Tema 1. Introducción a los computadores

1. Introducción a los computadores

La **informática** es la ciencia que estudia el procesamiento automático de la información. Existen cuatro **niveles** conceptuales para la descripción del computador: hardware, sistema operativo, aplicaciones y servicios red.

El **Hardware** es la parte material, tangible (componentes eléctricos, mecánicos, etc.). El **Software** es el conjunto de programas, instrucciones y reglas informáticas, intangible.

Un **computador** es una máquina electrónica que procesa información a través de una serie de instrucciones. Necesita unos dispositivos de entrada (recibe información) y de salida (envía información) y dispone de otros para almacenar y procesar la información.

La **información** que se procesa está expresada en forma binaria, por tanto, la que entra ha de codificarse para ser procesada y descodificarse para ser mostrada.

El **sistema operativo** es un programa que gestiona los recursos del computador en base a los programas que se ejecutan sobre la máquina. Este es el primer programa que se usa cuando ponemos en marcha nuestro dispositivo y hace que el resto funcionen sobre él. El **SO** interactúa directamente con el hardware gestionando los programas y operaciones. Forma parte de los programas del sistema. *MacOS, Windows, Linux*.

Entre los **programas del sistema** también encontramos el compilador y ensamblador, los cuales se encargan de traducir las instrucciones escritas en un lenguaje de alto nivel a uno simple para que el hardware las pueda ejecutar. Las **aplicaciones** son programas orientados a los usuarios.

Gracias al desarrollo de **Internet** es impensable considerar al computador como un elemento aislado. Existen muchos **servicios** proporcionados a través de aplicaciones de red a los que el usuario del computador podría tener acceso.

2. Representación de la información

Los programas y datos tienen que ser representados acorde a las máquinas que los usan. Los computadores usan 2 valores (**binario**) para que sea más fácil de almacenar.

El **bit** es la unidad de información elemental *(0 ó 1)*. Un **byte** son 8 bits ($2^8 = 256$). Podemos representar los números en distintas **bases**.

Para hablar de las capacidades de una memoria RAM, disco duro, etc. Usamos los múltiplos de los bytes

3. Programas traductores: los lenguajes del computador

Los programas se escriben en un **lenguaje de programación** de bajo nivel *(ensamblador)* o alto *(C)*. El ordenador solo entiende instrucciones en **lenguaje máquina** (binario), por eso se necesitan unos *programas traductores*.

La **compilación** es el proceso general de traducción. Tiene 4 fases:

1. Preprocesado: edición inicial con inserciones de código
2. Compilación: traducción de lenguaje de alto a bajo nivel (ensamblador de la máquina)
3. Ensamblado: traducción de lenguaje de bajo nivel a código máquina
4. Enlace: resolución de llamadas a códigos de librerías estáticas

4. El hardware del ordenador

Los computadores se dividen en 3 bloques (**esquema de Von Neumann**):
1. Unidad central de proceso (CPU), formada por la unidad de control (UC) y camino de datos (CD)
2. Memoria principal, que incluye instrucciones y datos
3. Entrada/Salida

Los **buses** son caminos de datos eléctricos que se mueven en bits entre los componentes. El ancho en bytes se conoce como palabra del computador.

Los **dispositivos de E/S** son la conexión del sistema con el mundo exterior, se conectan al bus de E/S mediante adaptadores.

La **memoria principal** es el almacenamiento temporal de instrucciones y datos que éstas manejan. **RAM (DRAM)**.

El **procesador (CPU o UCP)** ejecuta instrucciones de la memoria principal usando unidades de memoria de tamaño de la palabra del computador (registros). Uno de ellos es el **PC**, que indica la instrucción que se carga para ejecutar, posiblemente usando ALU.

La **CPU** ejecuta unos *tipos de instrucciones*:

- Carga. Copia datos desde la memoria principal a un registro
- Almacenamiento. Copia datos desde un registro a la memoria principal
- Operación. Copia de datos desde dos registros a la ALU, ejecuta una operación y copia el resultado a un registro.
- Salto. Copia una dirección en el PC, indicando la dirección de la siguiente instrucción a ejecutar.

Una memoria grande es más lenta que una pequeña. *RAM (DRAM) – grande; disco duro SATA – muy grande y lenta; registros CPU (SRAM) – muy rápida y pequeña*.

Los **procesadores** se hacen más rápidos con más facilidad que la memoria.

La **memoria caché** es una memoria intermedia que guarda datos a los que es probable que se acceda y se implementan con SRAM.

5. El sistema operativo: gestor de recursos del hardware

El **sistema operativo** es un programa que gestiona los recursos del computador en beneficio de los programas que se ejecutan sobre la máquina. Es el primer programa que entra en funcionamiento, gestiona la ejecución del resto y protege e interactúa con el hardware.

El **SO** forma parte de los **programas del sistema** donde encontramos, entre otros, el compilador y el ensamblador.

Abstracciones del SO.

Procesos:

- Abstracción de procesador + memoria + I/O para un programa
- Cada aplicación en un proceso "cree" ser el único software en ejecución en la máquina
- Los procesos requieren acciones sobre el hardware a través de las *llamadas al sistema* que ofrece el SO
- Puede haber varios procesos ejecutándose de forma *concurrente*
- Los procesos pueden tener varias unidades de ejecución, *hilos*

Memoria virtual:

- Abstracción de las distintas memorias de las que dispone la máquina
- Los procesos tienen la misma vista de un *espacio de direcciones virtual*
- El SO gestiona el mapeo de la memoria virtual en la física y la traducción de direcciones

Ficheros:

- Secuencia de bytes designados por un nombre y gestionados por el SO
- A parte del sentido tradicional, en los sistema UNIX todos los dispositivos de E/S se modelan como ficheros (discos, teclados, pantallas, interfaces de red)
- Método uniforme de acceso al hardware

6. Otros conceptos básicos

Esquema de almacenamiento LITTLE ENDIAN

Guarda los bytes de un valor de forma que los más significativos están en direcciones más altas de memoria.

Esquema de almacenamiento BIG ENDIAN

Guarda los bytes de un valor de forma que los más significativos están en direcciones más bajas de memoria y los menos en las más altas.

La **concurrencia** es la ejecución de múltiples tareas de forma simultánea.

El **paralelismo** es el uso de la concurrencia para hacer que un sistema funcione más rápido. Ejecuta varias tareas a la vez. El paralelismo a diferentes niveles:

1. **Concurrencia a nivel de procesos o hilos**. Varios procesos se ejecutan de forma simultánea. En sistemas *uniprocesador* es simulada, alternando procesos o hilos. En *multiprocesador* puede ser real.
2. **Paralelismo a nivel de instrucción**. Ejecución de múltiples instrucciones a la misma vez en el hardware. Si los procesadores son capaces de completar más de una instrucción por ciclo son *procesadores superescalares*.

3. **Paralelismo de única instrucción, múltiples datos (SIMD)**. Ejecución de operaciones sobre múltiples datos a partir de una única instrucción.

