o máquina virtual.
2. Multiplataforma cuando el software es implementado e interopera en varias plataformas. Una aplicación multiplataforma se puede ejecutar en:
  - tantas plataformas como existan (caso ideal de software independiente de la

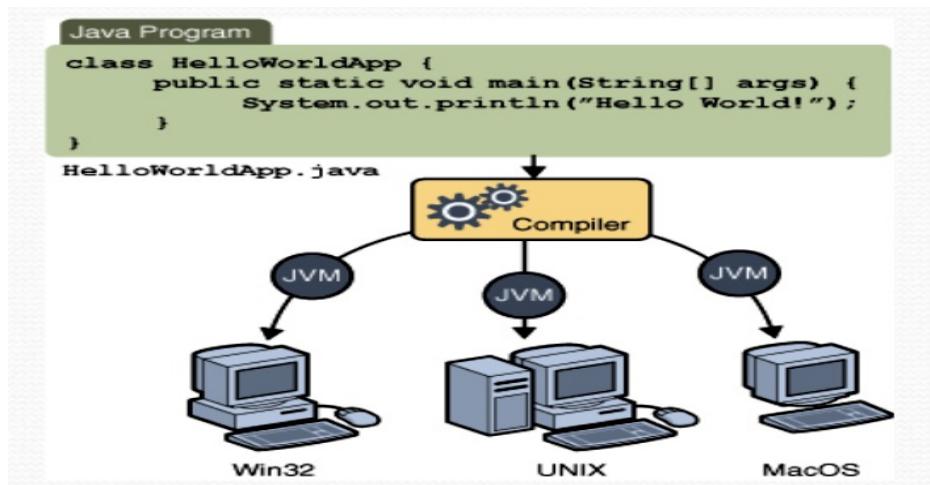
plataforma), o

- tan sólo en dos plataformas diferentes, por ejemplo, una aplicación multiplataforma se podría ejecutar en Microsoft Windows y Linux en arquitecturas x86.

## 1.3 Software multiplataforma

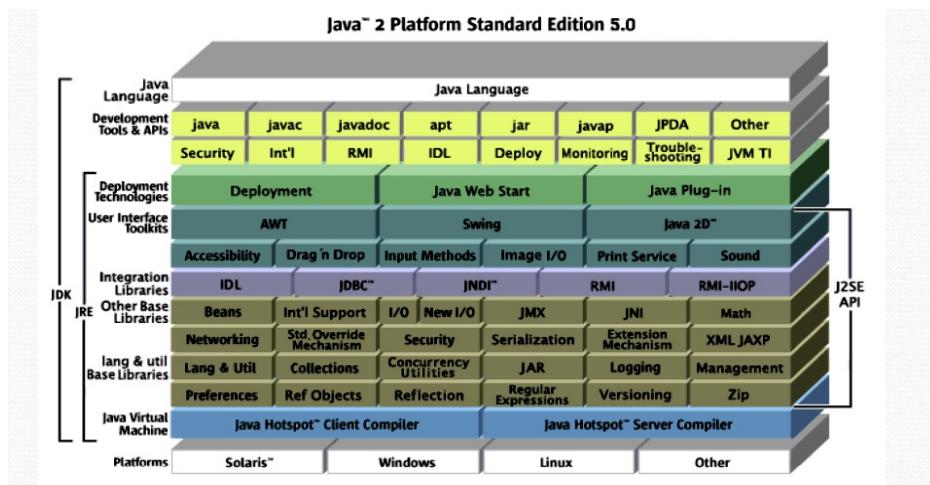
El software multiplataforma puede dividirse en dos tipos:

1. Aplicaciones que requieren su creación o compilación para cada plataforma específica donde se ejecutará.
2. Aquellas otras aplicaciones que directamente se pueden ejecutar en más de una plataforma sin preparación especial (e.g. plataforma Java): aplicaciones escritas en un lenguaje interpretado (o precompilado en un código intermedio) portable, es decir, cuando el intérprete y los paquetes para su ejecución son estándares para varias plataformas.

