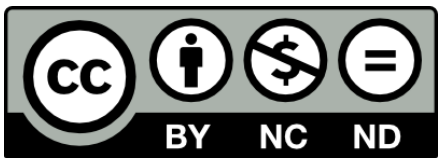
Fundamentos del Software

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Fundamentos del Software

Los Del DGIIM, losdeldgiim.github.io

Arturo Olivares Martos

Granada, 2023

Índice general

1. Sistemas de Cómputo

Definición 1.1 (Firmware)**.** El firmware de un sistema de cómputo es la lógica de más bajo nivel (controla los circuitos electrónicos de un dispositivo $\longrightarrow$ hardware de más bajo nivel) $\longrightarrow$ conjunto de instrucciones máquina grabadas en una memoria de solo lectura (microprogramas/microinstrucciones).

Definición 1.2 (BIOS)**.** La BIOS es el firmware de una computadora para activarla desde su encendido y preparar el entorno para cargar el S.O. en la memoria RAM y en el disco duro.

Definición 1.3 (Sistema Operativo)**.** Es una capa intermedia entre las "utilidades y herramientas" y el hardware para que:

- Los usuarios finales no se tengan que preocupar del hardware y vean el ordenador como un conjunto de aplicaciones.
- Los programadores desarrollen programas de aplicación utilizando utilidades y otros programas del sistema. Tienen la ayuda del SO: editores, enlazadores, compiladores, depuradores...
- Acceder a los dispositivos de E/S de forma transparente a las instrucciones específicas que utiliza cada dispositivo.
- Acceder a los ficheros de forma controlada.
- Detectar distintos tipos de errores durante la ejecución.
- Acceder a recursos del sistema sin conflictos.
- Monitorizar el sistema y proporcionar estadísticas de uso.

En definitiva, facilitar el uso del ordenador de una forma eficiente.

Definición 1.4 (Interrupciones)**.** Señal recibida por el procesador que indica que debe "interrumpir" la ejecución del código actual y pasar a ejecutar un código específico para tratar esa situación (suspensión temporal de procesos).

- Instrucciones por hardware Son las habituales de las operaciones de E/S. No son producidas por instrucciones del proceso, sino por señales del dispositivo de E/S para indicar al procesador que necesita ser atendido.
- Interrucpiones por software o *trap* Son lo mismo que las llamadas al sistema generadas por un programa durante su ejecución. (Siempre que hay una instrucción que requiere del SO para su ejecución, por ejemplo, acceso al disco duro, donde se necesita ejecutar una rutina del SO)

Definición 1.5 (Excepciones)**.** Interrupciones síncronas causadas típicamente por un error en un programa (división por 0, acceso a direcciones de memoria protegidas, ...). En esos casos, se produce una ejecución del SO para garantizar la integridad de los datos, intentar solucionar el error o al menos informar de ello al usuario.

2. Introducción a los Sistemas Operativos

Definición 2.1 (Procesamiento en serie)**.** Se dio a finales de los 40 y a principios de los 50. Se caracteriza por:

- Tarjetas perforadas
- No había SO: se cargaba el programa en código máquina utilizando las tarjetas
- Los errores se examinaban a mano $\longrightarrow$ reiniciar desde el principio.
- Los usuarios pedían cita

Definición 2.2 (Sistemas por lotes (Sistemas Batch))**.** Se caracteriza por la existencia de un software específico llamado *monitor*.

- Agrupación de trabajos similares (evitar los problemas del procesamiento en serie)
- Parte del monitor estaba cargado en memoria (monitor residente) y otras utilizadas se cargaban como subrutinas del programa del usuario.

El proceso seguido era:

1. Monitor lee un trabajo del core.
2. El trabajo se carga en memoria.
3. El procesador ejecuta el trabajo.
4. Vuelta al 1).

Los problemas eran:

- El procesador estaba ocioso
- Recursos infravalorados.

Definición 2.3 (Multiprogramación)**.** Cuando un proceso se pone en espera, en lugar de estar el procesador inactivo esperando, el SO cambie a otro proceso y lo ejecute (y así sucesivamente). Implica:

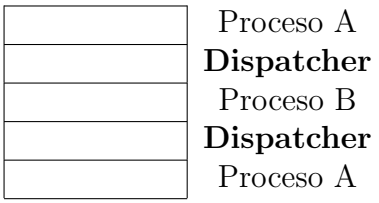
- Tener cargado en memoria todos los procesos

- Tener una planificación de la CPU (decidir entre varios procesos activos)

Definición 2.4 (SO de Tiempo Compartido)**.** SO multiprogramado donde se establecen turnos de CPU y el SO va decidiendo que procesos va a ejecutarse en el siguiente ciclo. Logra eficiencia a la hora de ejecutar todos los procesos en el menor tiempo posible.

Un ejemplo de tiempo compartido es el DMA, ya que mediante el uso de interrupciones, el controlador del dispositivo de E/S se encarga de la operación de E/S mientras el procesador puede estar encargándose de otros procesos (el controlador va activando interrupciones).

Definición 2.5 (Dispatcher)**.** Es un módulo del SO que se encarga de intercambiar procesos.



Definición 2.6 (Proceso)**.** Está formado por un programa ejecutable y datos que necesita el SO para ejecutar el programa. Está relacionado con:

- La operación en lotes en sistema multiprogramados (interrupciones).
- Tiempo compartido (múltiples procesos ejecutándose utilizando pequeños intervalos de tiempo)
- Sistemas de transacciones en tiempo real
  - Proceso $\longrightarrow$ Uso de transacciones (los procesos se suspenden a través de las por interrupciones, se almacena su estado y el procesador pasa a ejecutar otro proceso)

Pueden producirse errores relacionados con:

- Sincronizaciones inadecuadas (se interrumpe un proceso y no se restaure adecuadamente).
- Violaciones de exclusión mutua (hay recursos del sistema que sólo pueden utilizarse por un único proceso simultáneamente).
- Operaciones no deterministas de los programas (protección del espacio de memoria de los procesos).
- Interbloqueos (que existan varios programas que se estén esperando entre sí debido al uso que se está haciendo de los recursos).

Para evitar estos errores, los procesos constan de:

1. Un programa ejecutable.

2. Datos necesarios que requiere el SO para ejecutarlo (incluyendo el contexto de la ejecución). Se denomina bloque de control de proceso (PCB) formado por:

   a) Identificador de proceso

   b) Contexto de ejecución: contenido de los registros del procesador.

   c) Memoria donde reside el programa y sus datos.

   d) Información relacionada con recursos del sistema.

   e) Estado: en que situación se encuentra el proceso en cada momento (modelo de estados).

   f) Otra información.

Definición 2.7 (Traza de ejecución (de un proceso))**.** Es la secuencia de instrucciones que se ejecutan en cada ciclo para ese proceso.

2.1. Modelo de dos estados

Tiene el inconveniente de que es demasiado simple.

Un proceso puede estar en ejecución o no (los cambios de un estado a otro los realiza el dispatcher).

2.2. Modelo de cinco estados

Cabe contemplar que los estados que NO están en ejecución pueden estar:

Nuevos que todavía no han sido aceptados para ejecución.

Preparados para ejecutarse.

Bloqueados (ej: porque están esperando a que termine una operación de E/S)

Terminados debido a que se han extraídos del conjunto de procesos aceptados para ejecución.

2.3. Gestión de Procesos

- Lista de procesos: posiciones de memoria en la que se almacena la dirección de memoria de inicio de cada proceso.
- Cada proceso es una estructura de datos para coordinar su ejecución.
- El PC almacena la dirección de la siguiente instrucción que debe ejecutarse del proceso que indica el registro "índice de proceso" (IP).
- El registro base contiene la dirección incial del proceso.
- El registro límite incluye el tamaño del proceso

Para crear un proceso, se ha de inicializar el PCB de la siguiente forma:

1. Asignar un identificador único al proceso.
2. Asignar un nuevo PCB.
3. Asignar la memoria correspondiente.
4. Inicializar PCB.
   - PCB $\longrightarrow$ Dirección inicial de comienzo del programa.
   - SP $\longrightarrow$ Dirección de la pila de sistema.
   - Memoria donde reside el programa
   - El resto de campos se inicializan a valores por defecto.

2.4. Cambios de modo

Definición 2.8 (Cambios de Modo)**.** Cuando se produce el cambio de modo usuario a modo supervisor/kernel o viceversa.

Tras una interrupción, excepción o llamada al sistema; puede haber un cambio de modo de usuario a kernel.

1. El hardware guarda automáticamente el PC y PSW como mínimo. El bit de modo cambia a kernel.
2. Se determina la ruta del SO que debe ejecutarse y se almacena en el PC la dirección de comienzo.
3. Se ejecuta la rutina. Suele empezar por guardar toda la información y terminar restaurándola.
4. Volver a la rutina del SO previa. El hardware restaura el PC y PSW.

2.5. Cambios de Contexto

Definición 2.9 (Cambio de contexto)**.** Cuando se pasa la ejecución de un proceso a otro.

Se producen al cambiar de proceso, y son llevados a cabo por el Dispatcher. Los pasos a seguir, tras una interrupción, excepción o llamada al sistema, son:

1. Guardar los registros en el PCB del proceso ejecutándose.
2. Actualizar el campo del estado del proceso al nuevo estado e insertar el PCB en la cola.
3. Seleccionar un nuevo proceso de los "Preparados". Se encarga el Planificador de la CPU o el *Scheduler*.
4. Actualizar el estado del nuevo proceso a "Ejecutándose", y sacarlo de "Preparados".

5. Cargar los registros del procesador con el PCB del proceso.

Hasta el momento, hemos visto que la tarea fundamental del SO es gestionar procesos:

- Reservarles recursos
- Permitirles intercambiar información
- Garantizar uso recursos y protección
- Sincronizar recursos

2.6. Hebras o Hilos

Definición 2.10 (Hebra)**.** Tiene como objetivo separar:

- El control de la asignación de los recursos (procesos)
- De la ejecución de los programas (división de los procesos en hilos que se pueden ejecutar de forma independiente)

Para ello,

- Sigue habiendo un único bloque de control del proceso (PCB) y espacio de direcciones del usuario.
- Cada hilo tiene una pila y un bloque de control (TCB, *Thread Control Block*) que contiene valores de los registros, prioridad, resto de información de estado, etc.

Por tanto, todos los hilos de un proceso comparten el estado y los recursos asociados al proceso. Por tanto, es necesario planificar la sincronización de las actividades de los hilos.

Con el concepto de hebra, pasamos a *sistemas multihilo*, los cuales separan el punto de control de la asignación de recursos (procesos) y el de ejecución de los programas (cada proceso se divide en hebras, de forma que la ejecución de una hebra se produce de forma independiente al resto de hebras).

Definición 2.11 (Modelo Monohilo)**.** Un único bloque de control (PCB), espacio de direcciones de usuario, pilas de usuario y kernel.

Definición 2.12 (Modelo Multihilo)**.** Hay un único bloque de control del proceso (PCB) y espacio de direcciones del usuario. Cada hilo tiene su propio bloque de control (TCB), sus propias pilas de usuario y kernel.

2.6.1. Modelo de 5 estados para hebras

- Cuando se crea un espacio, se crea un hilo para dicho proceso.
- Posteriormente los hilos pueden generar nuevos hilos para el mismo proceso.
- Cada nuevo hilo, se coloca en la cola de **preparados**.
- Cuando un hilo necesita esperar a un evento, se **bloquea**.
- Cuando se produce el evento en espera, el hilo pasa al estado **preparado**.
- Cuando termina la ejecución del hilo, se liberan los registros y pilas asociados.

2.6.2. Ventajas de las Hebras

- Menor tiempo de creación de una hebra en un proceso ya creado que crear un nuevo proceso.
- Menor tiempo de finalización de una hebra que de un proceso.
- Menor tiempo de cambio de contexto entre hebras de un mismo proceso.
- Facilitan la comunicación entre hebras de un mismo proceso.

2.7. Gestión de Memoria

El concepto de multiprogramación se basa en compartir la memoria entre todos los procesos de forma eficiente y segura. Es segura, ya que cada proceso debe tener su espacio de memoria.

Normalmente los programadores desconocen a priori en qué posiciones de memoria se va a almacenar el programa durante su ejecución (direcciones absolutas).

Es conveniente poder intercambiar los procesos en la memoria principal para hacer más eficiente el uso del procesador $\longrightarrow$ trabajar con direcciones relativas $\longrightarrow$ hardware del procesador y del SO que se encarga de ir traduciendo de direcciones relativas a direcciones absolutas.

2.7.1. Carga de procesos en memoria

El primer paso que es necesario para generar un proceso consiste en cargarlo en memoria $\longrightarrow$ **cargador**.

1. Carga absoluta: Durante la compilación se añaden al código máquina direcciones absolutas de memoria. $\longrightarrow$ El programa no es reubicable (transparencia 34 $\longrightarrow$ X a N).

2. Carga reubicable: La carga del programa puede realizarse en cualquier espacio de la memoria principal. Para ello, el compilador genera direcciones lógicas relativas (de 0 a M).

a) Reubicación estática: El cargador decide dónde se va a ubicar en memoria el programa y añade la dirección base a las relativas (compilador).

b) Reubicación dinámica en tiempo real: la traducción a direcciones absolutas se realiza en tiempo de ejecución ⟶ es necesario un hardware del procesador.

2.7.2. Espacio de direcciones de memoria

Pueden ser direcciones lógics o físicas. Diapositiva 37.

- Las direcciones que genera un procesador son direcciones lógicas (relativas a la posición 0).
- Hay un hardware para traducir las direcciones lógicas a direcciones físicas. Es el MMU: *memory management unit*.
- Mapa de memoria de un ordenador: espacio de memoria direccionable.
- Mapa de memoria de un proceso: Tamaño total de direcciones lógicas del proceso ⟶ correspondencia con el tamaño de direcciones físicas.

2.7.3. Fragmentación

El problema de la fragmentación consiste en dividir el espacio de memoria de los procesos en fragmentos más pequeños que no se tengan que cargar necesariamente en posiciones de memorias contiguas.

Ha de realizarse adecuadamente para evitar la fragmentación tanto interna como externa:

- Fragmentación Interna:

  Espacio inutilizable dentro de un bloque de la memoria.

- Fragmentación Interna:

  Espacio inutilizable fuera de un bloque de la memoria.

1. **Paginación**

   - El espacio lógico de direcciones de un proceso se divide en elementos del mismo tamaño (páginas).
   - El espacio físico en memoria se divide en fragmentos del mismo tamaño (marcos de páginas).
   - Tamaño de página = Tamaño de marco de página = 2^n
   - Las direcciones lógicas son (p, d), mientras que las direcciones físicas son (m, d) donde p =número de página, m =número de marco, d = desplazamiento.