Entre las **abstracciones** más representativas tenemos:

- El **repertorio de instrucciones** ofrece una abstracción del hardware del procesador
- Las **máquinas virtuales** permiten la ejecución de diferentes máquinas con distintos SO sobre el mismo hardware

En el SO:

- Los **procesos** son una abstracción del código y los datos de un programa
- La **memoria virtual** son una abstracción de la memoria que se pueden usar los programas
- Los **ficheros** son una abstracción de los dispositivos de E/S

El **tiempo de ejecución / respuesta** es el tiempo total requerido por el computador para completar una tarea, incluyendo acceso a disco, memoria, etc.

El **tiempo de CPU** es únicamente el tiempo que un procesador (CPU) pasa realizando un cálculo para una específica tarea. Su fórmula es:

$$Tcpu = (InstruccionesAplic * Ciclos(Media) / Instrucción) / FrecuenciaProcesador$$

El **ancho de banda** es la cantidad de datos que se pueden transmitir entre *dos puntos* en un segundo. Se mide en bits/s (bps) o en Bytes/s (B/s). Depende del ancho del bus y de la frecuencia del bus.

$$AnchoBanda = FrecBus * AnchoBus$$
$$AnchoBanda = FrecCanal * AnchoCanal$$

7. Otros conceptos básicos

La primera máquina de programa almacenado, instrucciones de salto condicional y válvulas de vacío fue creada por **Von Neumann**.
En la segunda generación aparecen los **transistores** propiciada por los laboratorios Bell. Sen incluye el sw del sistema en el ordenador y se usan lenguajes de programación de alto nivel.
En la tercera generación se implementan celdas de memoria y puertas lógicas, es decir, el **circuito integrado**. También, aparece el concepto de *microprogramación* y se anuncia el primer *supercomputador*.
En la cuarta y quinta generación se fabrica la **memoria semiconductora**.

Recientemente, el mercado se segmenta en sobremesas, servidores y portátiles. Se han hecho mejoras y avances en Intel.

A partir de la 2º de los ’90 hay 3 grandes grupos de ordenadores gracias a la aparición de Internet:

- **Ordenadores personales**: uso individual. Buen rendimiento a bajo coste.
- **Servidores**: muchos usuarios. Aseguran la fiabilidad y escalabilidad. Hay de varios tipos.
- **Ordenadores empotrados**: pensados para ejecutar solo una aplicación. Tienen fuertes limitaciones en coste, consumo de energía y fiabilidad.

Actualmente, se ha desarrollado el **ARM** y cualquier procesador comercial pasó a ser un procesador **multinúcleo**. Esto hace que ahorremos energía, tiempo y los procesos funcionen mejor. El problema es que estos son más difíciles de programar ya que usan la **programación paralela**.

Tema 2. Representación de la información

Un ordenador es una máquina que procesa información: *instrucciones* se ejecutan como *datos*. La máquina solo entiende de 0s y 1s, por tanto, la *codificación* es el proceso de asignar a cada uno de los caracteres una secuencia de 0s y 1s (**Código E/S**).

1. Representación de enteros

Para convertir de decimal a binario separamos la parte entera de la fraccionaria. En la **parte entera** vamos dividiendo entre dos y en la **parte fraccionaria** vamos multiplicando por dos.

Operaciones aritméticas

SUMA	RESTA	PRODUCTO	DIVISIÓN
0 + 0 = 0	0 - 0 = 0	0 · 0 = 0	0 / 0 = Indetermin.
0 + 1 = 1	0 - 1 = 1 y debo 1	0 · 1 = 0	0 / 1 = 0
1 + 0 = 1	1 - 0 = 1	1 · 0 = 0	1 / 0 = ⬚
1 + 1 = 0 y llevo 1	1 - 1 = 0	1 · 1 = 1	1 / 1 = 1

Operaciones lógicas

SUMA LÓGICA (OR)	PRODUCTO LÓGICO (AND)	COMPLEMENTO (NOT)	OR EXCLUSIVO (XOR)
0 + 0 = 0	0 · 0 = 0	0' = 1	0 ⊕ 0 = 0
0 + 1 = 1	0 · 1 = 0	1' = 0	0 ⊕ 1 = 1
1 + 0 = 1	1 · 0 = 0		1 ⊕ 0 = 1
1 + 1 = 1	1 · 1 = 1		1 ⊕ 1 = 0

- Representación gráfica de las operaciones:

Los **valores sin signo** son el valor absoluto del número entero en binario natural.

- Rango = 0 … $2^n - 1$

Representación CON signo

1. Signo y magnitud

- N negativo, S = 1 // N positivo, S = 0
- Rango = $-(2^n - 1)$ … $2^n - 1$ *doble representación del 0*

2. Sesgada

- S es una cte denominada sesgo
- El resultado de la suma del entero a representar y el sesgo
- Sesgo = 2^{n-1}
- Rango = $-(2^{n-1})$ … $2^{n-1} - 1$ *no hay doble representación del cero*

3. Complemento a 2

- Cambiamos sus 0s por 1s y viceversa y le sumamos 1
- C2(C2(N)) = N // -(-N) = N
- Se usa para poder realizar las restas como sumas
- N negativo, S = 1 // N positivo, S = 0
- Rango = $-(2^{n-1})$ … $2^{n-1} - 1$ *no hay doble representación del cero*
- A veces ocurre desbordamiento

2. Representación de reales

Se usa generalmente la **notación exponencial**, donde M es la mantisa, B la base y E el exponente.

$$N = M * B^E$$

Estructura IEEE 754

$$Valor = (-1)^s * M * B^E$$

1. Campo de signo

S = 0, positivo // S = 1, negativo

2. Campo del exponente

Entero sesgado donde $S = 2^{ne-1} - 1$ y $e = E + S$

3. Campo de la mantisa

M = 1,m

Formatos más habituales de IEEE 754:

- **Simple precisión**: n = 32 bits (exponente = 8, mantisa = 23), S = 127
- **Doble precisión**: n = 64 bits (exponente = 11, mantisa = 52), S = 1023

Situaciones especiales:

a) Si e = 0 y m != 0, número desnormalizado. Desbordamiento gradual
b) e = 0 y m = 0, N = 0
c) e = 111…1:
c.1 – m = 0, infinito ó -infinito
c.2 – m != 0, NaN

Rango y precisión

Los valores representables se aglutinan en magnitudes menores. Debemos tener unas consideraciones respecto al tamaño:

- Precisión. Cifras de la mantisa que se pueden representar
- Rango. Intervalo de representación
- Si hay mayor número en la mantisa, hay más cifras significativas → mayor precisión
- Si hay mayor número en el exponente, hay mayor exponente posible → mayor rango

Redondeos

Algunos números reales no pueden ser representados de forma exacta con nm+1 cifras binarias significativas. Tenemos 4 modelos propuestos de redondeo por IEEE 754:

1. Redondeo al par
- Se redondea al número representable más cercano. Si el error es el mismo se elige el número par.
- Analizar el bit menos significativo que quedaría en la mantisa y añadir 2 bits adicionales (el de guarda y el de redondeo)

2. Redondear siempre hacia arriba
- Se redondea al número representable inmediatamente superior

3. Redondear siempre hacia abajo
- Se redondea al número representable inmediatamente inferior

4. Truncar siempre
- Se desechan siempre los dígitos no representables

Errores

Los redondeos incluyen un error implícito, entonces, es fácil que un número decimal más o menos redondo no pueda ser representado con absoluta exactitud.
Ciertas propiedades de los números reales, pej la asociatividad, no pueden cumplirse.
Los errores se van acumulando en los cálculos

Todo está relacionado con que al fin y al cabo IEEE 754 es un intento de representar un conjunto continuo de números (reales) con una representación fija de un conjunto de bits.