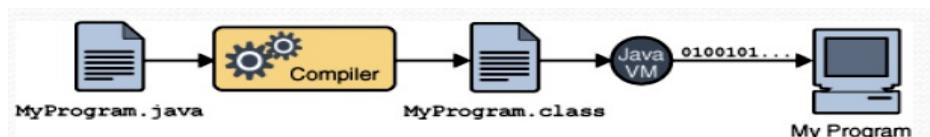


## 1.4 Plataforma Java

Ejemplo de plataforma a mayor nivel de abstracción (incluye lenguaje de programación) que el proporcionado por un sistema operativo.



- Requiere máquina virtual JVM (Java Virtual Machine), lo cual posibilita que el mismo código se pueda ejecutar en todos los sistemas que implementen JVM.
- Los ejecutables Java no se ejecutan nativamente sobre el sistema operativo, i.e., ni Windows, Linux, Mac OS X, etc, ejecutan programas Java directamente. Sin embargo, Java Native Interface (JNI) permite el acceso a funciones específicas del sistema operativo.- Para aplicaciones Java móviles: Windows y Mac OS utilizan plugins en los navegadores para su ejecución; y Android soporta Java directamente.
- JVM ejecuta programas Java compilados a lenguaje intermedio (bytecodes) independiente de hardware y sistema operativo donde se ejecuta; los programas Java son multiplataforma, pero no la JVM (hay una para cada sistema operativo).
- Hay un compilador JIT (Just In Time) dentro JVM (version 1.2 en adelante) que traduce Java bytecodes en instrucciones nativas del procesador en tiempo de ejecución, las cuales son almacenadas para su posterior reutilización.
- El uso del compilador JIT permite que, después de un breve retardo en la carga y prácticamente su total compilación, las aplicaciones Java se ejecuten tan rápidamente como programas nativos.



## 2. Framework de desarrollo de aplicaciones

### 2.1 Herramientas básicas para el desarrollo software en Linux

Fases	Herramientas
Generador de código fuente	Editores de texto ( <code>pico</code> , <code>emacs</code> , <code>xemacs</code> , ...).
Sangrado código fuente	<code>indent</code> sangra un programa en C sintácticamente correcto
Compilación código fuente	<code>gcc</code> y <code>g++</code> de GNU pre-procesa, compila, optimiza, y enlaza para generar archivos ejecutables.
Gestión de software basado en módulos	<code>make</code> actualiza archivos en base a relaciones de dependencia previamente almacenadas
Gestión de bibliotecas	<ul style="list-style-type: none"> <li>• <code>ar</code> permite crear y manipular archivadores en base a conjunto de archivos.</li> <li>• <code>ranlib</code> genera y añade una tabla de índice de contenidos a archivadores acelerando la fase de enlazado.</li> <li>• <code>nm</code> visualiza información de archivos objeto que ayuda a depurar bibliotecas</li> </ul>
Control de versiones	CVS (Sistema Concurrente de Versiones), es una interfaz a RCS (Sistemas de Control de Revisiones), permite gestión de versiones en múltiples directorios y con múltiples desarrolladores.

## 2.2 Integrated Development Environment (IDE)

- Entorno integrado de desarrollo (IDE): aplicación que proporciona un conjunto de herramientas relacionadas para el desarrollo del software: creación y edición de código fuente, generadores (compiladores, intérpretes, enlazadores, gestores de bibliotecas, etc) de código objeto, y despliegue y depuración de programas.
- La integración de herramientas contrasta con el desarrollo utilizando las herramientas aisladas que incluye Linux (gcc, make...)
- IDEs ejemplo: Microsoft Visual Studio, Eclipse, SharpDevelop...
- IDEs actuales para el desarrollo de software orientado a objetos (ej: Netbeans) incluyen otras herramientas adicionales: navegadores para diagramas de jerarquía de clases, inspectores de objetos, etc.