   La tabla de páginas contiene la información necesaria para traducir cada dirección lógica a física:

- Hay una por proceso.
- Tiene una entrada por página del proceso.
- Tiene los siguientes elementos:
  - Número de marco asociado a la página
  - Bits de protección (modo de acceso autorizado)
  - Bits adicionales, de haberlos.

La ejecución de cada instrucción requiere por tanto dos accesos a memoria. En primer lugar, para la traducción, se accede a la tabla de páginas; y en segunda lugar ya se accede a memoria.

- El **RBTP** *(registro base de la tabla de procesos)* está en el PCB del proceso y apunta a la tabla de páginas.
- El búfer de traducción adelantada, **TLB** *(translation look-aside buffer)* resuelve el problema de acceder dos veces a memoria. Es una caché hardware.

2. **Segmentación**

   Similar a la paginación, solo que en este caso cada página no mide lo mismo.

   La tabla de segmentos contiene:

   - Número de segmento
   - Dirección base, en la que empieza el segmento.
   - Longitud del segmento; es decir, desplazamiento máximo.
   - Bits de protección:
     - Bit de presencia: activado si la página se encuentra en memoria.
     - Bit de modificado: el contenido de la página se ha modificado desde que se cargó en memoria.
     - Bits de protección RWX
     - Bits de accedido

   La tabla de segmentos de encuentra en la memoria principal. Adicionalmente, tenemos:

   - El **RBTS** *(registro base de la tabla de segmentos)* está en el PCB del proceso y apunta a la tabla de segmentos.
   - El *registro Longitud de la Tabla de Segmentos* **(STLR)** indica el número de segmentos del proceso. Sirve para comprobar si un segmento s es válido. Se almacena en el PCB.

3. Compilación y Enlazado de Programas

Cód. fuente (Leng. alto nivel) $\xrightarrow{(*)}$ Cód. objeto (Cód. máquina o ensamblador)

Definición 3.1 (Compilación)**.** Proceso mediante el cual se pasa de código fuente a código objeto ($*$). Se emplea para:

- Comprobar que no hay errores en el código fuente.
- Generar ficheros objeto.

Definición 3.2 (Enlazado)**.** Proceso mediante el cual, a partir de los ficheros objeto, se obtienen los ficheros ejecutables.

3.1. Gramática

Definición 3.3 (Gramática)**.** La gramática $G = \{V_N, V_T, P, S\}$ está formada por:

1. V_N o símbolos no terminales:
   Aquellos símbolos auxiliares que podemos usar para operar con la gramática.
2. V_T o símbolos terminales:
   Aquellos símbolos que podemos usar al programar.
3. P o reglas de producción:
   Combinaciones válidas de los símbolos.
4. S o axioma:
   Uno de los símbolos no terminales que se usa como símbolo inicial.

Ejemplo. Sea $G = (\{0, 1\}, \{S\}, S, P)$, con P:

$$P = \{S ::= ^{1} 0|1|0S1\}$$

Por tanto, tengo tres reglas de producción.

- $S \longrightarrow 0$: Sí es válido, usando la primera regla.
- $S \longrightarrow 1$: Sí es válido, usando la segunda regla.

[1] Aquí se ha empleado la notación de Backus.

- $S \longrightarrow 101$: No es válido, ya que no es posible que empiece con el 1.

Ejercicio 3.1.1. La gramática definida por $G = (\{0,1,2,3,4,5,6,7,8,9\}, \{N, C\}, N, P)$, con:

$$P = \{N ::= NC|C, \quad C ::= [0-9]\}$$

Esta gramática puede generar los naturales, pero también admite los 0 no significativos. Modificar para que no admita los 0 no significativos.

Las reglas de producción serían:

$$P = \{N ::= D|ND|N0, \quad D ::= [1-9]\}$$

Definición 3.4 (Gramática Ambigua)**.** Una gramática es ambigua cuando admite más de un árbol sintáctico para una misma secuencia de símbolos de entrada.

Ejemplo de esto es la precedencia de los operadores en un lenguaje de programación, ya que se usan los paréntesis para evitar la ambigüedad.

Definición 3.5 (Gramática libre de contexto)**.** Se dice que una gramática es libre de contexto cuando en el lado izquierdo de cada regla de producción solo puede haber un símbolo no terminar. Formalmente, cada producción es de la forma:

$$A \longrightarrow \alpha \qquad A \in V_N \qquad \alpha \in (V_N \cup V_T)^*$$

donde $(V_N \cup V_T)^*$ representa todas las combinaciones posibles de dichos conjuntos.

Se dice que es libre de contexto porque A se puede sustituir por α independientemente del contexto en el que aparezca.

3.2. Traducción

Definición 3.6 (Traductor)**.** 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 máquina equivalente.

Existen dos tipos de traductores, los compiladores y los intérpretes.

3.2.1. Compilador

Definición 3.7 (Compilador)**.** Un compilador traduce la especificación de entrada (archivos fuente) a lenguaje máquina incompleto (archivos objeto) y con instrucciones máquina incompletas. Por tanto, se necesita un complemento llamado enlazador.

Definición 3.8 (Enlazador/Linker)**.** El linker completa los programas ligando las instrucciones máquina necesarias y genera un programa ejecutable para la máquina real.

3.2.2. Intérprete

Definición 3.9 (Intérprete)**.** Un intérprete hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. Cabe destacar que no se genera ningún archivo objeto ni equivalente al descrito en el compilador.

Algunas ventajas del intérprete son:

- Es más fácil detectar errores, ya que suele ser posible detenerlo para conocer los valores de las variables. Esto solo sería posible en otro caso con un *debugger*.
- Es más pedagógico.

No obstante, los inconvenientes son:

- Cada vez que se ejecute, se ha de interpretar de nuevo, ya que no se genera archivo objeto. Con un compilador, aunque la traducción sea más lenta, solo ha de realizarse una vez.
- Una instrucción que se encuentre en un bucle se ha de interpretar tantas veces como se ejecute el bucle.
- La optimización solo se puede realizar línea a línea, no se puede realizar a nivel del programa completo.

Un ejemplo de intérprete es Bash.

3.2.3. Fases en el proceso de la traducción

En primer lugar, ocurre la fase de análisis del código. Este se realiza en tres pasos: léxico, sintáctico y semántico. Posteriormente, en la fase de síntesis, se optimiza y se genera el código objeto.

1. **Fase de Análisis.**

   1.1) Análisis Léxico

   Para entender esta etapa, son necesarios los siguientes conceptos:

   Definición 3.10 (Lexema/Palabra)**.** Es un conjunto de caracteres del alfabeto que tienen significado propio.

   Definición 3.11 (Token)**.** Es el concepto asociado a un conjunto de lexemas o palabras que, según la gramática del lenguaje fuente, tienen la misma misión sintáctica.

   En el caso del lenguaje español, un token podría ser los determinantes artículos, ya que tienen la misma función sintáctica independientemente de la oración.

   En el caso de un lenguaje de programación, un token podría ser "identificador" asociado a los nombres de las variables o funciones u "operador aritmético" asociado a estos operadores.

Los tokens asociados a más de una palabra (la mayoría) deben ir acompañados del lexema reconocido anteriormente. Este es el denominado atributo, y es necesario para las fases posteriores de la traducción. Por ejemplo, el token que recoja los operadores es necesario que tenga el atributo del operador en sí, ya que a la hora de la traducción será necesario saber si se trata de una multiplicación o una división, por ejemplo.

Definición 3.12 (Patrón)**.** Es una descripción de la forma que pueden tomar los lexemas de un token. Se suelen emplear expresiones regulares.

En el caso de una palabra clave como token, el patrón es sólo la secuencia de caracteres que forman dicha palabra clave. Para los identificadores y algunos otros tokens, el patrón es una estructura más compleja (normalmente dada con una expresión regular).

Por tanto, la **función** del analizador léxico es leer el texto de origen, identificar lexemas, asociarles el token al que pertenecen pero, además, debe eliminar los comentarios y caracteres superfluos existentes en el texto de entrada (espacios en blanco, tabuladores y retornos de carro). La equivalencia entre cada lexema con su token se guardan en la **tabla de símbolos**, que es donde su guarda la información.

Ejemplo. Ejemplos de tokens son:

- **IF**: lexema asociado *if*.
- **ELSE**: lexema asociado *else*.
- **IDENT**: lexemas asociados *pi*, *dato3* o *i*, por ejemplo.

Se producirá un **error léxico** cuando el carácter de la entrada no tenga asociado a ninguno de los patrones disponibles en nuestra lista de tokens. Por ejemplo, un carácter extraño en la formación de una palabra reservada, como *wh$\tilde{n}$le*.

1.2) Análisis Sintáctico

Tiene como objetivo analizar las secuencias de tokens y comprobar que son correctas sintácticamente.

A partir de una secuencia de tokens el analizador sintáctico nos devuelve el orden en el que hay que aplicar las producciones de la gramática para obtener la secuencia de entrada; es decir, el árbol sintáctico abstracto en el que aparece el token con el atributo. Este se guarda también en la tabla de símbolos.

Se produce un **error sintáctico** cuando no se puede llegar desde el axioma hasta la palabra buscada; es decir, no se puede construir el árbol sintáctico. Ejemplo de esto podría ser, por ejemplo, paréntesis mal balanceados.

1.3) Análisis Semántico

La semántica de un lenguaje de programación es el significado dado a las distintas construcciones sintácticas. En los lenguajes de programación, el significado está ligado a la estructura sintáctica de las sentencias.

En el caso de que no se produzcan errores, actualiza en la tabla de símbolos el árbol semántico abstracto resuelto; es decir, el árbol semántico con los lexemas y su significado.

Se producen **errores semánticos** cuando se detectan construcciones sin un significado correcto (p.e. variable no declarada, tipos incompatibles en una asignación, llamada a un procedimiento con número de argumentos incorrectos, ...).

2. **Fase de Síntesis**

   2.1) 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.

   Es posible la generación de código intermedio para facilitar el proceso.

   2.2) Optimización de código

   Esta fase existe para mejorar el código mediante comprobaciones locales a un grupo de instrucciones (bloque básico) o a nivel global. Se suele realizar en el código intermedio.

   Ejemplo de esto es una asignación del tipo $b = 7{,}3$ dentro de un bucle for. En esta fase se sacará dicha instrucción del bucle.

3.3. Modelos de Memoria de un Proceso

3.3.1. Tipos de Datos (desde el punto de vista de su implementación en memoria)

- Datos estáticos - existen a lo largo de toda la vida del programa.
  - Según el ámbito de visibilidad.

    Las globales a todo el programa se encuentran fuera del main y no dependen de ningún archivo.

    Las de módulo se especifican con `static` (del módulo en cuestión) o `extern` de otro módulo, siendo un módulo cada uno de los archivos.

    Las de bloque pertenecen a una parte del programa delimitada por las llaves.

    Los datos estáticos de clase o función pertenecen a cada clase o función en concreto.
  - Constantes o variables.

    Según si se pueden modificar o no. Las constantes se almacenan en un espacio de la memoria de solo lectura, mientras que las variables han de poder modificarse también.
  - Con o sin valor inicial.
- Datos dinámicos asociados a la ejecución de una función:

  Se almacenan en la pila *(stack)* y se crean al ser llamada la función y se destruyen cuando esta termina.

Corresponde a los datos locales y a los parámetros de una función.

- Datos dinámicos controlados por el programa.

  Se almacenan en el *heap* (zona de memoria usada tiempo de ejecución para albergar los datos no conocidos en tiempo de compilación). En C++, se controlan mediante los operadores `new` y `delete`. Generalmente se gestionan mediante punteros.

  Su tiempo de vida no esta vinculado a la activación de una función sino bajo el control directo del programa que los crea cuando los necesita.

Definición 3.13 (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 en función del valor de un registro.

Por ejemplo, una instrucción puede ser `MOV R4, 32+PC`. Dependiendo del valor del `PC`, se almacenará el valor de $R4$ en posiciones distintas.

3.4. Ciclo de Vida de un Programa

Una vez que el programador ha finalizado la escritura del programa, éste debe pasar por varias fases antes de que pueda ejecutarse. Estas son:

1. Preprocesado (archivos `.i` en C).

   Se procesan los includes, las directivas de preprocesador, etc.

2. Compilación (archivos `.s` en C).

   Se genera el código en lenguaje ensamblador.

3. Ensamblado (archivos `.o` en C).

   El código ensamblador se traduce por el *assembler* a código máquina. Las referencias a símbolos que no están definidos en el módulo quedan pendientes de resolver.

4. Enlazado (archivos .exe y a.out).

   El enlazador/*linker* se encarga de, a partir de los archivos objeto y las bibliotecas, resolver las referencias pendientes y generar el archivo ejecutable.

   Como diferencia entre los archivos ejecutables y los archivos objeto, tenemos principalmente que el ejecutable contiene una cabecera en la que se indica el punto de inicio del mismo, es decir, la primera dirección que se cargará en el PC.

5. Carga y ejecución.

   Por tanto el cargador/*loader* es el que ayuda a la asignación y carga del programa como un proceso en MP en estado nuevo.

3.4.1. Compilación

En esta fase, el compilador procesa cada uno de los archivos de código fuente para generar el correspondiente archivo objeto. Se realizan las siguientes acciones:

1. Genera el código objeto y determina cuanto espacio ocupan los diferentes tipos de datos.

2. Asigna direcciones a los símbolos estáticos definidos en el módulo.

   Estas son consecutivas: en primer lugar van las constantes, luego las variables con valor inicial, y por último las que no tiene valor inicial.

   Como son estáticas y permanecen durante todo el programa, se puede asignar la dirección directamente.

3. Resuelve las referencias a los símbolos estáticos externos definidos en el módulo.

   Al ser estáticos, en el proceso de compilación ya tienen una dirección asignada. Al ser del mismo módulo, no se requiere aún del enlazador.

   Las referencias pueden resolverse mediante un direccionamiento absoluto (será necesario reubicación) o relativo al PC (estaremos ante un PIC).

4. Las referencias a los símbolos estáticos externos definidos fuera del módulo se resolverán en el enlazado.

5. Resuelve las referencias a los símbolos dinámicos almacenados en la pila.

   Se resuelven mediante direccionamiento relativo a pila. Al no aparecer en el fichero objeto (se generan en tiempo de ejecución), no requieren reubicación.

6. Resuelve las referencias a los símbolos dinámicos almacenados en el *heap*.

   Se resuelven mediante direccionamiento indirecto mediante punteros. Al no aparecer en el fichero objeto (se generan en tiempo de ejecución), no requieren reubicación.

7. Genera la Tabla de símbolos e información de depuración.

3.4.2. Enlazado

El enlazador *(linker)* debe agrupar los archivos objetos de la aplicación y las bibliotecas, y resolver las referencias entre ellos. Concretamente, se llevan a cabo las siguientes tareas:

1. Se completa la etapa de resolución de símbolos.

2. Se agrupan en regiones las zonas de las mismas características de los módulos .

   Es decir, todas la parte de código de todos los módulos se agrupa en una región; y lo mismo ocurre con datos inicializados y los no inicializados.