Otras representaciones

Brain Floating Point (bfloat):

- orientada al cálculo con matrices usadas en técnicas de aprendizaje en máquina (tensores)
- 1 bit signo, 8 bits exponente, 7 bits mantisa
- Bfloat16 tiene el mismo rango en 16 bits que IEEE 754 32 bits

- Codificación de signo, exponente y mantisa equivalente a IEEE 754
- Mejora rendimiento y disminuye el uso de memoria
- Cálculos vectoriales en bfloat16 y acumulación de resultados en IEEE 754 32 bits

Posits:
- Busca una solución para longitud fija de parte de exponente y mantisa
- 1 bit signo, conjunto de bits de régimen iguales, conjunto de es bits de exponente y conjunto de bits de mantisa sobrantes
- Número de bits iguales r determina un valor de exponenciación adicional
- Si s = 1, el resto de bits están en C2
- Representación única de valores
- No existe el desbordamiento gradual ya que no hay números desnormalizados
- No existen los NaN
- Cómputo más sencillo y hardware más simple
- Mejor representación de números cercanos al 0

3. Representación de caracteres

Los **códigos de E/S** asocian a cada carácter una determinada combinación de bits. Necesitamos un número de bits n tal que $n \geq \log_2 m$ *(para codificar {0,1,…,9} se necesitan 4 bits)*
El código ASCII es el más popular y se extiende desde los años 60.

El código ASCII

Para rellenar hasta 8 bits, se usa un bit adicional para el control de errores, caracteres gráficos o extensiones de idiomas. También, se usan bits adicionales para comprobar que no se almacena información de manera errónea.

Unicode

Estándar para representar todo texto posible en todos los sistemas de escritura del mundo. Cuenta con 110.000 caracteres.
A cada símbolo / carácter se le define por un nombre e identificador numérico, *code point (U+num.hex)*

4. Representación de contenido multimedia

Las imágenes están compuestas por infinitos puntos y cada uno de ellos tiene un nivel de gris o un color. Para codificar una imagen tenemos que ver cuántos puntos vamos a considerar y asociarles este nivel o color. Consideraremos que una imagen está formada por una matriz de píxeles y se representa con un mapa de bits donde los atributos de los píxeles se almacenan sucesivamente

Resolución de la imagen: $nº\ elementosLínea * nºelementosColumna$

Con los **mapas de vectores** representamos como colecciones de objetos ya que estos se modelan mediante vectores y ecuaciones. Al visualizar una imagen en pantalla se evalúan las ecuaciones y se escalan los vectores para poder generar la imagen. Algunas características de éstas son que son fáciles de procesar o escalar a cualquier tamaño.

Ejemplo de formato de representación de imagen

1. Mapa de bits: BMP, PICT, PPM, JPEG
2. Mapas de vectores: DXF, EPS, ODG

Representación del color

Las imágenes se pueden representar en escala de grises o en color.
En escala de grises, cada píxel tiene asociado un valor medo de gris.
Representación del color **RGB**. 8 bits. Se usa para la visualización en pantalla.
Representación del color **CYMK**. Se usa para la impresión con tinta.
La profundidad de color (bpp) es la cantidad de bits necesaria para almacenar el color de un bit que junto a la resolución, permite obtener el tamaño de la imagen codificada:

$$TamañoImagen(SinCompresion) = bpp * resolución$$

Sonido digitalizado

El sonido también debe ser discretizado mediante un proceso de **codificación**. Hay distintas etapas: conversión A/D, cuantificación y mapeo a bits.
La recuperación del sonido analógico se realiza mediante la **descodificación**.

En la **codificación** se busca reducir la cantidad de información digital necesaria y minimizar la pérdida de calidad perceptible al descodificar.

Parámetros en la codificación**:**

- Número de canales: estéreo (2), mono (1)
- Frecuencia de muestreo: muestras x segundo (Hz)
- Tasa de bits: bits por segundo (bps) y por canal necesarios en la representación
- Fidelidad: semejanza al audio original
- Complejidad en el codec
- Retardo asociado al codec

$$TamañoSonido(SinCompresion) = nºcanales * frecMuestreo(Hz) * tasaBits(bps) * duración(seg)$$

El sonido codificado en ristras de bits se puede almacenar en distintos **formatos de ficheros de audio**. Algunos definen una codificación concreta y otros son genéricos. *(WAV, MP3,WMA)*

Video digital

Aglutina sonido e imágenes digitalizados donde la imagen proviene de una sucesión de imágenes, fotogramas *(frames)*. Ha ido evolucionando con el paso del tiempo mejorando la resolución.

Resolución temporal**:** número de *frames* por segundo

Ilusión del movimiento: ~24 fps

Técnicas de exploración de la imagen para disminución del parpadeo:

- Entrelazada. Combinación de fotogramas con líneas pares e impares de la imagen
- Progresiva. Barrido continuado línea a línea

$$TamañoVideo(SinCompresion) = tamañoImagen + tamañoAudio$$
$$TamañoImagen(sinCompresion) = profundidadColor * resolucion * fps * duracion\ (seg)$$
$$TamañoSonido(SinCompresion) = nºcalanes * frecMuestreo(Hz) * tasaBits(bps) * duracion(seg)$$

Compresión del vídeo:

- RGB ofrece intensidad de los colores primario, separación de brillo y tonalidad
- Submuestreo de color mediante reducción de los bits usados para la crominancia
- Compresión espacial equivalente a las técnicas de compresión usadas en imágenes
- Compresión temporal similitud en *frames* sucesivos

Existen diversos **tipos de formatos de vídeo**: en crudo, con o sin compresión…

Formatos conocidos: AVI, MOV, MP4, MXF

Dentro de cada fichero de vídeo se encuentra la imagen y el audio almacenados según una codificación concreta (**codec**).

Tema 3. Estructura del computador

1. El procesador

ISA: La interfaz entre hardware y software

Las aplicaciones y el SO son software que se ejecuta sobre el hardware.
La **arquitectura** del repertorio de instrucciones o **ISA** es un contrato entre hw y sw. Aquí se permite que evolucionen ambos de manera independiente, se fomente la compatibilidad del sw y se produzca una **abstracción** esencial del procesador donde se comporta como si las insts se ejecutasen secuencial mente cuando en realidad lo hacen en paralelo para obtener mayor rendimiento.