## 2.3 Objetivos de un IDE

- Maximizar la productividad de los programadores con herramientas provistas de interfaces de usuario similares; todo el desarrollo se lleva a cabo bajo una misma aplicación.
- Reducir tareas de configuración de múltiples herramientas proporcionando un conjunto de facilidades de forma cohesiva.
- Aprender rápidamente a utilizar un IDE que integra manualmente todas las herramientas.
- Acelerar el aprendizaje de lenguajes de programación, e.g., el código se puede analizar mientras se edita proporcionando información inmediata acerca de errores léxicos, sintácticos, etc.

## 2.4 Tipos de IDE

1. Dedicados a un sólo lenguaje de programación y con un conjunto de características propias del paradigma de programación al cual pertenece (e.g. herramienta para manejo de jerarquía de clases en orientación a objetos).

2. IDEs que soportan múltiples lenguajes de programación:
  - Alternativamente mediante plugins (es posible instalar varios lenguajes al mismo tiempo): Los IDEs Eclipse y Netbeans soportan entre otros C/C++, Ada, Perl, Python, Ruby, y PHP; o
  - al mismo tiempo para un conjunto de lenguajes/plataformas relacionados: Microsoft Visual Studio y Xcode (OS X/iOS y lenguajes C/C++, Objective-C, Java, AppleScript, Python...).

IDE ejemplo: NetBeans para Java.

## 2.5 IDEs y programación visual

La programación visual hace uso de IDEs que permiten a los diseñadores/programadores crear nuevas aplicaciones combinando bloques/nodos de código mediante diagramas de estructura y de flujos, normalmente basados en UML (Unified Modeling Language).

Ejemplos:

1. Lego Mindstorms System: utilizando la potencia de navegadores como Mozilla.
2. KTechlab: IDE abierto y simulador para desarrollar software para microcontroladores.
3. LabVIEW y EICASLAB: especializados en programación distribuida.

## 2.6 Framework de desarrollo Software

- Framework de desarrollo software: abstracción que proporciona software con funcionalidad genérica que puede cambiarse selectivamente mediante código de usuario para la creación de aplicaciones.
- Incluye herramientas similares a las encontradas en IDEs (compiladores, bibliotecas...), así como una API (Application Programming Interface).
- Un framework tiene características clave para la reutilización software que lo distinguen de otras alternativas tales como bibliotecas o bloques/nodos de código en IDEs.

## 2.7 Objetivos de un Framework

- Facilitar y reducir el tiempo el desarrollo evitando dedicar tiempo a detalles de bajo nivel. Por ejemplo, un desarrollador debe escribir la funcionalidad para la gestión de un sistema bancario en Web en lugar de los mecanismos para manejar peticiones y gestión de estado tales como crear hebras para atender una nueva petición cuando el resto de hebras ya sirven otras peticiones previas.
- Debido a la complejidad de las APIs, un framework conlleva un tiempo extra de aprendizaje inicialmente.
- Generalmente los frameworks se centran en dominios específicos: compiladores para

diferentes lenguajes y plataformas, sistemas de soporte a la decisión, middleware, computación de altas prestaciones.

## 2.8 Características de un Framework

- Inversión de control: a diferencia de las bibliotecas o aplicaciones normales, el código del framework invoca al código proporcionado por el usuario del framework.
- El framework tiene un comportamiento útil por defecto.
- Extensibilidad: un framework puede ser ampliado sobreescribiendo de forma selectiva o especializa su código con la funcionalidad específica proporcionada por el usuario.
- El código del framework no puede ser modificado, excepto por extensibilidad (como se ha comentado en el punto anterior).