   Esto reduce el número de regiones que ha de gestionar el Sistema Operativo.

3. Se realiza la reubicación de módulos formando regiones, ya que hay que transformar las referencias dentro de un módulo a referencias dentro de las regiones.

Hay distintos tipos de enlazado, que determinan el ámbito de cada dato.

- Enlazado externo.

  Cada vez que aparece un identificador con enlazado externo representa el mismo objeto o función a través del total de ficheros y librerías que componen el programa. Por tanto, esto equivale a tener visibilidad global.

- Enlazado interno.

  Cada vez que aparece un identificador con enlazado interno representa el mismo objeto o función solo dentro del mismo fichero. Los objetos con el mismo nombre en otros ficheros son objetos distintos. Por tanto, esto equivale a tener visibilidad de fichero.

- Sin enlazado.

  Cada identificador sin enlazar representa unidades únicas. Los objetos con el mismo nombre en otros bloques son objetos distintos. Por tanto, esto equivale a tener visibilidad de bloque. Identificadores sin enlazado son:

  - Cualquier identificador distinto a un objeto o función.
  - Parámetros de funciones.
  - Objetos de ámbito de bloque (entre llaves) sin el especificador `extern`.

3.5. Bibliotecas

Definición 3.14 (Biblioteca)**.** Colección de objetos, normalmente relacionados entre sí. Favorecen modularidad y reusabilidad de código.

Las bibliotecas se pueden clasificar según la forma en la que se enlazan:

1. Bibliotecas Estáticas

   Tienen extensión `.a`, y que se ligan con el programa en el proceso de enlazado. El archivo ejecutable las contiene.

   Algunos inconvenientes que tiene son:

   - El código de la biblioteca está en todos los ejecutables que la usan, lo que desperdicia disco y memoria principal.
   - Si actualizamos una biblioteca estática, debemos recompilar los programas que la usan para que se puedan beneficiar de la nueva versión.
   - Producen ejecutables grandes.

2. Bibliotecas Dinámicas

   Por norma general, tienen extensión `.so` y se integran con los procesos en tiempo de ejecución. En el proceso de montaje, se incluye un módulo de montaje dinámico (*enlazador dinámico*) que se encarga de cargar y montar las bibliotecas dinámicas usadas por el programa durante su ejecución.

El archivo correspondiente a una biblioteca dinámica se diferencia de un archivo ejecutable en los siguientes aspectos:

a) Contiene información de reubicación.

b) Contiene una tabla de símbolos propios de la biblioteca.

c) En la cabecera no se almacena información de punto de entrada.

4. Introducción a las bases de datos.

Definición 4.1 (Directorio)**.** Archivo especial que permite agrupar archivos según las necesidades de los usuarios.

Definición 4.2 (Archivo)**.** Un archivo es un conjunto de información respecto de un mismo tema. Está compuesto de registros homogéneos que contienen información sobre el tema.

Definición 4.3 (Registro)**.** Estructura dentro del archivo que contiene la información correspondiente a un elemento individual. Se divide en campos.

Definición 4.4 (Campo)**.** Dato que representa una información unitaria o independiente.

Clasificación de archivos según el tipo de registros

1. Longitud fija.
2. Longitud variable.
   - Con delimitador. Estos suelen ser caracteres especiales como el salto de línea.
   - Con cabecera: cada registro contiene un campo inicial que almacena el número de bytes del registro.
3. Longitud indefinida.

   El sistema operativo 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 ese registro.

4.1. Organización de Archivos

Existen diversas formas de organizar los archivos:

4.1.1. Organización secuencial

Los registros están almacenados físicamente de forma contigua.

Tiene como ventajas que es sencilla de usar y usa el espacio de una forma óptima. Además, permite leer los registros de forma secuencial de forma óptima.

- Añadir registros: Solo se puede añadir al final.
- Consultar registros: Para consultar el registro n, es necesario leer los $n-1$ anteriores.
- Modificación de registros: Solo es posible si no se aumenta su longitud
- Eliminación de registros: 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.

4.1.2. Organización secuencial encadenada

También llamado "celdas enlazadas". 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.

Permite insertar y borrar registros con facilidad, pero las consultas solo pueden realizarse de forma secuencial.

- Añadir registros: Tras localizar la posición en la que se desea intertar, se siguien los siguientes pasos:

  1. Escribir el registro en una zona libre de memoria.
  2. Modificar la dirección a la que apunta el registro anterior al creado.
  3. Modificar la dirección a la que apunta el creado al siguiente.

- Consultar registros: Es secuencial. Al leer cada registro se lee la posición del siguiente.
- Modificación de registros: 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.

- Eliminación de registros: Se asigna al puntero del registro anterior la dirección del registro siguiente.

4.1.3. Organización secuencial indexada

Está formado por dos estructuras: zona de registros y zona de índices.

- Zona de registros:

  Zona donde se encuentras los registros del archivo de forma secuencial. Está dividida en tramos, y es necesario que los registros estén ordenados según el valor de una llave.

- Zona de índices: Zona en la que por cada tramo de la zona de registros hay un registro que contiene:

  - La dirección del primer registro del tramo.
  - La llave del último registro del tramo. Al estar los registros ordenados, esta es la mayor llave del tramo.

La gestión de la estructura la realiza el sistema operativo 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.

Las operaciones sobre archivos se describen a continuación:

- Consulta de registros: Usando la zona de índices se localiza el tramo, y posteriormente se realiza la búsqueda secuencial en dicho tramo.

- Inserción, borrado y modificación: no son posibles.

4.1.4. Organización directa o aleatoria

Es un archivo para el cual existe una función de transformación que genera la dirección de cada registro 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. Este tipo de organización es útil para archivos donde los accesos deben realizarse por llave.

Las distintas funciones de transformación o métodos de direccionamiento son:

1. Direccionamiento directo Solo es factible cuando la llave es numérica y su rango de valores no es mayor que el rango de direcciones en el archivo.

   La dirección es la propia llave.

   Como inconveniente, se puede dar el caso de direcciones sin usar.

2. Direccionamiento asociado 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.

3. Direccionamiento calculado o *"hashing"*: La dirección de cada registro se obtiene realizando una transformación sobre la llave. Se usa cuando:

   - La llave no es numérica. En ese caso, se necesita una conversión previa para obtener un número a partir de ella. Por ejemplo se usa el equivalente decimal al código ASCII del carácter, y ya con dicho código se opera.

- La llave es numérica pero toma valores en un rango inadecuado para usarse directamente como dirección.

Es fundamental elegir adecuadamente la función de transformación o método de direccionamiento, ya que se pueden dar los siguientes casos:

1. Que haya direcciones que no se corresponden con ninguna llave y, por tanto, habrá zonas de disco sin utilizar.

2. 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**.

El problema de los sinónimos se resuelve de la siguiente forma:

1. Se reserva una zona de desbordamiento gestionada de otra forma (secuencial) donde se guardan los registros cuya dirección ya está ocupada por una clave sinónima.

2. Los registros cuya dirección ya está ocupada por una clave sinónima se guardan en la próxima dirección de memoria libre.

Las operaciones sobre archivos se describen a continuación:

- Consulta de registros: Por llave, aplicándole la transformación correspondiente. En el caso de que no se encuentre en su dirección, se procede según se haya resuelto el problema de sinónimos.

- Borrado: Ha de ser lógico.

- Inserción y modificación: Es posible, realizando la transformación de la llave correspondiente.

4.2. Bases de datos

Los problemas asociados a las gestiones de archivos mencionadas son:

1. Dificultad de mantenimiento. Si hay dos archivos con información duplicada, si se modifica uno ha de moficarse el otro. En caso contrario, habría información incoherente.

2. Redundancia. Hay redundancia si un dato se puede deducir a partir de otros datos, y estaríamos en el caso anterior.

3. Rigidez de búsqueda. La búsqueda ha de realizarse de una forma concreta, sin poder combinarse directa y secuencial.

4. Dependencia con los programas Si un programa lee dichos archivos y se modifican los archivos, ha de modificarse también el programa.

5. Seguridad.

Para resolver dichos problemas, surgen las bases de datos.

Definición 4.5 (Base de Datos)**.** Sistema formado por un conjunto de datos y un paquete software para gestión de dicho conjunto de datos de tal modo que:

1. Se controla el almacenamiento de datos redundantes.
2. Los datos resultan independientes de los programas que los usan.
3. Las relaciones entre los datos se almacenan junto a ellos.
4. Se puede acceder a los datos de diversas formas.

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

Definición 4.6 (Clave primaria)**.** 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 unívoca cada uno de los elementos de dicha entidad, y no existe ningún subconjunto de él que permita identificar a la entidad de manera única.

5. Generación y Depuración de Aplicaciones

5.1. Plataformas

Definición 5.1. Combinación de hardware y/o software para ejecutar aplicaciones software.

Ejemplos de plataformas son los sistemas operativos (Windows, Mac, etc.) o los navegadores.

La clasificación del software se realiza según la plataforma en la que se puede ejecutar.

1. Dependiente de la plataforma particular para la cual se desarrolla y ejecuta (bien sea una plataforma hardware, sistema operativo o máquina virtual).

2. Multiplataforma, cuando el software se ha desarrollado y opera en varias plataformas.

   El software multiplataforma se puede clasificar en dos tipos:

   a) Aplicaciones que requieren su creación o compilación para cada plataforma específica donde se ejecutarán.

   b) Aplicaciones que directamente se pueden ejecutar en más de una plataforma sin preparación especial.

   Ejemplo de esto son los programas en Java o aplicaciones escritas en lenguaje interpretado solo usando los paquetes estándar.

5.1.1. Plataforma JAVA

Tiene un alto nivel de abstracción, ya que incluso incluye un lenguaje de programación. Requiere la instalación de JVM *(Java Virtual Machine)* en cada SO. Por tanto, los programas Java son multiplataforma, pero no la JVM (hay una para cada sistema operativo).

Los programas no se ejecutan directamente en el sistema operativo, sino en la JVM. No obstante, dispone de la *Java Native Interface (JNI)* para cceder a funciones específicadas de cada SO.

Hay un compilador *JIT (Just In Time)* que traduce instrucciones en Java a instrucciones nativas del procesador. Esto permite que las aplicaciones JAVA se ejecuten de forma rápida.

5.1.2. Plataforma Android

La arquitectura de Android está formada por distintas capas, todas ellas basadas en software libre. Cada una de las capas utiliza elementos de la capa inferior para realizar sus funciones, concepto que se conoce como pila de sofware. Estas capas, de más inferior a superior, son:

1. Kernel de Linux: Administración de memoria de bajo nivel, gestión de subprocesos, etc.
2. Capa de abstracción de hardware *(HAL, Hardware Abstraction Layer)*: Módulos de biblioteca para un tipo específico de componente de hardware, como dispositivos Bluetooth.
3. Tiempo de ejecución de Android *(ART, Android Runtime)*: Ejecuta varias máquinas virtuales que optimizan los procesos.
4. Bibliotecas C/C++ nativas. Muchos componentes y servicios centrales del sistema Android, como el ART y la HAL, se basan en código que requiere bibliotecas nativas escritas en C/C++.
5. Marco de trabajo de la API de Java. El conjunto de funciones del SO Android está disponible mediante API escritas en Java.
6. Apps del sistema: Apps centrales como un navegador, email, calendario, etc.

5.2. Framework de Desarrollo de Aplicaciones

5.2.1. Herramientas básicas para el desarrollo software en Linux

Linux tiene herramientas de ayuda en cada fase del desarrollo de las aplicaciones:

1. Generador de código fuente: Editores de texto como `gedit`, `nano`, etc.
2. Sangrado código fuente: `indent` sangra un programa en C sintácticamente correcto.
3. Compilación código fuente: Tiene compiladores como `g++` o `gcc`.
4. Gestión de software basado en módulos: Para ello está la utilidad `make`.
5. Gestión de bibliotecas
6. Control de versiones

Definición 5.2 (IDE)**.** Un entorno integrado de diseño (IDE, *Integrated Development Environment*) es aplicación que proporciona un conjunto de herramientas relacionadas para el desarrollo del software. Permite:

- Crear el código

- Gestionar el ciclo de vida. Compilar, enlazar, etc.
- Despliegue (instalación) y depuración del código.

La comodidad que aportan contrasta con el uso de las herramientas aisladas en Linux.

Algunas características de los IDE a destacar son:

- Maximizar la productividad, ya que incluye diversas herramientas.
- Acelerar el aprendizaje de lenguajes de programación; ya que por ejemplo el código se puede analizar mientras se edita para conocer de forma inmediata errores léxicos, sintácticos, etc.

Definición 5.3 (Programación Visual)**.** Hace uso de IDEs que permiten a los programadores crear nuevas aplicaciones combinando bloques de código con diagramas de flujo, normalmente basados en UML.

5.2.2. Framework de desarrollo software

Definición 5.4 (Framework de desarrollo software)**.** Abstracción usada para crear aplicaciones que proporciona software con funcionalidad genérica que el usuario puede modificar mediante código.

Por ejemplo, algunos de los software genéricos que pueden incluir son la gestión de la base de datos, la autenticación de usuarios o la generación de interfaces de usuario, entre otras.

Tienen como objetivo facilitar y reducir el tiempo el desarrollo evitando dedicar tiempo a detalles de bajo nivel, pero debido a la complejidad de las APIs conllevan un tiempo extra de aprendizaje inicialmente.

5.3. Técnicas de Depuración de Programas

Definición 5.5 (Depuración)**.** Actividad consistente en encontrar y solucionar errores cometidos en el diseño y codificación de programas y solucionarlos.

Los objetivos de la depuración son:

- Incrementar la productividad con una detección de errores más rápida.
- Mejorar la calidad de los programas.
- Prevenir errores.
- Mejorar lenguajes de programación y herramientas.

La depuración consiste en la ejecución de programas como máquinas de estados. Cada **estado** viene determinado por el valor de las variables y la posición de ejecución, por lo que cada estado depende de los anteriores.

Las fases de la generación del fallo son:

1. Creación de un defecto: Pieza de código creada por el programador que *puede* causar infección como consecuencia de un error.
2. El defecto produce infección: Los estados alcanzados difieren de los previstos.
3. Propagación de la infección
4. La infección causa el fallo: Error observable externamente en el comportamiento de un programa que detecta el programador.

De esto se deduce que la ausencia de fallos nunca asegura la ausencia de defectos.

Definición 5.6 (Cadena de infección)**.** Todas las instrucciones que se han ejecutado desde que se produjo el defecto hasta que se ha detectado el fallo. Al depurar, buscamos retroceder en la cadena de infección hasta localizar el fallo.