ISA vs Micro-Arquitectura

La **micro-arquitectura** es la implementación específica de un ISA en una determinada CPU.
Tiene aspectos de implementación **invisibles** al sw , técnicas para conseguir mayor rendimiento.
Diferentes implementaciones de un mismo ISA pueden ejecutar el mismo sw.

¿Qué es un ISA?

Es una interfaz bien-definida entre hw y sw donde podemos tener una **definición funcional** de las operaciones y lugares de almacenamiento.
Una **descripción precisa** sería cómo invocar operaciones y acceder a los operandos.
Pero en un ISA no tenemos aspectos no funcionales como cómo implementar operaciones o cuáles consumen más energía.
Cada CPU implementa un determinado ISA. Algunos ejemplos de ISAs reales son *Intel x86, ARM*

Repertorio de instrucciones o ISA

Una instrucción definida en un ISA es una secuencia de bits (**lenguaje máquina**) que el procesador interpreta como una **orden** para realizar una **operación concreta**.
El **lenguaje ensamblador** es la representación **textual** de una instrucción en lenguaje máquina. Existe una correspondencia directa de cada instrucción en ensamblador y su código máquina.

Analogía: lenguas → ISAs

Comunicación: de persona a persona – de sw a hw
Estructura similar: narración – programa ; frase – instrucción; verbo – operación
Muchas lenguas distintas – Muchos ISAs diferentes
Diferencias entre lenguas e ISAs: las lenguas evolucionan mientras que los ISAs son explícitamente diseñados y ampliados sin ser ambiguos, al contrario que las lenguas.

Compatibilidad

Nadie compraría un ordenador nuevo si requieriese sw nuevo. Intel fue la primera compañía en darse cuenta de esto.
La **compatibilidad hacia atrás** se basa en que los nuevos procesadores deben soportar los programas existentes.
La **compatibilidad hacia adelante** se basa en que los nuevos programas deben funcionar sobre procesadores antiguos.

Características principales de un ISA

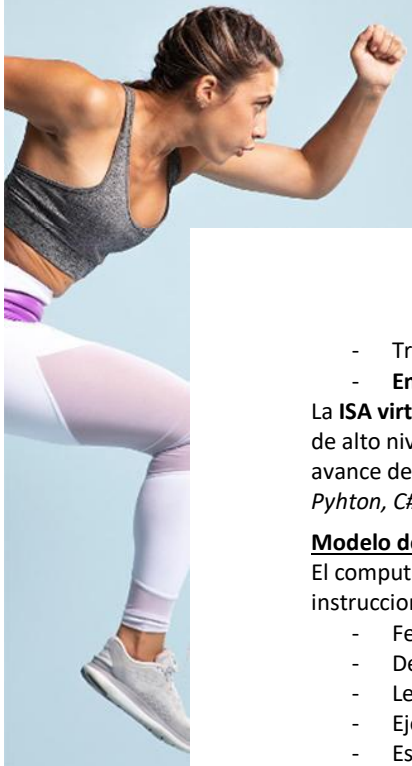
Codificación de las instrucciones: tamaño fijo (32-bits en MIPS & ARM) y tamaño variable (de 1 a 16 bytes en x86)
Número y tipo de registros: MIPS (32 registros enteros y 32 punto flotante) y ARM & x86 (16 enteros y 16 punto flotante)
Espacio de direccionamiento: x86_64 y ARM64 (64 bits)
Modos de direccionamiento de memoria: MIPS (dirección calculada mediante "registro + desplazamiento") y x86 (modos de direccionamiento mucho más complicados)

Tipos de Isa. Granularidad de las instrucciones

1. **CISC** (**Complex**) ISAs:
Instrucciones grandes y pesadas, se disminuye el número de instrucciones por programa y hay más "ciclos/instrucción" y/o más "segundos/ciclo"
2. **RISC** (**Reduced**) ISAs:
Aproximación minimalista a un ISA con instrucciones sencillas, menor "ciclos/instrucción" y "segundos/ciclo" y aumenta "instrucciones/programa" pero basándose en las optimizaciones del compilador.

Traducción e ISAs virtuales

Nueva interfaz para mejorar la compatibilidad: **ISA + sw de traducción**.



- Traducción **Binaria**: el código se transforma para ejecutar de forma nativa en la máquina
- **Emulación**: el código no se modifica sino que se interpreta instrucción a instrucción

La **ISA virtual** está diseñado para la traducción, no para la ejecución directa. El código va dirigido a un compilador de alto nivel o a un traductor de bajo nivel con la finalidad de tener portabilidad y mayor independencia con el avance del hw.
Pyhton, C# CLR

Modelo de ejecución de instrucciones

El computador es una máquina de estados finita con **registros**, **memoria** y **contador de programa**. Ejecuta instrucciones:

- Fetch: obtiene de la memoria la siguiente instrucción
- Decode: decodifica (averigua lo que hace) la instrucción
- Lee sus operandos de entrada: registros/memoria
- Ejecuta
- Escribe el resultado/salida: registros/memoria
- Pasa a la siguiente instrucción: actualiza en PC

Un programa son *datos en memoria*.

Longitud y formato de las instrucciones

La longitud de las instrucciones puede ser fija (+ común = 32 bits, implementación simple y con menos densidad de código) o variable (+ densidad de código y con Fetch complejo).
La codificación de las instrucciones puede ser en: RISC (pocos formatos, sencillo) o RISC (muchos formatos, complejo).

Operaciones y tipos de datos

En el sw los **tipos de datos** son un atributo del dato mientras que en el hw son un atributo de la operación. Los procesadores modernos suelen soportar operaciones aritmético-lógicas con enteros o aritméticas de punto flotante IEEE 754.
La mayoría de los procesadores tienen soporte SIMD que explota paralelismo de datos, operaciones enteras empaquetadas o de FP empaquetadas.

Ubicación de los datos. Operandos

Registros: memorias a *corto plazo*, más rápidos que la memoria y directamente nombrados por las instrucciones.
Memoria: memoria a *largo plazo*, más lenta que los registros, acceso por diferentes **modos de direccionamiento**.
Inmediatos: valores constantes codificados por la propia instrucción que solo actúan como operandos de entrada.

Granularidad de acceso

Direccionamiento de memoria a nivel de byte donde un **byte** es la granularidad mínima del ISA para leer o escribir en memoria.
Los ISAs también soportan lecturas/escrituras de mayor tamaño.

Transferencias del flujo de control

Se basa en una ejecución secuencial, pero existen algunas instrucciones que rompen esto como los saltos condicionales (*branches*) o los incondicionales (*jumps*). Se realizan llamadas y retornos a/de procedimientos.
Se calcula el destino del salto y se comprueba la condición de éste con códigos implícitos o registros e instrucciones específicas.