## 2.9 Arquitectura de un Framework

- Un framework se define mediante componentes básicos, y las relaciones entre ellos, que no pueden ser modificados (frozen spots).
- Existen partes predeterminadas donde el programador añade su código con la funcionalidad específica deseada (hot spots).
- Ejemplo de arquitectura de un framework orientado a objetos:
  - El framework consta de clases abstractas y concretas, y su instanciación consiste en componer y especializar dichas clases.
  - Las clases definidas por el usuario reciben mensajes de las clases del framework (inversión de control); los desarrolladores manejan ésto implementando los métodos abstractos de la superclase.

## 3. Técnicas de depuración de programas

---

Apuntes de clase:

1. Hago el programa.
2. Quiero ver si cumple la especificación o diseño (lo que tiene que hacer), puedo hacerlo de dos formas:
  - 2.1 Verificación formal de programas:

- Tengo una serie de sentencias escritas en un lenguaje y una especificación (es un conjunto de requisitos).
- Mi programa es un conjunto de órdenes.
- En la especificación de la semántica del lenguaje de programación se hace una especificación axiomática.

- Axiomática: la especificación de la semántica del lenguaje de programación es un sistema deductivo (conjunto de axiomas y conjunto de diferencias).
- Hay dos tipos de sentencias:
  - Semánticamente simples: leer una variable y escribir una variable. Se especifican mediante axiomas.
  - Semánticamente compuestas: se especifican mediante reglas de diferencias.
- A partir de la especificación semántica se construye un algoritmo (Lógica de Hoare). Se construye esa lógica (que es el lenguaje de la lógica), meto el sistema deductivo, admite el lenguaje de programación y el de especificación como lenguajes correctos.
- Una vez construida esa lógica, mi programa se convierte en una teoría (conjunto de fórmulas bien formadas).
- Por lo que la verificación formal consiste, con este sistema deductivo, en probar que cada uno de los requisitos de mi especificación se cumplen. Por lo que hay que probar que los requisitos se cumplen en esta teoría, con el sistema deductivo.
- Esta comprobación lo hace un humano.
- Se hace en programas muy delicados para asegurarme de que están bien.
- Lo que estoy verificando es que la literatura del programa satisface las especificaciones.
- Pero no verifico que si cuando se ejecute se va a cumplir.
- Por lo que hacemos otra cosa: probar la ejecución de programas.

## 2.2 Forma general:

- Se diseña un conjunto de datos para ejecutar el programa de los cuales sabemos los resultados.
- Se diseña para que haga todos los posibles recorridos que tiene que hacer el programa.
- Por lo que así no tengo ningún motivo para pensar que no es correcto, pero no puedo pensar que es correcto totalmente.
- Suponemos que el programa no funciona: hay que corregir el programa. En programas complejos es más habitual que el fallo se detecte donde no se ha producido el fallo. Por lo que hay que ir haciendo trazas en la ejecución, para ver dónde falla.
- Cuando uno corrige el fallo, hay que tener cuidado con la propagación de las correcciones, porque yo he podido corregir algo, pero estoy desajustando otras cosas.

## 3.1 Definición y objetivos

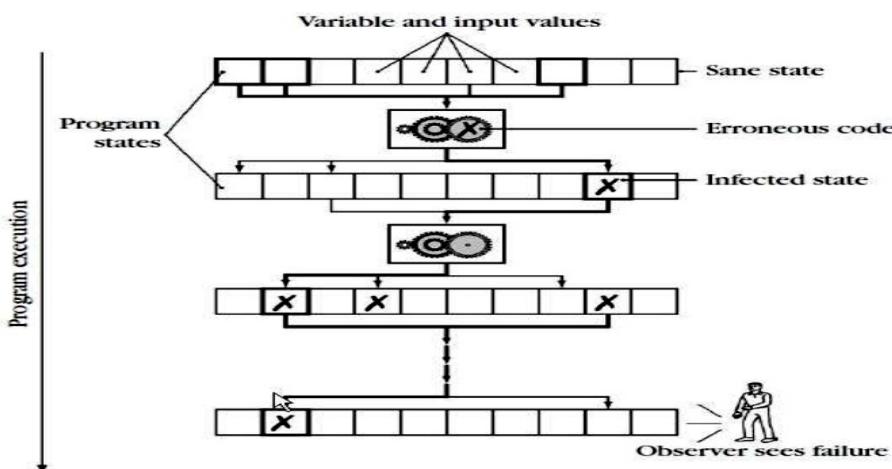
- **Depuración:** es una actividad compleja consistente en encontrar y solucionar errores cometidos en el diseño y codificación de programas.
- **Objetivos** de las técnicas aplicadas a la depuración:
  - Incrementar la productividad encontrando y solucionando errores más rápida y efectivamente.
  - Mejorar la calidad de los programas eliminando defectos.