Las fases de la depuración son:

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.

Los errores más comunes son escribir código desestructurado y programar sin pensar. Las características del código estructurado son:

- Uso de funciones.
- Uso de construcciones iterativas (`while`, `for`) en vez de `goto`.
- Uso de variables con nombre significativo.
- Uso de tipos de dato abstracto *(ADTs, Abstract data types)* o clases y objetos.

Definición 5.7 (Depurador)**.** Herramienta software que ayuda a ejecución controlada de un programa para ayudar a encontrar el defectos que producen un fallos.

Las características de los depuradores son:

- Permitir interacción con el programador.
- Proporciona un conjunto de instrucciones propias.
- Permite detectar errores en tres categorías:
    - Sintácticos. Detectados durante la compilación o enlazado.

- Ejecución. Errores que provocan el fin del programa. Dividir entre 0, acceder fuera del espacio de memoria reservado (*segmentation fault*), etc.
- Lógicos. Errores que debe detectar el programador, que no generan fallos en ejecución pero que muestran que el algoritmo no está bien implementado.

6. Relaciones de ejercicios

6.1. Sistema de Cómputo

Ejercicio 6.1.1. El método de comunicación de E/S en el que la CPU está esperando hasta que la operación de E/S ha finalizado se conoce como:

(a) E/S Programada.

(b) E/S Dirigida por Interrupciones.

(c) DMA.

(d) E/S a Distancia.

Ejercicio 6.1.2. El método de comunicación de E/S en el que el dispositivo de E/S informa a la CPU en qué momento está preparado el dispositivo para la transferencia de datos se conoce como:

(a) E/S Programada.

(b) E/S Dirigida por interrupciones.

(c) DMA.

(d) E/S a Distancia.

Ejercicio 6.1.3. Cuál de las siguientes afirmaciones es correcta:

(a) En algunas computadoras un programa puede ejecutarse sin necesidad de cargarlo en la memoria principal.

(b) Un programa, para que se ejecute, debe estar cargado en la memoria principal

(c) Un programa, para que se ejecute, basta con que esté en el disco duro

(d) Un programa, para que se ejecute, si está en lenguaje máquina, puede estar en cualquier unidad.

Ejercicio 6.1.4. Dado el esquema de un computador elemental según se ha descrito en el tema, el puntero de pila (SP) indica:

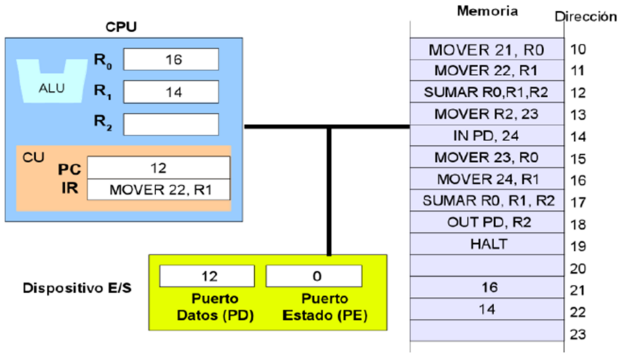
(a) La dirección de memoria donde debe saltar el programa después de ejecutarse la instrucción de retorno correspondiente.

(b) La dirección de memoria donde se encuentra la dirección donde debe saltar el programa después de ejecutarse la instrucción de retorno correspondiente[1].

(c) La dirección de memoria a donde se ha producido el último salto.

(d) La dirección de memoria donde se encuentra la dirección a donde se ha producido la última llamada a una subrutina.

Ejercicio 6.1.5. Sea un ordenador elemental con una arquitectura tal y como se muestra en la figura 6.1, es decir, tres registros de propósito general, registro contador de programa (PC) y registro de instrucción (IR). El registro SP (Puntero de pila) contiene la dirección 35 y la pila crece hacia posiciones menores de memoria. La memoria principal dispone de 256 palabras donde cada palabra tiene la longitud necesaria para albergar la instrucción de mayor tamaño. Describa el estado final de ejecución del procesador a partir del estado actual de la CPU mostrado en la figura 6.1. Ponga todos los valores de los registros de cada ciclo de instrucción realizado por el procesador hasta llegar a dicho estado final.



Instrucción	Descripción
`MOVER Orig,Dest`	Copia el valor del origen (Orig) al destino (Dest).
`SUMAR Ri,Rj,Rk`	Suma el valor de Ri y Rj, depositando el resultado en Rk.
`IN Pi, Dest`	Lee del Puerto (Pi) y lo deposita en el destino (Dest).
`OUT Pi, Orig`	Escribe el contenido del origen (Orig) en el puerto Pi.
`HALT`	Detiene al procesador.

Figura 6.1: Computador del ejercicio 6.1.5

.

El estado del computador en cada ciclo es:

[1]En realidad apunta a la dirección de memoria anterior.

PC	IR	SP	R0	R1	R2	PE	PD
12	MOVER 22, R1	35	16	14		0	12
13	SUMAR R0,R1,R2	35	16	14	30	0	12
14	MOVER R2,23	35	16	14	30	0	12
15	IN PD 24	35	16	14	30	0	12
16	MOVER 23, R0	35	30	14	30	0	12
17	MOVER 24, R1	35	30	12	30	0	12
18	SUMAR R0,R1,R2	35	30	12	42	0	12
19	OUT PD, R2	35	30	12	42	0	42
20	HALT	35	30	12	42	0	42

Tabla 6.1: Ejecución del programa del ejercicio 6.1.5

Memoria	
	⋮
16	21
14	22
30	23
12	24
	⋮

Tabla 6.2: Memoria del ejercicio 6.1.5

Ejercicio 6.1.6. Suponiendo que el lenguaje máquina de la arquitectura anterior dispone de 14 instrucciones distintas, muestre cuántos bits serían necesarios para codificar las instrucciones SUMAR R0,R1,R2 y MOVER 20,R0 respectivamente.

Puesto que dispone de 14 instrucciones distintas, se necesitaran 4 bits para codificarlas.

- **SUMAR R0, R1, R2:** Son necesarios además 2 bits para codificar cada uno de los 3 registros. Luego $4 + 3 \cdot 2 = 10$ bits
- **MOVER 20, R0:** Son necesarios además 2 para el registro R0. Para codificar la dirección, puesto que hay $256 = 2^8$ palabras de memoria principal, se necesitan 8 bits por dirección. Luego $4 + 2 + 8 = 14$ bits

Ejercicio 6.1.7. Imagina que el procesador está ejecutando el programa de usuario del ejercicio 6.1.5 y en este momento al terminar de ejecutar la instrucción actual, el procesador se da cuenta de que hay una interrupción pendiente. Escribe los pasos que se dan en el sistema y por quién (software o hardware) hasta que se resuelve el tratamiento de la interrupción y el programa finaliza, sabiendo que la rutina de tratamiento de la interrupción comienza en la dirección de memoria principal 56 y termina en la dirección de memoria principal 70.

En primer lugar, a nivel de hardware, se da:

1. El procesador indica el reconocimiento de la interrupción.

2. El procesador apila PSW y el PC en la pila de control. Es decir:

$$M[35] = PC = 12 \quad M[34 - n] = PSW \quad SP = 33 - n$$

3. El procesador carga un nuevo valor en el PC basado en la interrupción. En este caso, como la interrupción comienza en la dirección 56, $PC = 56$.

A nivel de software, se da:

4. Se salva el resto de la información de estado del proceso.

5. Se procesa la interrupción. En este caso, se ejecutan las instrucciones desde la posición 56 a 70. Es decir, al terminar, $PC = 71$.

6. Se restaura la información de estado del proceso.

7. Se restauran los valores de PSW y PC y se actualiza el puntero de pila. Es decir,

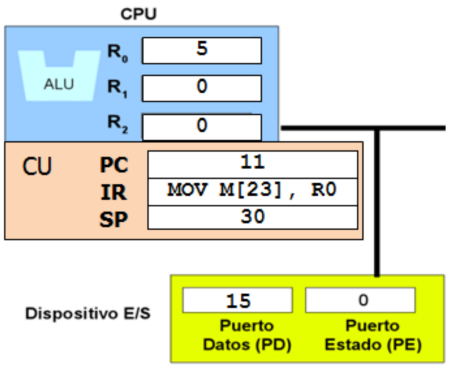
$$PSW = M[34 - n] \quad PC = M[35] = 12 \quad SP = 35$$

Ejercicio 6.1.8. Basándonos en el ejercicio 6.1.7, ¿hay diferencias si en vez de producirse una interrupción se ha producido una excepción? Indique cuales.

Sí, hay diferencias. En primer lugar, las excepciones son un evento síncrono, mientras que las interrupciones son asíncronas. El procesador aquí se ha dado cuenta al finalizar la ejecución de que había una interrupción pendiente, mientras que en el caso de una excepción se ejecuta directamente. Además, por norma general, una excepción conlleva un posible error en el código y cómo resolverlo, mientras que las interrupciones no suelen estar asociadas a errores.

Ejercicio 6.1.9. Sea un ordenador elemental con una arquitectura tal y como se muestra en la figura 6.2, es decir, tres registros de propósito general, registro contador de programa (PC), registro de instrucción (IR) y registro de pila (SP). La memoria principal dispone de 512 palabras donde cada palabra tiene la longitud necesaria para albergar la instrucción de mayor tamaño. Describa el estado final de ejecución del procesador a partir del estado actual de la CPU mostrado en la figura y tras la ejecución del programa (nótese que la instrucción de la dirección 10 ya se ha ejecutado).

Instrucción	Descripción
MOV M[N],Ri	Copia el valor de la dirección de memoria N al registro Ri
COMP Ri,Rj	Si Ri == Rj, activa el bit de estado. En otro caso, lo desactiva
IN Pi,M[N]	Lee del Puerto (Pi) y lo deposita en dirección de memoria N.
JNE N	Si el bit de estado no está activo, salta a la dirección de memoria N.
OUT Ri,Pi	Escribe el contenido del registro Ri en el puerto Pi.
HALT	Detiene al procesador
CALL N	Guarda el PC en la pila y salta a la dirección de memoria N.
RET	Saca un elemento de la pila y lo almacena en PC.
ADD Ri,Rj,Rk	Rk= Ri+Rj



Memoria	Dir
MOV M[23],R0	10
IN PD,M[26]	11
MOV M[26],R1	12
COMP R0,R1	13
JNE 18	14
MOV M[24],R0	15
OUT R0,PD	16
HALT	17
CALL 20	18
HALT	19
ADD R0,R1,R2	20
OUT R2,PD	21
RET	22
5	23
0	24
1	25
	26
	...
	30

Figura 6.2: Computador del ejercicio 6.1.9

El estado del computador en cada ciclo es:

PC	IR	SP	R0	R1	R2	PE	PD
11	MOV M[23], R0	30	5	0	0	0	15
12	IN PD, M[26]	30	5	0	0	0	15
13	MOV M[26], R1	30	5	15	0	0	15
14	COMP R0, R1	30	5	15	0	0	15
~~15~~ 18	JNE 18	30	5	15	0	0	15
~~19~~ 20	CALL 20	29	5	15	0	0	15
21	ADD R0, R1, R2	29	5	15	20	0	15
22	OUT R2, PD	29	5	15	20	0	20
~~23~~ 19	RET	30	5	15	20	0	20
20	HALT	30	5	15	20	0	20

Tabla 6.3: Ejecución del programa del ejercicio 6.1.9

Memoria	
	10
	⋮
	22
5	23
0	24
1	25
15	26
	⋮
19	30

Tabla 6.4: Memoria del ejercicio 6.1.9

6.2. Introducción a los Sistemas Operativos

Ejercicio 6.2.1. Sea un proceso en un SO con su información de contexto, de datos y de código según se muestra en la figura 6.5 y que ya ha sido atendido en un 50 % y le resta la otra mitad para finalizar su ejecución. Con la idea de optimizar el espacio de memoria para que el SO pudiera disponer de un mayor número de procesos en ésta, ¿podría reducirse el espacio que ocupa en memoria en alguna de las siguientes instancias?

Código	12922 KB
Pila	3002 KB
Datos	434 KB

Tabla 6.5: Información del proceso del ejercicio 6.2.1

(a) La lista de procesos.

El proceso aún no ha terminado, por lo que no es posible eliminarlo de la lista de procesos. Sí es cierto que los procesos que ya hayan terminado sí podrían eliminarse de la lista de procesos para liberar memoria; pero como este proceso no ha terminado no puede eliminarse de la lista de procesos.

(b) Información del contexto del proceso.

El contexto del proceso contiene información importante acerca de este, como todos los registros de control o el PC, por ejemplo. Por tanto, no se puede liberar espacio aquí, ya que se perdería información necesaria para poder continuar con el proceso.

(c) Tamaño de los datos.

El proceso no ha terminado, por lo que a priori, sin tener más información del proceso, no podemos eliminar datos que no sabemos si serán necesarios en un futuro. Por tanto, tampoco es posible liberar espacio aquí.

(d) Tamaño del código.

El proceso no ha terminado, por lo que si redujésemos el código podríamos caer en el error fatal de que el programa no pudiese continuar, ya que podríamos haber eliminado instrucciones necesarias para el programa.

El único caso excepcional en el que se podría eliminar parte del código es que el programa no incluyese instrucciones del salto, y justo en este caso sí podríamos quitar parte del programa del principio. No obstante, no sabríamos cuántas instrucciones liminar a priori.

Ejercicio 6.2.2. ¿Por qué cuando un proceso está en modo "ejecutándose" y pretende acceder a una dirección de memoria fuera del área asignada, se informa de que se ha producido un error en la ejecución? ¿Quién informa de ello? Razone la respuesta.

El SO se encarga de asignar a cada proceso solo una parte de la memoria a la que puede acceder, ya que en caso contrario se correría el riesgo de que un proceso

modificase aspectos importantes en la memoria. Por tanto, no está permitido hacer lo indicado.

Hay un dispositivo hardware (MMU, *Memory Management Unit*) encargado de lanzar la excepción correspondiente a este error. La rutina de tratamiento de dicha excepción es la que informará al usuario después de abortar el proceso.

Ejercicio 6.2.3. ¿Tiene sentido un modelo de 5 estados de los procesos en un SO monousuario? Razone la respuesta.

Si, ya que aunque haya solo un usuario, este puede querer ejecutar simultáneamente más de un proceso.

Ejercicio 6.2.4. Dado un proceso que está en modo "ejecutándose" y pretende acceder a una dirección de memoria fuera del área asignada, lo cual sería un error en la ejecución, ¿a qué modo pasaría dicho proceso? Razone la respuesta.

(a) Bloqueado.

(b) No cambia de modo.

(c) Finalizado.

(d) Preparado.

Pasaría a modo finalizado, ya que es un error no recuperable. El programa no podrá continuar con su ejecución, y el SO decidirá abortarlo tras el mensaje de error aportado por el MMU.