Tiempo de ejecución de un programa

El tiempo que tarda un programa en ejecutarse depende de 3 factores:

1. Instrucciones que se ejecutan para completar el programa:
   - **Número de instrucciones** dinámicas
   - Determinado por el ISA, programa y compilador
2. Rapidez con la que conmuta el procesador:
   - **Frecuencia de reloj**
   - Viene dado por el tiempo que le lleva a la electrónica de la CPU realizar una operación en el peor caso
   - Determinado por la micro-arquitectura y parámetros tecnológicos
3. Ciclos que tarda cada instrucción en ejecutarse
   - **Ciclos por instrucción** (CPI)
   - Determinado por el progama, compilador, ISA y micro-arquitectura

$$ExecutionTime = (instructionsProgram) * (secondsCycle) * (cyclesInstruction)$$

Implementación de un ISA

Fetch: obtiene instrucciones, traduce *opcode* en señales de control.

Unidad de control: determina qué cómputo lleva a cabo en base a las señales de control (encamina los datos a través del camino de datos).

Camino de datos: (*datapath*) realiza el cómputo.

CICLO: FETCH → DECODE → EXECUTE

Componentes de un procesador

Existen dos tipos de circuitos:

A - **Combinacionales**: computación sin estado. Funciones booleanas arbitrarias.

B – **Secuenciales**: almacenamiento de bits (**memoria**). Internamente contiene componentes combinacionales.

Concurrencia y paralelismo

Cada vez tenemos una demanda constante de mayor rendimiento del computador, es decir, hacer más cosas en menos tiempo. La **concurrencia** se basa en realizar múltiples actividades simultáneas pero no implica necesariamente paralelismo real.

El **paralelismo** es un uso de concurrencia para conseguir que el sistema vaya más rápido. Se puede explotar a diferentes niveles.

A - **A nivel de instrucción (ILP)**

Explotar que instrucciones cercanas en un mismo flujo de ejecución son independientes y se pueden ejecutar en paralelo. Las micro-arquitecturas actuales tienen ratio de 2-4 instrucciones/ciclo.

Con los procesadores segmentados las acciones requeridas por una instrucción se dividen en etapas con una limitación de ratio de ejecución máximo de 1 instrucción/ciclo. (CPI > 1)

Con los procesadores superescalares podemos ejecutar más de 1 instrucción/ciclo. (CPI < 1). Requieren muchos más recursos hw para explotar el ILP.

La CPU se divide en *front-end* y *back-end* con el objetivo de explotar el ILP para aumentar el rendimiento. En el ***front-end*** tenemos el control de instrucciones, decodificador, banco de registros y unidad de retirada de instrucciones. En el ***back-end*** tenemos la ejecución de instrucciones, despacha las operaciones primitivas requeredias a las unidades funcionales correspondientes siendo estas quienes realizan la operación.

B - **A nivel de tarea/hilo (TLP)**

Hasta hace 15 años la mayoría de sistemas eran **uniprocesador** donde un único procesadore conmutaba entre múltiples tareas.

A partir de los 2000s aparecen los **procesadores multinúcleo** con varias CPUs integradas en el mismo chip. Tienen un soporte para **SMT** donde cada una de las CPUs puede ejecutar varios flujos de ejecución simultáneamente decidiendo ciclo a ciclo qué hilo ejecutar.

C - **A nivel de datos SIMD**

El **SIMD** usa una misma instrucción que opera sobre varios datos a la vez en un mismo registro explotando el paralelismo a nivel de datos. Es la evolución más reciente en los ISAs.

2. La memoria

Estructura tradicional del bus CPU-memoria

El **bus** es una colección de cables en paralelo que interconectan los componentes del computador. Se divide en bus de direcciones, de datos y de control. Normalmente están compartidos por múltiples dispositivos.

Acceso a memoria: Transacción de lectura

Con la instrucción de carga (**MIPS**) la CPU lee el valor guardado en el registro, le suma el desplazamiento, coloca el resultado en el bus de direcciones *(dirección de memoria a leer)* y activa la señal de lectura en el bus de control. La memoria principal lee la dirección del bus de memoria y accede a los 32 bits. La memoria principal pone los 32 bits en el bus de datos.

La CPU lee los 32 bits del bus de datos y los copia al registro.

La memoria

Los programadores quieren memorias grandes y rápidas pero no existen a precios razonables con estas características. Usando la **jerarquía de la memoria** podremos tener algo parecido. Abajo se encuentran los dispositivos de almacenamiento más grandes, lentos y baratos; arriba los más pequeños, caros y rápidos.

Registros [0] – Caché L1 SRAM [1] – Caché L2 SRAM [2] – Caché L3 SRAM [2] – Memoria Principal SDRAM [3] – Almacenamiento secundario local (Disco duro) [4] – Almacenamiento secundario remoto (Servidores web, etc.) [4]

RAM *(Random Access Memory)*

Es una memoria de acceso aleatorio, direccionable a nivel de byte. Es **volátil** (pierde su estado si se apaga) y cada celda de memoria guarda un bit.

Hay dos tipos:

A - **SRAM** (Static): acceso muy rápido, mantiene su estado indefinidamente, 4-6 transistores por bit, mismo chip que la CPU.

B - **DRAM** (Dynamic): acceso más lento, se va refrescando su estado periódicamente, 1 transistor + 1 condensador por bit, integración limitada por necesidad de capacidad eléctrica.

Localidad de referencia

Principio de localidad: los programas tienden a usar datos e instrucciones cuyas direcciones están cercas de aquellas que han usado recientemente.

Localidad temporal: si se usa un dato, seguramente será usado próximamente. Mantiene los datos accedidos recientemente más cerca del procesador.

Localidad espacial: si se usa un dato, seguramente otros datos cercanos a él serán usados próximamente. Mueve los bloques de varias unidades a la vez.

Principio de localidad

Salvamos la diferencia de velocidad entre CPU y memoria usando el concepto de **localidad**. Buscamos una jerarquía de memoria para funcionar a un nivel más rápido y aprovechamos, en cada momento concreto, que los programas acceden a una parte relativamente pequeña de su espacio de direcciones.

Con la **localidad** detectamos zonas de memoria *(datos y código)* con más probabilidad de ser accedidas y llevarlas a la memoria más rápida.

La memoria caché

La **caché** es un almacenamiento pequeño y rápido que guarda un subconjunto de datos que hay en otro dispositivo mayor y más lento, es decir, que se encuentra en otro nivel mayor.

Las jerarquías de memoria funcionan por la **localidad**, los programas tienden a acceder a los datos que están en los primeros niveles antes que en los mayores. Luego, el almacenamiento en los niveles mayores será más lento, grande y barato por bit.

El computador ofrece una gran reserva de memoria a un coste por bit reducido para poder servir datos a los programas a una velocidad rápida.

Fallos de caché. Tipos

En frío: cuando la caché está vacía.

Capacidad: cuando el conjunto de bloques activamente accedidos es mayor que el tamaño de la caché.

Conflicto: cuando, aunque el nivel k es lo bastante grande, múltiples datos mapean todos al mismo bloque k.