- Prevenir errores como mejor solución.
- Mejorar lenguajes de programación y herramientas identificando errores estáticamente y detectando el incumplimiento de invariantes dinámicamente, e.g., definición de sistemas de tipos en Java y C#.

## 3.2 Fases para la generación de fallos

- **Creación de un defecto:** pieza de código creada por el programador que puede causar infección como consecuencia de un error, cambios en los requisitos originales, o interacciones impredecibles entre componentes (e.g. programas distribuidos).
- **El defecto produce infección.** los estados a alcanzar en la ejecución del programa difieren de los previstos por el programador.
- **Propagación de la infección.** Como un programa accede a su estado actual para su ejecución, una infección se puede propagar a otros estados.
- **La infección causa el fallo:** error observable externamente en el comportamiento de un programa.

## 3.3 Ejecución de un programa como secuencia de estados



## 3.4 Propiedades en la generación de fallos

- Un estado de programa viene definido por los valores de las variables y la posición de ejecución (contador de programa).
- Cada estado determina los siguientes estados hasta alcanzar el estado final.
- Las pruebas muestran la presencia de defectos pero nunca la ausencia:
  - Hay defectos que no provocan infecciones, y hay infecciones que no provocan fallos.
  - La ausencia de fallos no implica ausencia de defectos.

- La cadena de infección viene definida por la relación causa-efecto desde el defecto a fallo; la depuración consiste en identificar dicha cadena de infección para eliminar el defecto.

## 3.5 Fases de depuración

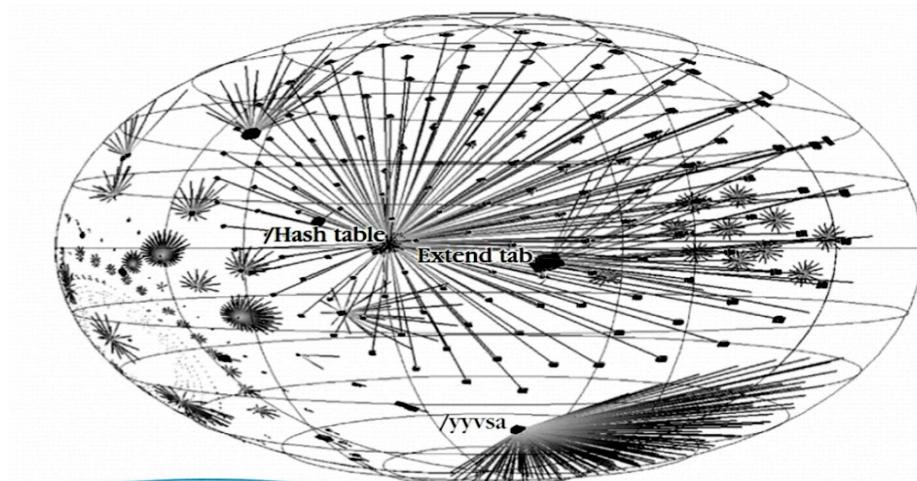
1. Registrar el problema.
2. Reproducir el fallo (the failure): más complicado para programas no-deterministas y de larga ejecución.
3. Automatizar y simplificar el caso de prueba.
4. Encontrar posibles orígenes de la infección.
5. Centrar la búsqueda en los orígenes más probables.
6. Aislar la cadena de infección.
7. Corregir el defecto: sencillo si está claro el defecto que produce el fallo.