Ejercicio 6.2.5. Un planificador de procesos tiene una tarea concreta dentro de un SO multiprogramado. ¿Tiene sentido disponer de un planificador de procesos en un SO monoprogramado? Razone la respuesta.

No, ya que solo puede ejecutar un programa a la vez. La utilidad del planificador de procesos es cambiar de uno a otro para que la CPU no esté sin realizar cálculos. Sin embargo, si solo hay un programa pierde su utilidad.

Ejercicio 6.2.6. Dado un SO multiprogramado, ¿bajo qué circunstancias se podría prescindir del planificador de procesos? Razone la respuesta.

A pesar de ser multiprogramado, cabe la posibilidad de que solo se estén ejecutando, como máximo, tantos procesos como procesadores hay. En ese caso, no sería necesario el planificador de procesos.

No obstante, esto es una situación excepcional y, por norma general, es necesario.

Ejercicio 6.2.7. Diga cuales de las siguientes operaciones pueden realizarse únicamente en modo supervisor, o modo kernel:

(a) Consultar la hora del sistema.

No es necesario, ya que no afecta al resto de procesos.

(b) Cambiar la fecha del sistema.

Sí es necesario el modo kernel; ya que otros procesos, como la alarma, dependen de la hora.

(c) Leer una pista/sector de un disco magnético.

Sí es necesario, ya que es una operación de muy bajo nivel y actúa directamente el SO.

(d) Generar una interrupción software.

Las interrupciones software se pueden generar desde ambos modos, tanto modo kernel como modo usuario.

(e) Generar una interrupción harware.

En este caso, al estar referidas a dispositivos hardware externos como podrían ser los de Entrada/Salida, tenemos que estos pueden lanzar la excepción indepedietemente del modo en el que se esté.

(f) Modificar la dirección de un vector de la tabla de vectores de interrupción.

En este caso es claramente necesario el modo kernel, ya que si se modificase de forma inadecuada no se podría llamar correctamente a dicha interrupción.

(g) Deshabilitar las interrupciones.

En este caso también es evidente la necesidad de hacerlo desde el modo kernel.

Ejercicio 6.2.8. En el caso de un ordenador que se vaya a usar únicamente para un único usuario, ¿qué interés puede tener la existencia de los modos de funcionamiento supervisor/usuario?

El principal interés de esta separación del funcionamiento del sistema en dos modos reside en que el usuario puede, de cierta manera, estar tranquilo, en tanto que sabe que no es capaz de hacer nada que resulte crítico para el correcto funcionamiento del sistema (como borrar archivos que contengan información sobre el arranque, por ejemplo).

Además, de ser el ordenador multiprogramado, es necesario que los procesos no interfieran entre ellos y que no se perjudiquen mutuamente. El modo kernel en este caso se encarga de que, por ejemplo, un proceso no pueda modificar la dirección de un vector en la tabla de vectores de interrupciones y así no perjudique al resto de procesos.

Así, con esta separación ganamos en seguridad y control para nuestro sistema.

Ejercicio 6.2.9. Cuestiones sobre procesos, y asignación de CPU:

1. ¿Es necesario que lo último que haga todo proceso antes de finalizar sea una llamada al sistema para finalizar? ¿Sigue siendo esto cierto en sistemas monoprogramados?

   Sí, es necesario ya que, en primer lugar, se debe realizar un cambio de contexto para que el proceso finalizado pase a dicho estado, mientras que un estado que se encontrase en "Preparándose" pase ahora al modo de ejecución.

Además, también es necesario para que se libere la memoria que ocupaba el programa finalizado, para así poder cargar en memoria un número mayor de programas.

En un sistema monoprogramado, si no se avisa al terminar un programa mediante una llamada al SO; entonces el procesador no va a pasar a otro proceso en ningún momento al no saber si se ha liberado la memoria correspondiente.

2. Cuando el controlador de un dispositivo produce una interrupción ¿se produce necesariamente un cambio de contexto?, ¿y cuando se produce una llamada al sistema?

   En ambos casos, se realizará un cambio de modo a superusuario, ya que el sistema operativo se tendrá que encargar de gestionar tanto de la interrupción como de la llamada al sistema.

   No obstante, no necesariamente ocurrirá un cambio de contexto; de hecho, esta ventaja de poder ejecutar rutinas del SO sin un cambio de contexto en el proceso ejecutándose es muy importante a la hora del diseño de ciertos tipos de Sistemas Operativos (aquellos que no están totalmente implementados con procesos). Sin embargo, cabe la posibilidad de que la rutina de tratamiento de la interrupción o llamada al sistema explícitamente ejecute un cambio de contexto.

3. Cuando un proceso se bloquea, ¿deberá encargarse él directamente de cambiar el valor de su estado en el descriptor de proceso o PCB?

   No. El bloqueo se deberá a alguna interrupción o llamada al sistema, por lo que el SO se encargará de realizar el cambio de contexto. En este cambio, se actualizarán los estados de los procesos implicados (el que pasa a "Bloqueado" y el que pasa a "Ejecutándose") y se modificará por tanto el PCB.

4. Sea un proceso que cambia de Ejecutándose a Bloqueado, ¿puede este cambio provocar un cambio de estado en otros procesos? Si es así, ¿en qué casos?

   Sí, ya que se llevará a cabo un cambio de contexto y uno de los procesos que estaba en estado de "Preparado" pasará a ejecutarse, siempre y cuando los recursos estén disponibles (por ejemplo, si todos los procesos preparados requieren de la impresora, y esta está ocupada, es necesario esperar).

5. Idem para el cambio de estado Bloqueado a Ejecutable.

   Al realizarse el cambio de estado de un proceso de "Bloqueado" a "Preparado", es posible que simplemente entre en la cola de los procesos preparados para su ejecución y que, por tanto, no provoque cambios en los demás. No obstante, si el planificador de procesos decide que el proceso que ha entrado tiene una prioridad más alta que el que se estaba ejecutando, es posible que pase este de "Preparado" a "Ejecutándose", mientras que el que se estaba ejecutando y era menos importante pase a estado "Bloqueado".

Ejercicio 6.2.10. En los primeros ordenadores, cada byte de datos leído o escrito, era manejado directamente por la CPU (es decir, no existía DMA Acceso Directo

a Memoria). ¿Qué implicaciones tenía esta organización para la multiprogramación?

Al no disponer de un módulo independiente a la CPU que gestionase las operaciones en Entrada/Salida, el procesador estaría ocupado durante todo el desarrollo de éstas, que por norma general suelen ser muy lentas.

Por tanto, en el caso de un sistema monoprogramado, durante el tiempo de ejecución de esta entrada y salida de datos el procesador estaría completamente ocioso.

En el caso de la multiprogramación, aunque se podrían ejecutar otros procesos durante la lectura, en ese tiempo se dispondría de menos capacidad de la CPU; por lo que la cantidad de programas que se podrían ejecutar haciendo uso de multiprogramación se vería bastante limitada. Por tanto, la principal ventaja de la multiprogramación se vería muy limitada.

Ejercicio 6.2.11. ¿Por qué no es el intérprete de órdenes (shell) parte del propio sistema operativo? ¿Qué ventajas aporta el no serlo?

La independencia del intérprete de órdenes respecto al SO supone diferentes ventajas, ya que siempre se busca minimizar el SO a lo exclusivamente esencial por diferentes motivos. Algunos de los beneficios de reducir el SO al mínimo son:

- Un fallo en la ejecución del shell no supondrá un fallo a nivel global en el SO, por lo que este se podrá encargar de gestionar dicho fallo. Los errores del sistema operativo suelen ser fatales, ya que no son de fácil gestión por él mismo.
- Al cargar lo mínimo posible en el sistema operativo, tenemos que al arrancar el ordenador este proceso de ejecuta de una manera más rápida y simple. Por tanto, esto aporta velocidad, ligereza y simpleza.
- Aporta versatilidad, ya que se puede actualizar el shell sin necesidad de actualizar simultáneamente el SO.
- El shell no puede realizar operaciones en modo kernel; algo que sí puede realizar el SO y podría provocar errores si no se gestiona adecuadamente.

Ejercicio 6.2.12. Para cada una de las llamadas al sistema siguientes, especificar y explicar si su procesamiento por el sistema operativo implica un cambio de contexto:

1. Crear un proceso.

   Se produce un cambio de contexto cuando un proceso que estaba ejecutándose pasa a otro estado, mientras que uno que estaba preparado pasa a ejecutarse.

   En este caso, si solo se crea el proceso no es necesario el cambio de contexto, ya que pasará a estado preparado pero no empezará a ejecutarse.

   No obstante, en el caso puntual en el que al crearlo tenga más prioridad que algún proceso que se esté ejecutando, ese proceso pasará a estar preparado; mientras que el que se acaba de crear pasaría a ejecutarse. En este caso, por tanto, sí se produciría el cambio de contexto.

2. Abortar un proceso, es decir, terminarlo forzosamente.

   Sí, en este caso se produce un cambio de contexto, ya que al terminarlo forzosamente se liberan los recursos que estaban siendo ocupados por él. Un proceso que estuviese en estado de "Preparado" pasaría a ejecutarse gracias a esa liberación de recursos. Por tanto, se produciría un cambio de contexto.

3. Suspender o bloquear un proceso.

   Sí, ya que el proceso pasaría de estado "Ejecutándose" a otro estado, por lo que se liberarían recursos que serían empleados para ejecutar un proceso que estaba preparado en espera. Por tanto, se produce un cambio de contexto.

4. Reanudar un proceso (inverso al caso anterior).

   También se produce un cambio de contexto, ya que un proceso pasa de estar "Preparado" a ejecutarse.

5. Modificar la prioridad de un proceso.

   No necesariamente implica un cambio de contexto.

   Si el cambio en la prioridad de un proceso que se esté ejecutando o de uno que esté en espera implica que la prioridad del que está en espera sea mayor que la del que se está ejecutando, entonces la CPU desasignará los recursos del proceso que se estaba ejecutando y los asignará al que estaba preparado. Es decir, un proceso que estaba en ejecución a pasado a estar preparado, mientras que uno preparado ha pasado a ejecutarse, por lo que se ha producido un cambio de contexto.

   No obstante, si el cambio en la prioridad no implica que la prioridad de un proceso en espera sea mayor que la de los que se están ejecutando, entonces no se producen cambios de contexto.

Ejercicio 6.2.13. ¿Tiene sentido mantener ordenada por prioridades la cola de procesos bloqueados? Si lo tuviera, ¿en qué casos sería útil hacerlo?

Depende del motivo por el cual estén bloqueados. En el caso de que estén bloqueados por motivos distintos, no tiene sentido ya que volverán a ejecutarse en cuanto termine el motivo por el cual bloqueados.

En el caso de que estén bloqueados por el mismo motivo, sí tiene sentido; ya que cabe la posibilidad de que un proceso sea más importante que otro y; por tanto, sea necesario que se ejecute en cuanto se termine el motivo por el cual ha sido detenido. Por ejemplo, si varios procesos están bloqueados en espera de la impresora, tiene sentido ordenarnos para determinar cuál va a poder usarla antes.

Ejercicio 6.2.14. ¿Por qué se utilizan potencias de dos para los tamaños de página, número de páginas en el espacio lógico de un proceso, y números de marcos de página?

Se utilizan potencias de 2 porque un ordenador trabaja internamente con números en sistema binario. Además, las potencias de dos son siempre divisibles en potencias de 2 de menor orden, por lo que se puede dividir una página en páginas de menor tamaño sin que quede espacio sin utilizar o fragmentado, evitando así la fragmentación externa.

Ejercicio 6.2.15. Sitúese en un sistema paginado, en donde la memoria real tiene un tamaño de 16 Mbytes, una dirección lógica ocupa 32 bits, de los cuales los 22 de la izquierda constituyen el número de página, y los 10 de la derecha el desplazamiento dentro de la página. Según lo anterior,

1. ¿Qué tamaño tiene cada página?

   El tamaño de página es $2^{10}\ B = 1024\ B = 1\ KB$. Es en bytes porque la unidad mínima en gestión de memoria es el byte.

2. ¿En cuántos marcos de página se divide la memoria física?

   Como el tamaño del marco de página es el mismo que el tamaño de la página, tengo que el tamaño del marco de página es 1024 B.

   Por tanto, el número de marcos de página es:

   $$\frac{16\ MB}{1024\ B} = \frac{16 \cdot 2^{20}\ B}{1 \cdot 2^{10}\ B} = 16 \cdot 2^{10} = 2^{14} \text{ marcos de página.}$$

3. ¿Qué tamaño deberá tener el campo **Número de Marco** de la Tabla de Páginas?
   Como tenemos 2^{14} marcos de página, necesitamos 14 bits para representar todos. Por tanto, el tamaño ha de ser 14 bits.

4. Además de dicho campo, suponga que la Tabla de Páginas tiene los siguientes campos con los siguientes valores:

   Protección: 1 bit (1=Sólo se permite leer; 0=Cualquier tipo de acceso).

   ¿Cuál es el tamaño de la Tabla de Páginas para un proceso cuyo espacio de memoria lógico es de 103 KB?

   Como el tamaño de cada página es de 1 KB, necesitamos 103 páginas en total. Como el tamaño de cada página es $14 + 1$ bits, tenemos que el tamaño de la tabla de páginas es:

   $$103 \times (14 + 1) = 1545\ bits$$

Ejercicio 6.2.16. Suponga que la tabla de páginas para el proceso actual se parece a la de la figura. Todos los números son decimales, la numeración comienza en todos los casos desde cero, y todas las direcciones de memoria son direcciones en bytes. El tamaño de página es de 1024 bytes.

N. Pág. Virtual	N. Marco de Página
0	4
1	7
2	1
3	2
4	10
5	0

Tabla 6.6: Tabla de Páginas para el proceso del ejercicio 6.2.16.

¿Qué direcciones físicas corresponderán con cada una de las siguientes direcciones lógicas del proceso?

1. 999
   Calculamos en primer lugar el número de página, el marco y el desplazamiento. Tenemos que:
   $$999 = 1024 \cdot 0 + 999$$
   Como el cociente de $\frac{999}{1024}$ es 0, tenemos que el número de página es el 0. Equivalentemente, el marco de página es el 4.
   Como el resto de $\frac{999}{1024}$ es 999, tenemos que el desplazamiento es 999.
   Por tanto, la dirección física es:
   $$\begin{aligned}\text{Dir. Física} &= \text{Num. Marco} \times \text{Tam. Marco} \ + \ \text{Desplazamiento} \\ &= 4 \cdot 1024 + 999 = 5095\end{aligned}$$
   Además, la dirección lógica sería $(0, 999)$.