Lectura de caché

$S = 2^s$ sets - $E = 2^e$ líneas por conjunto - $B = 2^b$ bytes por bloque de caché (los datos)

Dirección de palabra: *t bits – s bits – b bits* donde t = etiqueta, s = índice conjunto, b = desplazamiento bloque (offset).

1. Localizar conjunto
2. Comprobar si hay línea en conjunto con tag coincidente
3. Si + línea válida → Acierto
4. Localizar dato empezando en el offset

Correspondencia directa: una línea por conjunto.

Si no hay coincidencia se expulsa la línea antigua y es reemplazada por el bloque solicitado.

Asociatividad: dos líneas por conjunto.

Si no hay coincidencia se elige una de las líneas del conjunto para ser expulsada y reemplazada.

Estructura de caché: políticas de escritura

Existen múltiples copias de los datos: L1, L2, L3, memoria principal, disco.

Si tenemos un **acierto de escritura** podemos tener:

- Escritura directa: la escritura se hace inmediatamente al siguiente nivel de la jerarquía
- Post – escritura: la escritura al siguiente nivel de la jerarquía se retrasa hasta que se reemplaza el bloque. Se necesita un bit de modificado *(dirty)* donde el bloque es diferente de la copia en memoria.

Si tenemos un **fallo de escritura** puede ocurrir:

- Asignación en escritura: el bloque se lleva a caché y allí se modifica

- Sin asignación en escritura: la escritura va directa a memoria, no se lleva el bloque a caché.

Configuraciones típicas:

- Escritura directa, sin asignación en escritura.
- Post – escritura con asignación en escritura.

3. La entrada / salida

E/S: puertos controladoras y canales

Los dispositivos de E/S son muy diferentes entre sí. Tienen usos y modos de funcionamiento diversos y muchas de sus características vienen determinadas por la tecnología. Pese a esto tienen ciertas reglas comunes como la programación ***(sondeo, interrupciones, DMA)*** y la comunicación entre la CPU y el dispositivo *(puertos E/S)*.
Los **puertos** son registros externos a la CPU integrados en un chip llamado **controladora del dispositivo**. Pueden realizar registro de datos: lectura/escritura del dato a comunicar entre CPU y dispositivo; de control: la CPU escribe "órdenes" con las tareas a realizar; de estado: información sobre cómo va el proceso.
Los **canales** o buses son líneas de comunicación entre distintos componentes que se organizan jerárquicamente.

Programación de los dispositivos de E/S

Direccionamiento y acceso a los puertos E/S. 2 modos:

- Espacio de direcciones de E/S mapeado en memoria: parte del espacio se asocia a los dispositivos de E/S, instrucciones de lectura y escritura convencionales y se captura por la *controladora de dispositivo* correspondiente.
- Espacio de direcciones de E/S aislado: necesita instrucciones especiales.

Los **mecanismos de comunicación CPU-dispositivos** son por sondeo, por interrupciones y por interrupciones + DMA.

E/S por sondeo *(polling)*

La CPU realiza todo el trabajo. Sondea el puerto correspondiente leyendo el registro estado. Puede realizar un sondeo continuo con bucles de espera activa o periódico (el ratón) con la CPU comprobando periódicamente si el usuario lo ha movido, en caso afirmativo el SO informa al programa asociado.
El **dispositivo** informa a la CPU modificando su registro de estado.
Un inconveniente de este mecanismo es que la CPU sondea muchas veces el dispositivo comprando que no ha cambiado por lo que se obtiene una pérdida de tiempo.

E/S por interrupciones

Asíncrona con respecto al programa en ejecución. Llegan en cualquier momento, mientras la CPU está realizando otra tarea.
Fases de una transacción de E/S:
1. La CPU encarga al dispositivo la realización de una transferencia usando el registro de control
2. La CPU continúa con otra tarea en lugar de estar chequeando la finalización de la transferencia
3. El dispositivo avisa a la CPU a través de una interrupción cuando la transacción de E/S ha finalizado. Cada dispositivo tendrá su número de interrupción asociada.
4. La CPU actúa según corresponda. Puede usar los resultados de la transferencia.

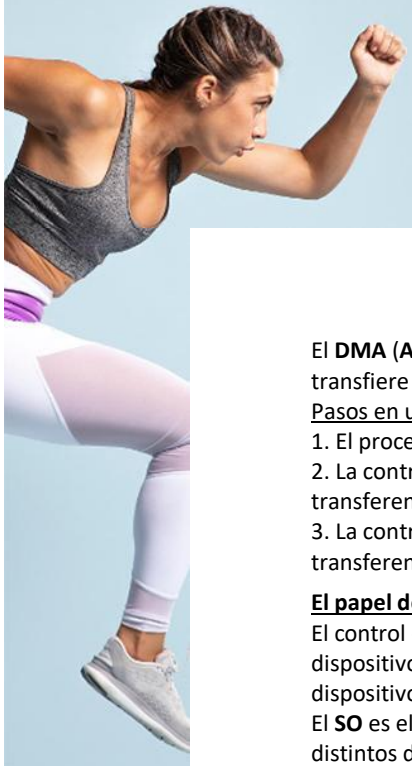
Las ventajas frente al **sondeo** son que la CPU delega parte del trabajo en el dispositivo de E/S. No gasta ciclos útiles en comprobar si el dispositivo de E/S está listo.

Tratamiento de las interrupciones

1. La CPU deja lo que esté haciendo para atender la interrupción.
2. Se guarda la información mínima para poder reanudar el proceso interrumpido. Como mínimo se guarda el registro de PC (punto en el que estaba el programa cuando llegó la interrupción).
3. La CPU pasa a ejecutar otro código que atiende la interrupción: **rutina de servicio de interrupción** (RSI). Parte del SO y guarda los registros que modifica antes de hacerlo.
4. Al finalizar la RSI, se recupera el estado guardado y se reanuda el proceso interrumpido. Esto provoca la reanudación del proceso de forma completamente transparente.

Interrupciones + DMA

El **sondeo** y las **interrupciones** son adecuados para dispositivos con **escaso ancho de banda** ya que reduce el coste de la controladora de dispositivo (CPU y SO realizan casi todo el trabajo) y cada transferencia de una palabra da lugar a una secuencia de instrucciones de CPU para transferirla. En dispositivos con **gran ancho de banda** se producen transferencias de grandes bloques de datos que mantienen el procesador ocupado demasiado tiempo y por eso son bastante inadecuados.



El **DMA** (**Acceso directo a memoria**) es implementado mediante un chip especializado ***(controladora de DMA)*** y transfiere datos desde el dispositivo a la memoria o viceversa sin tener q intervenir un procesador.
Pasos en una transferencia DMA:
1. El procesador inicializa la controladora de DMA.
2. La controladora de DMA va pidiendo el bus y, cuando lo consigue, va realizando tantas operaciones de transferencia entre el dispositivo y memoria como sean necesarias.
3. La controladora de DMA genera una interrupción informando al procesador de la finalización de la transferencia o de una condición de error.