## 3.6 Características del proceso de depuración

- Las fases 4-6 están directamente relacionadas con comprender cómo se produce el fallo.
- La depuración es un problema de búsqueda en dos dimensiones:
  - i. Espacio: parte del estado (conjunto de variables) que está infectado.
  - ii. Tiempo: cuando tiene lugar la infección (estado).
- La búsqueda es de envergadura incluso para los programas sencillos:
  - i. Los estados pueden comprender muchas variables.
  - ii. La ejecución puede constar de muchos estados.

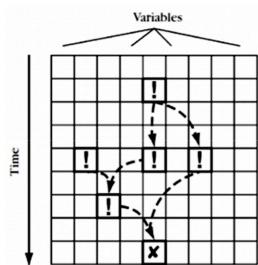
## 3.7 Ejemplo de estado de ejecución

- Compilador GNU.
- 44.000 variables (vértices).
- Aproximadamente 42.000 referencias entre variables (arcos).



## 3.8 Búsqueda del defecto

- Se aplican dos principios básicos para la búsqueda:
  - i. separar estados sanos de infectados, y
  - ii. separar variables relevantes e irrelevantes.
- Un fallo puede ocurrir debido a ciertos valores de variables en estados anteriores (!), lo cual determina dependencias que ayudan a localizar el defecto.



## 3.9 Programación estructurada y depuración

La construcción de programas estructurados ayuda de forma importante a la depuración gracias a las siguientes características:

1. Contiene un conjunto de funciones bien definidas.
2. Utiliza construcciones iterativas (como while y for) en vez de goto.
3. Utiliza variables que tienen un propósito y a las cuales se les ha dado un nombre significativo.
4. Utiliza tipos de datos estructurados para representar datos complejos.
5. Utiliza ADTs (Abstract Data Type) o directamente el paradigma de programación orientado a objetos.

## 3.10 Depurador

Definición: Herramienta software que ayuda a la ejecución controlada de un programa para ayudar a encontrar defectos que producen fallos.

Características:

1. Interactivo. Aunque se puede usar en modo batch, su rol principal es interactuar con el programador.
2. Proporciona un conjunto de instrucciones que indican las acciones de depuración a realizar.
3. Permite detectar errores en tres categorías:
  - Sintácticos: detectados durante la fase de compilación o la de enlazado.

- Ejecución: e.g., acceder fuera del espacio de direcciones asignadas (segmentation fault) o realizar una operación de división por cero, ambas produciendo la terminación del programa.
- Lógicos: e.g., bucles que iteran más o menos veces de las previstas.

## 3.11 Tipos de errores comunes

Los tipos de errores más comunes que se producen cuando se programa son:

1. Escribir código de una forma desestructurada. Cuando un programa se estructura adecuadamente, se divide el problema en subproblemas sobre los cuales se puede trabajar independientemente (razonar sobre su solución y probarla) y después componer las distintas partes. Esto facilita mucho la depuración posterior.
2. Programar sin pensar: antes de comenzar a codificar hay que diseñar la solución al problema planteado, pensando las distintas alternativas de solución posibles y entendiendo el algoritmo que se ha de seguir.

## Bibliografía

---

- A. Prieto, A. Lloris, J.C Torres, Introducción a la informática, McGraw-Hill, 2006.
- W. Stallings, Sistemas Operativos, Aspectos Internos y Principios de Diseño (5<sup>a</sup> Edición). Pearson Education, 2005.
- J. Carretero, F.García, P. de Miguel, F. Pérez, Sistemas Operativos (2<sup>a</sup> Edición), McGraw-Hill, 2007.
- A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compiladores. Principios, Técnicas y Herramientas (2a Edición). Addison Wesley, 2008.