2. 2121
   Calculamos en primer lugar el número de página, el marco y el desplazamiento. Tenemos que:
   $$2121 = 1024 \cdot 2 + 73$$
   Como el cociente de $\frac{2121}{1024}$ es 2, tenemos que el número de página es el 2. Equivalentemente, el marco de página es el 1.
   Como el resto de $\frac{2121}{1024}$ es 73, tenemos que el desplazamiento es 73.
   Por tanto, la dirección física es:
   $$\begin{aligned}\text{Dir. Física} &= \text{Num. Marco} \times \text{Tam. Marco} \ + \ \text{Desplazamiento} \\ &= 1 \cdot 1024 + 73 = 1097\end{aligned}$$
   Además, la dirección lógica sería $(2, 73)$.

3. 5400
   Calculamos en primer lugar el número de página, el marco y el desplazamiento. Tenemos que:
   $$5400 = 1024 \cdot 5 + 280$$
   Como el cociente de $\frac{5400}{1024}$ es 5, tenemos que el número de página es el 5. Equivalentemente, el marco de página es el 0.
   Como el resto de $\frac{5400}{1024}$ es 280, tenemos que el desplazamiento es 280.
   Por tanto, la dirección física es:
   $$\begin{aligned}\text{Dir. Física} &= \text{Num. Marco} \times \text{Tam. Marco} \ + \ \text{Desplazamiento} \\ &= 0 \cdot 1024 + 280 = 280\end{aligned}$$
   Además, la dirección lógica sería $(5, 280)$.

Ejercicio 6.2.17. ¿Qué tipo de fragmentación se produce en un sistema de gestión de memoria paginado? ¿Qué decisiones de diseño se pueden tomar para minimizar dicho problema, y cómo afectan estas decisiones al comportamiento del sistema?

La paginación se realiza en páginas de la misma longitud. Al ejecutar cada proceso, se emplean tantas páginas como sean necesarias según el tamaño del proceso. El problema en este caso es que si el tamaño del proceso no es múltiplo del tamaño de página, se desperdicia memoria, ya que la última página asociada a dicho proceso no estará completa. Esto se denomina **fragmentación interna**.

Para minimizar dicho problema, se podría emplear un tamaño de página menor, para así conseguir que la memoria desperdiciada en cada proceso fuese menor. No obstante, esto provocaría que la tabla de páginas aumentase de tamaño, disminuyendo así la eficiencia del programa.

Ejercicio 6.2.18. Suponga que un proceso emite una dirección lógica igual a 2453 y que se utiliza la técnica de paginación, con páginas de 1024 palabras:

1. Indique el par de valores (número de página, desplazamiento) que corresponde a dicha dirección.

   Tenemos que:

   $$2453 = 1024 \cdot 2 + 405$$

   Como el cociente de $\frac{2453}{1024}$ es 2, y el resto (desplazamiento) es 405, tenemos que la dirección lógica sería $(2, 405)$.

2. ¿Es posible que dicha dirección lógica se traduzca en la dirección física 9322? Razónelo.

   Supongamos que sí. Como el desplazamiento de la memoria lógica es 405, significa que ese marco de página empieza en la dirección física $9322 - 405 = 8917$. En paginación, tenemos que todos los marcos tienen el mismo tamaño. Como 8917 no es múltiplo del tamaño del marco (1024), tenemos que no es posible un marco de página que empiece en ese valor.

Ejercicio 6.2.19. Suponga que tenemos 3 procesos ejecutándose concurrentemente en un determinado instante. El sistema operativo utiliza un sistema de memoria con paginación. Se dispone de una memoria física de 131072 bytes (128K). Sabemos que nuestros procesos al ser ejecutados tienen los parámetros que se muestran en la tabla.

Proceso	**Código**	**Pila**	**Datos**
A	20480	14288	10240
B	16384	8200	8192
C	18432	13288	9216

Tabla 6.7: Parámetros de los procesos del ejercicio 6.2.19.

Los datos indican el tamaño en bytes de cada uno de los segmentos que forman parte de la imagen del proceso. Sabiendo que una página no puede contener partes de dos segmentos diferentes (pila, código o datos), hemos de determinar el tamaño

de página que debería utilizar nuestro sistema y se barajan dos opciones: páginas de 4096 bytes (4K) o páginas de 512 bytes (1/2K). Se pide:

1. ¿Cuál sería la opción más apropiada, 4096 $B = 4\ KB$ o 512 $B = 0{,}5\ KB$?. Justifica totalmente la respuesta mostrando todos los cálculos que has necesitado para llegar a dicha conclusión.

   Calculamos el número de marcos de páginas que serían necesarios para cada opción:

   $$\frac{131072\ B}{4096\ B} = 32 \text{ marcos de página} \qquad \frac{131072\ B}{512\ B} = 256 \text{ marcos de página}$$

   Suponiendo la opción de 4096 B, el número de páginas de cada parte y cada proceso sería:

Proceso	**Código**	**Pila**	**Datos**
A	5	4	3
B	4	3	2
C	5	4	3

   En total serían necesarias 33 páginas, pero solo disponemos de 32 de ellas.

   Suponiendo la opción de 512, el número de páginas de cada parte y cada proceso sería:

Proceso	**Código**	**Pila**	**Datos**
A	40	28	20
B	32	17	16
C	36	26	18

   En total serían necesarias 233 páginas, por lo que sí nos caben los 3 procesos en memoria física.

2. ¿Cuál es el formato de cada entrada de la Tabla de Páginas con el tamaño de página elegido? Justifica el tamaño de los campos con direcciones. Puedes añadir los bits que consideres necesarios para el buen funcionamiento del sistema indicando para qué van a ser utilizados.

   Como hay 256 marcos de página, necesitamos 8 bits para codificar el marco de página.

   Respecto a los bits adicionales, sería conveniente incluir 1 bit de protección[2] y 2 bits para codificar el tipo de página (Código, Pila o Datos).

   Por tanto, cada entrada de la Tabla contendría el marco de página (8 bits) y los bits adicionales (3 bits).

[2]Para codificar si es de lectura o escritura.

3. ¿Cuántas Tablas de Páginas habrá en este sistema?¿Cuántas entradas hay en cada tabla de páginas (filas)?

   Habrá una tabla de página por proceso. Por tanto, habrá 3 Tablas de Páginas.

   Para el proceso A, serían necesarias $40 + 28 + 20 = 88$ páginas, por lo que se necesitarían 88 entradas.

   Para el proceso B, serían necesarias $32 + 17 + 16 = 65$ páginas, por lo que se necesitarían 65 entradas.

   Para el proceso C, serían necesarias $36 + 26 + 18 = 80$ páginas, por lo que se necesitarían 80 entradas.

Ejercicio 6.2.20. En la gestión de memoria en un sistema paginado, ¿qué estructura/s de datos necesitará mantener el Sistema Operativo para administrar el espacio libre?

La estructura de datos que emplea el SO en el caso de un sistema paginado es la Tabla de Páginas.

Ejercicio 6.2.21. Estamos trabajando con un sistema operativo que emplea una gestión de memoria paginada. Cada página tiene un tamaño de 2.048 bytes. La memoria física disponible para los procesos es de 8 MBytes. Suponga que primero llega un proceso que necesita 31.566 posiciones de memoria (o bytes) y, después, llega otro proceso que consume 18.432 posiciones cuando se carga en memoria. Se pide calcular la fragmentación interna provocada en cada proceso.

Respecto al proceso A, tenemos que el cociente de $\frac{31566}{2048}$ es 15, por lo que se emplearían 15 páginas completas. Además, como el resto es 846, se emplearían 846 B de la página 16, y de esta página se desperdiciarían $2048 - 846 = 1202$ B.

Respecto al proceso B, como la división de $\frac{18432}{2048} = 9$ es una división entera, no se produce fragmentación interna, ya que se ocupan 9 páginas completamente.

Ejercicio 6.2.22. Considere la siguiente tabla de segmentos:

Segmento	**Dirección base**	**Longitud**
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Tabla 6.8: Tabla de segmentos del ejercicio 6.2.22.

¿Qué direcciones físicas corresponden a las direcciones lógicas *(nº de segmento, desplazamiento)* siguientes? Si no puede traducir alguna dirección lógica a física, explique el por qué.

1. 0, 430

$$\begin{aligned}\text{Dir. Física} &= \text{Dir. Base} + \text{Desplazamiento} \\ &= 219 + 430 = 649\end{aligned}$$

2. 1, 10

$$\text{Dir. Física} = 2300 + 10 = 2310$$

3. 3, 400

$$\text{Dir. Física} = 1327 + 400 = 1727$$

4. 4, 112

$$\text{Dir. Física} = 1952 + 112 = \text{ERROR}$$

   En este caso no es posible, ya que el desplazamiento (112) es mayor o igual que la longitud del segmento correspondiente (96).

Ejercicio 6.2.23. ¿Qué cambio de contexto tardará menos y por qué?

(a) El producido entre dos hebras del mismo proceso.

(b) El producido entre dos hebras de distintos procesos.

Tarda menos en producirse el cambio de contexto entre hebras del mismo proceso, ya que estas comparten el PCB y las direcciones de memorias asocidadas, ya que estas no dependen de la hebra sino del proceso. Por tanto, al tener que cambiar menos información, el cambio de contexto es más rápido al ser del mismo proceso.

Ejercicio 6.2.24. Para cada uno de los siguientes casos, indicad razonada y brevemente hasta qué punto el uso de multiprogramación aumentará **mucho, poco o nada** el número de programas finalizados por unidad de tiempo (**productividad**), en comparación con el uso de un sistema monoprogramado.

1. Una situación en la cual hay que ejecutar un único programa que emplea el 90 % de su tiempo en esperas de entrada/salida, y el resto en cálculos en la CPU.

   No afecta porque se está ejecutando un único proceso.

2. Una situación en la cual hay que ejecutar un único programa que emplea el 10 % de su tiempo en esperas de E/S, y el resto en cálculos.

   En este caso, la multiprogramación no aumentaría nada la productividad, ya que al haber solo un programa la CPU no tiene nada que hacer mientras que se está esperando al dispositivo de E/S.

3. Una situación en la que hay que ejecutar 5 programas, cada uno de los cuales emplea el 90 % de su tiempo en esperas de entrada/salida, y el resto en cálculos en la CPU (cada proceso espera un dispositivo de E/S distinto)

   En este caso, la multiprogramación aumentaría mucho la productividad en comparación con un sistema monoprogramado. Dado que cada proceso pasa la mayor parte de su tiempo esperando la entrada/salida, mientras que la CPU está inactiva, la multiprogramación permitiría a la CPU realizar otras tareas mientras espera a que se completen las operaciones de entrada/salida de cada proceso. De esta manera, el sistema puede ejecutar otros procesos en paralelo, en lugar de esperar a que se completen los procesos que están inactivos debido a esperas de entrada/salida.

4. Ua situación en la cual hay que ejecutar 5 programas, cada uno de los cuales emplea el 10 % de su tiempo en esperas de E/S , y el resto en cálculos de la CPU (cada proceso espera un dispositivo de E/S distinto)

   Depende de la potencia de nuestro procesador se podrá mejorar mucho o poco el rendimiento, ya que si nuestra CPU admite mucho cálculo al mismo tiempo, entonces con multiprogramación podemos desarrollar todos los procesos a la vez y ganar mucha eficiencia (pues el 90 % del tiempo de cada proceso se emplea en estos cálculos). Sin embargo, si nuestro procesador no es muy potente y no podemos ejecutar al mismo tiempo los cálculos de los 5 programas, tan solo ganaríamos un poco en la eficiencia por los solapamientos de las esperas de Entrada / Salida (ya que, en principio, se espera a dispositivos distintos).

5. Una situación en la cual hay que ejecutar 5 programas, cada uno de los cuales emplea el 99 % de su tiempo en esperas de E/S, y el resto en cálculos de la CPU (todos los procesos usan un mismo dispositivo de E/S, no compartible por más de un proceso durante una operación de E/S).

   En este caso se aumentaría un poco la productividad mediante la multiprogramación, ya que se permitiría que, mientras que el dispositivo de E/S está ocupado, la CPU avance con otros programas hasta la instrucción que se comunica con el dispositivo de E/S. Sin embargo, al haber tan pocos cálculos y ser caso todo tiempo de espera, y este no se puede reducir al ser el dispositivo común para todos los procesos, tampoco se reduciría en gran medida.

Ejercicio 6.2.25. Considerad un sistema de computación en el que se ejecutan tres programas (procesos) que se encuentran cargados en memoria desde el primer instante y con las siguientes cargas de trabajo:

- P1: 5ds de CPU, tras lo cual utiliza 2ds la impresora y acaba utilizando la CPU 2ds.
- P2: 1ds de CPU, tras lo cual utiliza 5ds el escáner. Posteriormente, vuelve a utilizar la CPU durante 3ds y acaba utilizando la impresora durante 3ds.
- P3: 4ds de CPU.

1. Dibujad un diagrama de distribución de tiempo de la CPU y de los dispositivos de E/S (Impresora y Escáner) suponiendo que se utiliza un sistema operativo multiprogramado y que el orden en el que se ejecutan los procesos es: P1, P2 y P3.

CPU	/	/	/	/	/	+	-	-	-	-	/	/	+	+	+			
Impresora						/	/									/	/	/
Escáner							+	+	+	+	+							
Tiempo(ds)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

El proceso uno se representa con /, el proceso 2 con + y el proceso 3 con -

2. Indicad la cantidad de tiempo (total) que han estado los procesos P2 y P3 en estado preparado.

   El proceso P2 ha estado 6 ds esperando (1-5 y 12) y el proceso P3 6 ds también (1-6).

3. ¿Cuánto tiempo hubiera empleado el sistema en ejecutar estos procesos si el sistema operativo hubiese sido monoprogramado?

   Hubiera empleado 25 ds (9 + 12 + 4).

6.3. Compilación y Enlazado de Programas

Ejercicio 6.3.1. Un procesador (CPU) puede interpretar y ejecutar directamente las instrucciones de un programa en:

(a) Lenguaje de alto nivel de tipo intérprete.

(b) Lenguaje ensamblador o en lenguaje máquina, cualquiera de los dos.

(c) Sólo lenguaje máquina.
Ya que la CPU no entiende ninguna otra cosa; de hecho, la existencia del propio lenguage máquina no tendría sentido si este no fuese el caso.

(d) En pseudocódigo o en lenguaje ensamblador.

Ejercicio 6.3.2. ¿Es lo mismo un token que un lexema? Muestre algún ejemplo.

No, no es lo mismo. Un token relaciona a un conjunto de lexemas que tienen la misma función sintáctica.

Por ejemplo, el token IDENT que contiene a todos los identificadores posibles de variables puede estar formado por las combinaciones de letras y números. En este caso, los lexemas asociados a ese token podrían ser `i`, `var` o `lado1`, por ejemplo.

No obstante, para el caso de las palabras reservadas se tiene que hay un token por cada palabra reservada.