El papel del sistema operativo
El control de dispositivos de E/S es heterogéneo y complicado. El **SO** mantiene el control del estado de los dispositivos, conoce las direcciones de los puertos para leer/escribir los datos y las órdenes asociadas a cada dispositivo, maneja los errores, etc.
El **SO** es el primer programa que se carga al arrancar el ordenador. Carga las rutinas de petición de E/S de los distintos dispositivos y las posibles RSI asociadas *(drivers)*. Sólo él conoce los puertos, órdenes, etc.
El usuario solicita sus servicios a través de mecanismos sencillos con llamadas al sistema que tienen parámetros bien definidos: *syscall*. Estas llamas preestablecidas serán la única forma de acceder a la E/S, siendo esto una especie de mecanismo de seguridad.

Tecnologías de almacenamiento
Discos duros: almacenamiento de bits como campo magnético al cual se accede por vía electro-mecánica.
Memoria no volátil *(flash)*: almacenamiento de bits como carga eléctrica. Estructura 3D con 100+ niveles de celda y 3 bits de datos por celda.

Tecnologías de memoria no-volátil
La **DRAM** y la **SRAM** son memorias volátiles, pierden la información cuando se apagan.
Las **memorias no volátiles** (NVMs) mantienen su valor tras apagado. ROM, EEPROM, **memorias flash**,etc. Se usan en programas *firmware* almacenados en ROM y en discos de estado sólidos.

Discos. Geometría y capacidad
En la **geometría de un disco** encontramos que consta de varios **platos**, cada uno con dos **superficies**. Cada superficie consta de anillos concéntricos, **pistas**. Cada pista consta de **sectores**, separados por huecos. El sector es la unidad mínima de lectura/escritura. *Normalmente 512 bytes x sector*.
Capacidad:

(# bytes/sector) x (avg. #sectors/track) x (# tracks/Surface) x (# surfaces/platter) x (# platters/disk)

Discos. Tiempos de acceso
El **tiempo de acceso a disco** está dominado por tiempo de búsqueda del sector + latencia de rotación. El primer bit de un sector es el más costoso mientras que el resto de bits del sector son a coste 0.
Tiene varios órdenes de magnitud mayor que las memorias volátiles (es más lento que la memoria SRAM y que la DRAM).

Discos de estado sólido
Páginas: 512KB to 4KB, agrupadas en bloques: 32 – 128 páginas.
Los datos se leen / escriben en unidades de páginas. Una página se puede escribir únicamente cuando su bloque ha sido borrado por completo. El agotamiento de un bloque llega tras ~100.000 escrituras.
Acceso secuencial más rápido que acceso aleatorio. Escrituras más lentas que lecturas.
Modificar una página implica mover el resto de las páginas a un bloque nuevo. La **capa traducción flash** acumula series pequeñas de escrituras antes de escribir un bloque.
Como no tiene partes móviles, son más rápidas y consumen menos. Pero, el agotamiento es progresivo, se van desgastando. También, las capas son más caras por byte.

Tema 4. El sistema operativo

1. Aspectos fundamentales sobre el Sistema Operativo (SO)

Ubicación del sistema operativo

El SO se sitúa entre el hw y las aplicaciones de usuario.
Nos ofrece servicios que facilitan el uso del computador (llamadas al sistema, interfaces de usuario, sistemas de fichero, lanzamientos de programas, etc) y administra los recursos hw (CPU, memoria y E/S) de manera que controla la eficiencia de los recursos para el beneficio de las aplicaciones.

Abstracciones que ofrece el SO

Ficheros: abstracción de los dispositivos de E/S
Memoria virtual: abstracción de la memoria principal y los dispositivos de E/S de almacenamiento (discos)
Procesos: abstracción de procesador + memoria + E/S para un programa

Procesos

Abstracción del SO para un **programa en ejecución**: procesador + memoria + E/S.
El SO nos permite ser el único programa en ejecución en el sistema, tener uso exclusivo del hw, ejecutar su código sin interrupciones y mantener en la memoria del sistema sus datos y código del programa.
Los **procesos** solicitan acciones sobre el hardware a través de **llamadas al sistema** que ofrece el **SO**. Puede haber varios procesos ejecutándose de forma recurrente.
La CPU se puede compartir entre procesos concurrentes mediante **cambios de contexto** donde el **contexto del proceso** incluye su contador de programa (PC) y los datos en los registros y en memoria. Cuando se cambia de proceso en ejecución se guarda el contexto actual y se carga el del nuevo proceso.
Un proceso puede tener varios flujos de ejecución: **hilos**. Éstos comparten el mismo código y datos globales del proceso y permiten una ejecución más rápida de aplicaciones cuando hay un soporte nativo de la CPU.

Hilos

Gracias a los **hilos** sabemos que un proceso puede tener varios flujos de ejecución. Todos estos se ejecutan en el contexto del proceso compartiendo el mismo código y datos globales. Pero, cada hilo tiene sus propios datos locales.
Cada vez está más extendido el modelo programación multi-hilo ya que resulta más fácil compartir datos entre hilos que entre procesos. También, un programa se ejecuta más rápido si hay múltiples procesadores disponibles.

Memoria virtual

Abstracción sobre la memoria principal para dar **a cada proceso** la *ilusión* de que tiene para sí **toda la memoria en exclusiva**.
Proporciona a cada proceso un **espacio de direcciones virtual** que funciona gracias a una sofisticada interacción entre el hw y el sw del SO.
La **idea principal** se basa en almacenar el contenido de la memoria virtual de cada proceso en disco y usar la memoria principal como caché de disco.

Ficheros

Son una secuencia de bytes tratada como una unidad lógica de almacenamiento para datos que necesitan persistencia. Contiene datos arbitrarios y es válido para cualquier clase de datos.
En los sistemas UNIX todos los dispositivos de E/S se modelan como ficheros y por ello la programación es **independiente del dispositivo**.

Interfaces de usuario: GUI vs CLI

La **interfaz de usuario** permite al usuario dar órdenes al sistema. Tenemos de dos tipos:

A - **Gráfico** (**GUI**). *Ratón, ventanas, paneles, atajos de teclado, etc.*
Muestran al usuario una visión sencilla del sistema y se basan en un gestor de ventanas que permite arrancar aplicaciones, trabajar con varias actividades, manipular ficheros, etc.

B - **Línea de comandos** (**CLI**). *Terminal con órdenes tecleadas con diversos parámetros y opciones.*
Los intérpretes de comandos pueden ejecutar comandos internos como cd o exit y externos como ls o cp.

Visiones de un SO

Podemos ver un SO como:
Máquina extendida: ofrece un uso más sencillo del computador con programas de sistema y llamadas al sistema.
Administrador de recursos: controla y coordina que todos los recursos hw se manejen de forma eficiente.

Flujo de control regular y excepcional

Cada núcleo del procesador lee y ejecuta una **secuencia de instrucciones**, **de una en una**. Esto se conoce como **el flujo de control de la CPU**.