Ejercicio 6.3.3. ¿El compilador es la única utilidad necesaria para generar un programa ejecutable en una computadora?

No, también son necesarias otras. De hecho, con el compilador tan solo se pasa del código fuente a código ensamblador.

Otras utilidades necesarias para generar un programa ejecutable son el ensamblador para pasar a código máquina, el enlazador, o el preprocesador.

Ejercicio 6.3.4. El análisis léxico es una etapa de la compilación cuyo objetivo es:

(a) Extraer la estructura de cada sentencia, reconociendo los componentes léxicos (tokens) del lenguaje.

La segunda parte sí sería correcta, pero el análisis léxico no se encarga de extraer la estructura; ese es el objetivo de la fase de análisis.

(b) Descomponer el programa fuente en sus componentes léxicos (tokens).

(c) Extraer el significado de las distintas construcciones sintácticas y elementos terminales.

Esta fase es el análisis semántico.

(d) Sintetizar el programa objeto.

Esto pertenece a la fase de síntesis.

Ejercicio 6.3.5. El análisis sintáctico es una etapa de la compilación cuyo objetivo es:

(a) Extraer la estructura de cada sentencia, reconociendo los componentes léxicos (tokens) del lenguaje.
Es esta la única función del analizador sintáctico, el comprobar la correcta estructura de cada una de las sentencias del programa.

(b) Descomponer el programa fuente en sus componentes léxicos (tokens).

(c) Extraer el significado de las distintas construcciones sintácticas y elementos terminales.

(d) Sintetizar el programa objeto.

Ejercicio 6.3.6. Para el siguiente código que aparece a la izquierda en lenguaje C++ (fichero `test.cpp`, código fuente 1), indique el nombre de la fase en la que el compilador produce el mensaje de error que aparece a la derecha y explique la naturaleza del mismo:

```
1  int main (void)
2  {
3      int i;
4      char* j;
5
6      j = i;
7
8      if (i == 0)
9          i += ;
10
11     ¬;
12
13     return 0;
14 }
```

Código fuente 1: Fichero `test.cpp`

1. `test.cpp:9: error: expected primary-expression before ';' token`

   Tenemos que se trata de un error sintáctico; ya que aunque todos los lexemas de la línea 9 son válidos (tienen un token asociado) no se puede llegar a esa línea con las reglas de producción. Esto se debe a que falta el $r - value$, es decir, a la derecha del operador binario `+=` falta un valor (dada la naturaleza del mismo precisada en su nombre).

2. `test.cpp:6: error: invalid conversion from 'int' to 'char*'`

   Es un error semántico. Todas las secuencias de la línea tienen asociadas un token preciso y además la estructura de la sentencia (sintaxis) es correcta; sin embargo, intentar hacer una conversión de tipo entero a tipo puntero a

carácter no tiene sentido en el lenguaje de programación: esta sentencia es irrealizable y se produce el error mencionado.

3. `test.cpp:11: error: stray '\302' in program`

   Se trata de un error léxico, ya que el carácter ¬ no tiene ningún token asociado (no es un identificador, operador, expresión aritmética, palabra clave, etc.) pues no pertenece al alfabeto de la gramática de C++.

Ejercicio 6.3.7. Muestre un ejemplo a partir de una sentencia en lenguaje C++ en la que un error léxico origine un error sintáctico derivado y otro error léxico que no derive en error sintáctico.

Consideramos en primer lugar la línea `int mai?n(){` donde suponemos que en la siguiente línea continua el código. Tenemos que se trata de un error léxico, ya que no reconoce `mai?n` como un identificador válido. Además, dará error sintáctico por la falta de la función `main`.

Consideramos ahora la línea `int ñum;`. Tenemos que se trata de un error léxico ya que la letra `ñ` no pertenece al alfabeto de C++, por lo que no encontrará un token asociado. No obstante, no genera un error sintáctico ya que la sintaxis es correcta, al especificar el tipo, el nombre de la variable y terminar con el `;`.

Observación. Estos casos no obstante no son normales, ya que al detectar un error se detiene la compilación. No se ha corregido por tanto en clase.

Ejercicio 6.3.8. Muestre un ejemplo a partir una sentencia de en lenguaje C++ en la que un error léxico origine un error sintáctico y semántico derivados y otro error léxico que no los derive.

Un ejemplo sería declarar una variable de la forma int 1variable = 2; int var = 1variable; El identificador no es válido y provocaría un error léxico , que derivará en uno sintáctico y en uno semántico, pues se utiliza abajo una variable no declarada.

Observación. Estos casos no obstante no son normales, ya que al detectar un error se detiene la compilación. No se ha corregido por tanto en clase.

Ejercicio 6.3.9. ¿Sería siempre posible realizar la depuración de un archivo objeto? Razone la respuesta.

No, ya el fichero objeto no es ejecutable. Además, al no haber sido enlazado no contiene la cabecera en la que, por ejemplo, figura dónde empieza el programa, por lo que el depurador no sabría por donde empezar.

Como caso excepcional, se podría depurar una parte concreta del programa, pero no es útil y en la práctica no se considera este caso. En la mayoría de depuradores no es ni siquiera posible.

Ejercicio 6.3.10. Dado un programa escrito en lenguaje ensamblador de una arquitectura concreta, ¿sería directamente interpretable ese código por esa computadora? En caso contrario ¿qué habría que hacer?

No, para que fuese interpretable por parte de la computadora necesitaría traducirse el código a código máquina. De esta traducción se encargaría el ensamblador.

Ejercicio 6.3.11. ¿Sería necesario usar siempre el enlazador para obtener un programa ejecutable?

Sí, ya que si no se enlaza la cabecera del archivo no se actualizará, y por tanto no se tendrá la instrucción con la que se desea empezar.

Ejercicio 6.3.12. Dado un único archivo objeto, ¿podría ser siempre un programa ejecutable y correcto simplemente añadiendo la información de cabecera necesaria?

En general no, ya que se necesitaría el uso del enlazador porque, aunque haya un solo archivo objeto, puede que haya bibliotecas externas que tengan que ser cargadas.

Ejercicio 6.3.13. Dado un programa ejecutable que requiere de una biblioteca dinámica, ¿por qué no es necesario recompilar el código fuente de dicho programa si se modifica la biblioteca?

Las bibliotecas dinámicas se integran en el programa en tiempo de ejecución. Por tanto, como el programa compilado no contiene a la biblioteca, no es necesario recompilar para guardar los cambios. En el caso de hacer uso de bibliotecas estáticas, el código obsoleto sí que estaría incluido en el código máquina del programa ejecutable, y por tanto debería recompilarse para actualizarse.

En único caso en el que la actualización de la biblioteca dinámica haría necesario volver a compilar el programa es que se cambiasen las cabeceras de las funciones.

Ejercicio 6.3.14. Indique en qué fase del proceso de traducción y ejecución de un programa se realizará cada una de las siguientes tareas:

1. Enlazar una biblioteca estática: Fase de enlazado.
2. Eliminar los comentarios del código fuente: En la compilación, en la fase de análisis léxico.

   También es válido indicar que ocurre en el preprocesado, dependiendo de la fuente consultada.
3. Mensaje de error de que una variable no ha sido declarada: Fase de análisis semántico.

   En particular, seguramente ocurrirá porque no la encontrase en la tabla de símbolos (cualquier cuestión relacionada con el ámbito será asunto de esta fase).
4. Enlazar una biblioteca dinámica: Fase de ejecución.

Ejercicio 6.3.15. Indique en qué fase o fases del proceso de compilación de un lenguaje de programación de alto nivel se detectarían los siguientes errores:

1. Una variable no está definida: Fase de análisis semántico.

   De nuevo, cuestión de ámbito dentro del programa.
2. Aparece un carácter o símbolo no esperado: Fase de análisis léxico.
3. Aparecen dos identificadores consecutivos: Fase de análisis sintáctico.
4. Aparecen dos funciones denominadas bajo el mismo nombre: Fase de análisis semántico.

   No obstante, dependiendo del lenguaje puede ser que no diese error, ya que la función puede estar sobrecargada.

5. Aparece el final de un bloque de sentencias pero no el inicio del mismo: Fase de análisis sintáctico.

6. Aparece un paréntesis cerrado y no se ha podido emparejar con su correspondiente paréntesis abierto: Fase de análisis sintáctico.

7. Una llamada a una función que no ha sido definida: Fase de análisis semántico.

8. En la palabra reservada `main` aparece un carácter extraño no esperado, por ejemplo `mai¿n`: Fase de análisis léxico que derivará en error sintáctico.

Ejercicio 6.3.16. ¿Todo error sintáctico origina un error semántico? En caso contrario, demuéstrelo usando algún contraejemplo.

No, y como contrajemplo, sea el siguiente programa:

```
int num=7;
num += ;
```

Tenemos que hay un error sintáctico en la segunda línea al faltar el $r - value$, pero esto no genera ningún error semántico.

Observación. Estos casos no obstante no son normales, ya que al detectar un error se detiene la compilación. No se ha corregido por tanto en clase.

Ejercicio 6.3.17. Consideramos la siguiente gramática $G = \{V_T, V_N, S, P\}$ donde el axioma $S = programa$ y P contiene las siguientes reglas de producción:

- programa $\longrightarrow$ {lista_sentencias}
- lista_sentencias $\longrightarrow$ sentencia | sentencia ; lista_sentencias
- sentencia $\longrightarrow$ identificador = expresión
- identificador $\longrightarrow$ letra | identificador digito | identificador letra
- letra $\longrightarrow$ a|b|c|...|z
- digito $\longrightarrow$ 0|1|2|...|9
- expresion $\longrightarrow$ identificador operador identificador | identificador
- operador $\longrightarrow$ $+|-|*|/$

1. Obtener la lista de Tokens

   En primer lugar, obtengo la lista de Tokens:

TOKEN	PATRÓN
LLAVE_AB	{
LLAVE_CE	}
PUNTO_Y_COMA	;
ASSIGN	=
IDENTIF	letra(letra|digito)*
OPERADOR	$+\|-\|*\|/$

Tabla 6.9: Lista de Tokens de la gramática del ejercicio 6.3.17

2. Ver si las siguientes sentencias se pueden generar mediante esta gramática:

   a) $\{8a = b + c\}$

   El proceso de derivación es:

   programa $\longrightarrow$ {lista_sentencias} $\longrightarrow$ {sentencia} $\longrightarrow$
   $\longrightarrow$ {identificador = expresión}

   Por tanto, tenemos que no es posible generar dicha sentencia, ya que un identificador ha de empezar por letra.

   b) $\{\text{suma} + b = a * c\}$

   El proceso de derivación es:

   programa $\longrightarrow$ {lista_sentencias} $\longrightarrow$ {sentencia} $\longrightarrow$
   $\longrightarrow$ {identificador = expresión}

   Por tanto, tenemos que no es posible generar dicha sentencia, ya que un identificador no puede contener el operador +.

   c) $\{\text{suma} = b * z\}$

   El proceso de derivación es:

   programa $\longrightarrow$ {lista_sentencias} $\longrightarrow$ {sentencia} $\longrightarrow$
   $\longrightarrow$ {identificador = expresión} $\longrightarrow$
   $\longrightarrow$ {identificador = identificador operador identificador} $\longrightarrow$
   $\longrightarrow$ {suma= $b * z$}

6.4. Introducción a las Bases de Datos

Ejercicio 6.4.1. Se desea diseñar un esquema relacional de una base de datos para un centro de enseñanza que contenga información sobre los alumnos (dni, nombre, dirección . . .), las asignaturas (código de asignatura y nombre de esta se considera la información mínima) y las calificaciones que se obtienen en cada una de las mismas. Desarrollar un modelo ER del mismo y posteriormente reducirlo a tablas.

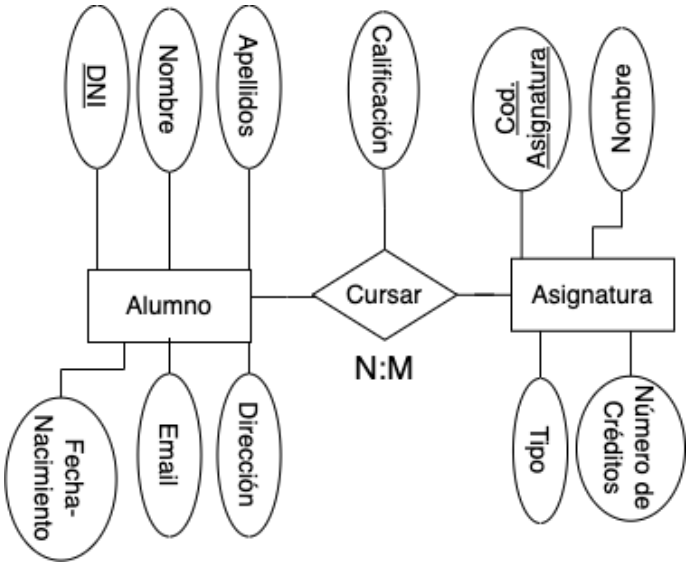


Figura 6.3: Modelo Entidad-Relación del ejercicio 6.4.1

Tendríamos tres tablas:

Alumno (DNI, Nombre, Apellidos, Dirección, Email, Fecha de Nacimiento)

Asignatura (Cód-Asignatura, Nombre, Tipo, Número de Créditos)

Cursar (DNI, Cód. Asignatura, Calificación)

Ejercicio 6.4.2. Se desea diseñar la base de datos de una biblioteca particular, de modo que para cada libro se deberá almacenar: su título, número de páginas, ISBN, materia, año de edición, editorial y autor o autores del mismo, para los que, además de su nombre, se recogerán los siguientes datos: dirección de correo electrónico, nacionalidad y fecha de nacimiento. Además, para cada editorial se deberá guardar su dirección, localidad y país. Teniendo en cuenta que se pueden añadir los campos que se consideren oportunos para poder relacionar convenientemente las distintas entidades del problema, realizar lo que se pide en cada uno de los siguientes apartados:

- Dibujar el esquema conceptual, utilizando el modelo entidad-relación.

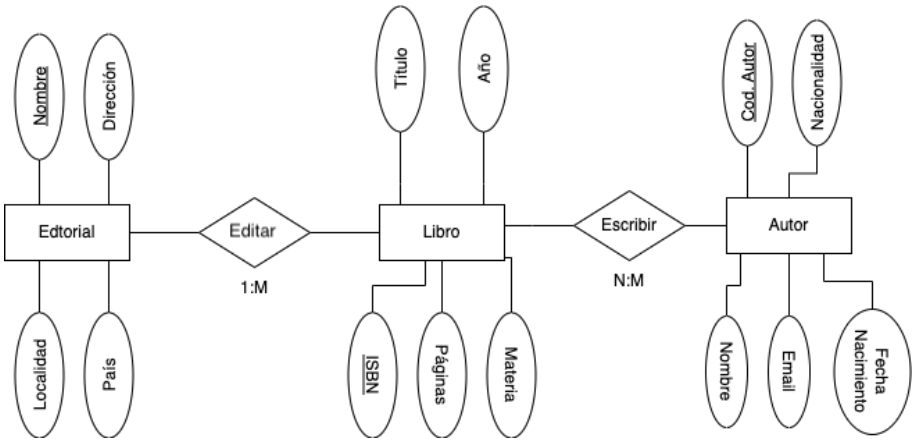


Figura 6.4: Modelo Entidad-Relación del ejercicio 6.4.2

- Obtener, a partir de lo realizado en el apartado anterior, las tablas que se tendrían que crear en un SGBD relacional, indicando qué campos compondrían cada tabla y cuál sería la clave primaria de cada una de ellas.

  Tendríamos cuatro tablas:

  Autor (Cod. Autor, Nombre, Email, FechaNacimiento, Nacionalidad)

  Libro (ISBN, Título, Páginas, Materia, Año, NombreEd.)

  Editorial (NombreEd., Direcciones, Localidad, País)

  Escribir (Cod. Autor, ISBN)

Ejercicio 6.4.3. Suponga que la base de datos para una Universidad del ejercicio (1) considera además de la información sobre los alumnos y las asignaturas, las carreras que se pueden estudiar. Construir un modelo ER y pasarlo posteriormente a un esquema relacional teniendo en cuenta las siguientes restricciones:

- Un alumno puede estar matriculado en muchas asignaturas.
- Una asignatura sólo puede pertenecer a una carrera.
- Una carrera puede tener muchas asignaturas.

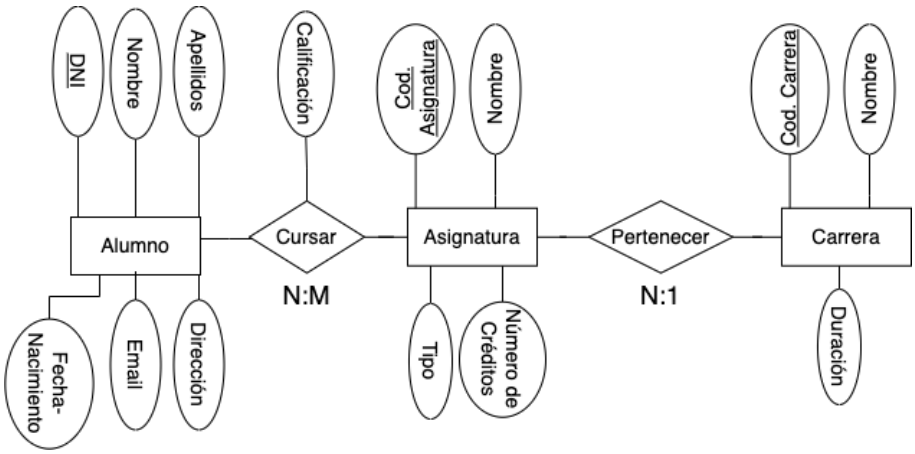


Figura 6.5: Modelo Entidad-Relación del ejercicio 6.4.3

Tendríamos tres tablas:

Alumno (DNI, Nombre, Apellidos, Dirección, Email, Fecha de Nacimiento)

Asignatura (Cód. Asignatura, Nombre, Tipo, Número de Créditos, Cód. Carrera)

Carrera (Cód. Carrera, Nombre, Duración)

Cursar (DNI, Cód. Asignatura, Calificación)

Observación. Se podría considerar que el nombre de titulación fuese el campo llave.

Ejercicio 6.4.4. Se desea diseñar una base de datos para un centro comercial organizado por secciones que contenga información sobre los clientes que han comprado algo, los trabajadores, el género que se oferta y las ventas realizadas. Construir un modelo ER y pasarlo posteriormente a un esquema relacional teniendo en cuenta las siguientes restricciones:

- Existen tres tipos de trabajadores: gerentes, jefes y vendedores.
- Cada sección está gestionado por un gerente.
- Un determinado producto sólo se encuentra en una sección.
- Los jefes y vendedores sólo pueden pertenecer a una única sección.
- Un gerente tiene a su cargo a un cierto número de jefes y éstos a su vez a un cierto número de vendedores.
- Una venta la realiza un vendedor a un cliente y debe quedar constancia del artículo vendido. Sólo una unidad de cada artículo por apunte de venta.

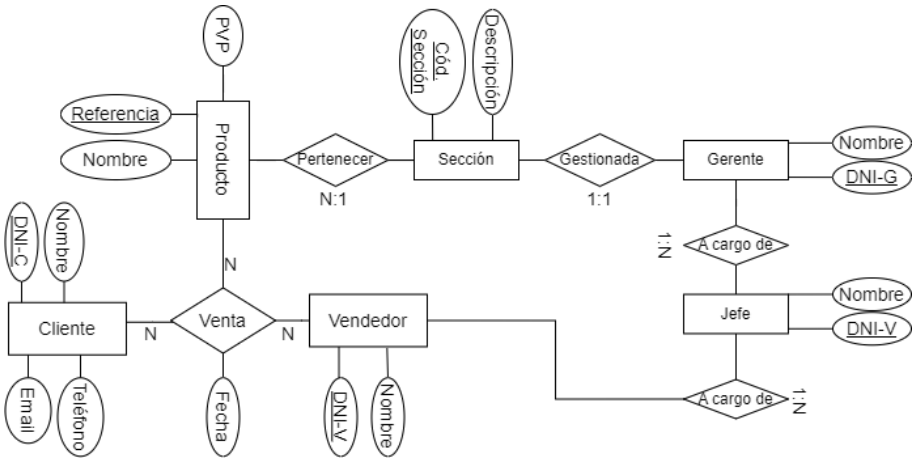


Figura 6.6: Modelo Entidad-Relación del ejercicio 6.4.4

Tendríamos las siguientes tablas:

Producto (Referencia, Nombre, PVP, Cód-Sección)

Sección (Cód. Sección, Descripción)

Gerente (DNI-G, Nombre, Cód-Sección)

Jefe (DNI-J, Nombre, DNI-G)

Vendedor (DNI-V, Nombre, DNI-J)

Comprador (DNI-C, Nombre, Teléfono, Email)

Venta (Fecha, DNI-C, DNI-V, Referencia)

Observación. En la tabla **Venta**, el campo Fecha sería clave o no si consideramos que la misma venta se puede repetir en fechas distintas.

Ejercicio 6.4.5. (EXAMEN) Se desea implementar un sistema de gestión de cupones de la ONCE en el que se registre cada venta de cupones. En el caso de que un cupón resulte premiado, el comprador ha de acudir a una Administración a cobrarlo. Tenga en cuenta las siguientes restricciones:

- Cada cupón es único.
- Cada vendedor tiene asociado su código de la seguridad social.
- Cada comprador puede comprar más de un cupón y a distintos vendedores.
- Existe una única administración por provincia.

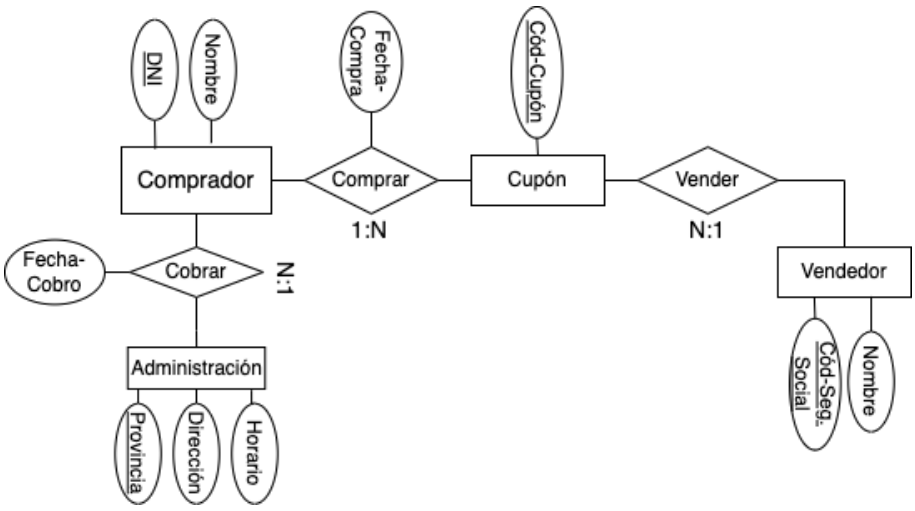


Figura 6.7: Modelo Entidad-Relación del ejercicio 6.4.5

Tendríamos las siguientes tablas:

Vendedor (Cód-Seguridad Social, Nombre)

Cupón (Cód-Cupón, Cód-Sedguridad Social)

Comprador (DNI, Nombre)

Comprar (DNI, Cód-Cupón, Fecha-Compra)

Administración (Provincia, Dirección, Horario)

Cobrar (Provincia, DNI, Fecha-Cobro)

Ejercicio 6.4.6. Los profesores de la asignatura de Bases de Datos de una Escuela Universitaria deciden crear una base de datos que contenga la información de los resultados de las pruebas realizadas a los alumnos. Para realizar el diseño, se sabe que:

- Los alumnos están definidos por su número de matrícula, nombre y grupo al que asisten a clase.
- Dichos alumnos realizan dos tipos de pruebas a lo largo del curso académico:
  1. Exámenes escritos: cada alumno realiza varios a lo largo del curso, y se definen por el número de examen, el número de preguntas que que consta y la fecha de realización (la misma para todos los que realizan el mismo examen). Evidentemente, es importante almacenar la nota de cada alumno por examen.
  2. Prácticas: se realiza un número indeterminado de ellas durante el curso académico, algunas serán en grupo y otras individuales. Se definen por un código de la práctica, título y el grado de dificultad. En este caso, los alumnos pueden examinarse de cualquier práctica cuando lo deseen, debiéndose almacenar la fecha y la nota obtenida.
- En cuanto a los profesores, únicamente interesa conocer (además de sus datos personales: DNI y nombre), quién es el que ha diseñado cada práctica, sabiendo que en el diseño de una práctica puede colaborar más de uno, y que un profesor puede diseñar más de una práctica. Interesa, además, la fecha en que ha sido diseñada cada práctica por el profesor correspondiente.

Para ello se pide:

1. Diseñar un modelo Entidad Relación para este caso.

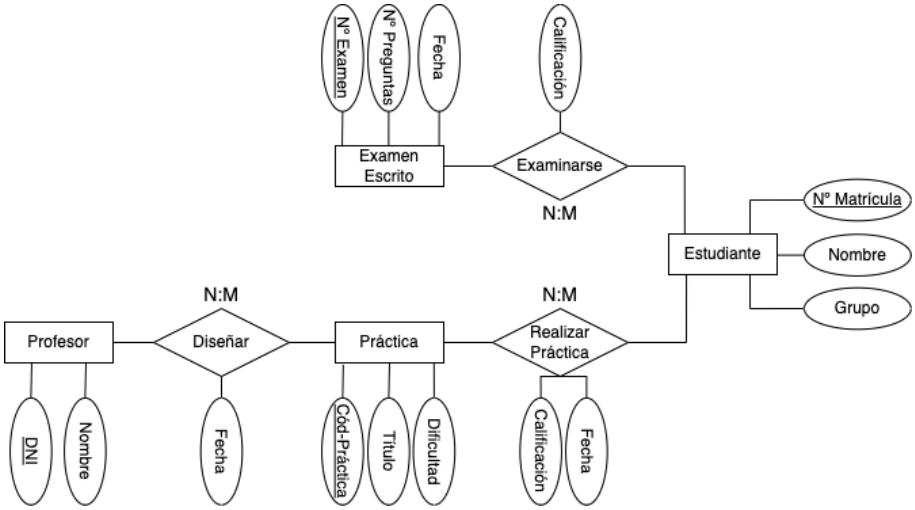


Figura 6.8: Modelo Entidad-Relación del ejercicio 6.4.6

2. Obtener, a partir de lo realizado en el apartado anterior, las tablas que se tendrían que crear en un SGBD relacional, indicando qué campos compondrían cada tabla y cuál sería la clave primaria para cada una de ellas.

Tendríamos las siguientes tablas:

Profesor (DNI, Nombre)

Diseñar (Cód-Práctica, DNI, Fecha)

Práctica (Cód-Práctica, Título, Dificultad)

Realizar Práctica (Cód-Práctica, Nº Matrícula, Fecha, Calificación)

Estudiante (Nº Matrícula, Nombre, Grupo)

Examinarse (Nº Matrícula, Nº Examen, Calificación)

Examen (Nº Examen, Nº de preguntas, Fecha)

Ejercicio 6.4.7. Se desea diseñar un esquema relacional de una base de datos para un colegio mayor que represente que los residentes (DNI, edad, nombre, apellidos) tienen un responsable asignado (DNI, nombre, apellidos, dirección, móvil, parentesco) desde una determinada fecha para que el director de la residencia pueda contactar con ellos en caso de incidencias. Un residente puede tener un único tutor asignado y un tutor solo puede ser responsable de un alumno.

Para ello se pide:

1. Diseñar un modelo Entidad Relación para este caso.

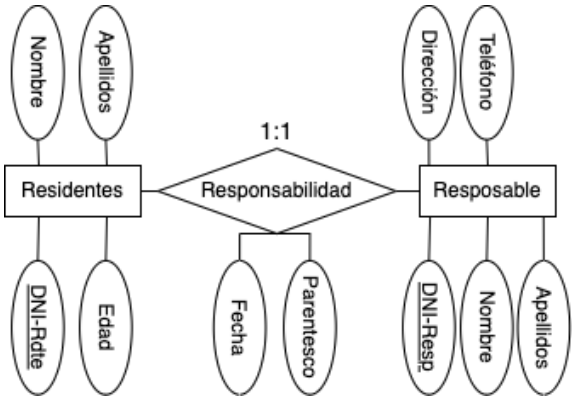


Figura 6.9: Modelo Entidad-Relación del ejercicio 6.4.7

2. Obtener, a partir de lo realizado en el apartado anterior, las tablas que se tendrían que crear en un SGBD relacional, indicando qué campos compondrían cada tabla y cuál sería la clave primaria para cada una de ellas.

   Tendríamos las siguientes tablas:

   Residente (DNI-Rdte, Nombre, Apellidos, Edad)

   Responsable (DNI-Resp, Nombre, Apellidos, Dirección, Teléfono)

   Responsabilidad (DNI-Rdte, DNI-Resp, Fecha, Parentesco)

   Observación. Hemos supuesto que un residente tiene el mismo responsable durante toda su estancia, que no puede cambiarlo.

   En el caso de que un residente pudiese cambiar de responsable, la cardinalidad sería 1:N, y el campo llave de la tabla responsabilidad sería DNI-Resp o ambos DNIs, a elección.