Los mecanismos para cambiar el flujo de control (secuencial) en un programa permiten reaccionar a cambios en el **estado del programa** *(cambios en los valores de las variables)*.

Si tenemos cambios en el **estado del sistema** podemos reaccionar ante ellos usando, por ejemplo, *Ctrl-C* para interrumpir el proceso.

El sistema necesita mecanismos para el **flujo de control excepcional** mediante combinación de soporte hw y **sw del SO**.

¿Cuándo se ejecuta el código del SO?

Excepción: transferencia del flujo de control al núcleo del SO en respuesta a algún evento. Hay dos tipos.

A - Excepciones **asíncronas**: causadas por **eventos externos** al procesador. También son conocidas como ***interrupciones hardware***. Se generan al activar el pin de interrupción del procesador.

B - Excepciones **síncronas**: causadas por eventos que ocurren como resultado de la ejecución de una instrucción del programa. **Llamadas al sistema** (intencionada), **fallos** (involuntarios pero recuperable), **abortos** (involuntario e irrecuperable).

Llamadas al sistema: interfaz programas y SO

Las **llamadas al sistema** definen el interfaz entre programas de usuario y el SO. Hay distintos tipos:

1. **Procesos**. Creación de nuevos o terminación, etc.
2. **Acceso a dispositivos**. Apertura y cierre de ficheros, lectura y escritura.
3. **Gestión de la memoria**. Solicitud dinámica de espacio, liberación de espacio, etc.
4. **Manipulación de sistema de ficheros**. Creación y borrado de directorios, manipulación de permisos, etc.
5. **Otros**. Sincronización entre procesos, etc.

Modos de ejecución: usuario vs núcleo

Es necesario un soporte hardware para garantizar la seguridad del sistema y evitar que el código usuario ejecute determinadas instrucciones privilegiadas. Por eso la CPU tiene dos modos de funcionamiento.

El **modo USUARIO** permite solo instrucciones *normales* de la CPU y prohíbe cualquier tipo de acceso directo al hw. Normalmente es usado por los procesos de usuario.

El **modo NÚCLEO** permite ejecutar todo el repertorio de instrucciones de la CPU, incluyendo el acceso al hw. Solo el núcleo (SO) tiene permiso para ejecutarse en este modo.

Transición modo usuario ←→ modo núcleo

Modo usuario → modo núcleo

El proceso realiza explícitamente una **llamada** a un servicio del SO. Se produce una **interrupción hardware** donde se reclama la atención inmediata del SO. Puede ocurrir que haya alguna **excepción** o **fallo** en el proceso de ejecución e impida continuar.

Modo usuario ← modo núcleo

Regreso de una llamada al sistema o tras realizar una RSI. Siempre viene precedido de la **restauración del contexto** del proceso a reanudar.

Arranque del sistema operativo

Cuando encendemos un ordenador, la CPU ejecuta un programa en ROM que realiza un autodiagnóstico del hardware y comprueba que todo esté bien. Lee el disco y ejecuta el programa *cargador* para dejar residente el núcleo básico ***(kernel)*** del SO en memoria. El **kernel** toma el control y establece las estructuras internas básicas. Cuando el núcleo se ha inicializado, se lanza *init* para lanzar procesos auxiliares y *demonios* según la configuración. Finalmente, se lanza el proceso de *login* que nos permite identificarnos y empezar a trabajar.

Núcleo del sistema operativo

El **kernel** del SO es la parte más interna del SO y la primera en arrancar. Reside en memoria continuamente y no es un proceso separado, es decir, se ejecuta como parte de uno existente.

En Linux, el espacio del kernel está siempre presente en memoria y mapea las mismas direcciones físicas para todos los procesos. El código y los datos de kernel están siempre direccionables, listos para manejar interrupciones y llamadas al sistema en todo momento.

Linux: características

Es un clon de **Unix** iniciado en 1991. Tiene **open source**, su código fuente disponible. Es multiplataforma y muy portable por estar casi todo **escrito en C**. Multiusuario/multitarea.

Tiene protección máxima entre procesos y soporte para múltiples sistemas de ficheros, protocoles de red e infinidad de dispositivos de E/S.
También tiene múltiples **distribuciones** *(Ubuntu)* y está presente en todos los segmentos.

Distribución = núcleo de Linux + software variado + sistema configuración

2. Gestión de procesos

Programa vs Proceso

Un **programa** es código ejecutable almacenado en disco. Puede ser cargado en memoria para ser ejecutado con un concepto estático. Es una entidad **pasiva**, fichero que contiene instrucciones guardadas en disco.
Un **proceso** es un programa en ejecución, con recursos asignados. Tiene un concepto dinámico con estado cambiante. Es una entidad **activa**, con un PC *(siguiente instrucción a ejecutar)* y un conjunto de recursos asociados. Es la **unidad de trabajo** del SO y es mucho más que simplemente el código del programa.
Programa → proceso: cuando un fichero ejecutable se carga en memoria y se ejecuta.
Aunque dos procesos pueden estar asociados al mismo programa, se consideran dos secuencias separadas de ejecución.

Abstracciones ofrecidas por el proceso

El concepto de **proceso** proporciona a cada programa dos abstracciones clave:
1. *Flujo de control lógico*. Cada programa parece tener uso exclusivo de la CPU y se obtiene gracias a un mecanismo del SO llamado cambio de texto.
2. *Espacio de direcciones privado*. Cada programa parece tener uso exclusivo de la memoria principal y se obtiene gracias a un mecanismo del SO llamado memoria virtual.

Multiprogramación: la ilusión

El computador ejecuta muchos procesos simultáneamente como aplicaciones de uno o varios usos, tareas de segundo plano o procesos de monitorización de los dispositivos de E/S.

Multiprogramación: la realidad

Un único procesador ejecuta múltiples procesos concurrentemente donde se alterna el uso de la CPU, el sistema de memoria virtual gestiona los espacios de direcciones de cada proceso y los valores de los registros para los procesos que no están en ejecución se guardan en memoria.
Al interrumpir un proceso que está en ejecución, usando la CPU, el SO debe primero guardas los valores de los registros de la CPU en ese instante. De esta manera, luego podrá ser reanudado. El kernel del SO mantiene en su memoria los registros guardados de todos los procesos que no se están ejecutando.
El SO decide mediante el planificador de procesos cuál es el siguiente que debe ejecutarse. Restaura los guardados e intercambia el espacio de direcciones (cambio de contexto).

Multiprogramación: la realidad (en la actualidad)

Hay procesadores multinúcleo con varias CPUs integradas en el mismo chip, todas comparten la memoria principal y cada núcleo de procesamiento puede ejecutar un proceso separado.

Cambio de contexto entre procesos

El **flujo de control** pasa de un proceso a otro mediante un cambio de contexto.

Estado de proceso

Los procesos cambian de estado conforme se ejecutan y su **estado** suele estar definido por la actividad actual de dicho proceso.
Los **estados** son: nuevo, listo, en ejecución, esperando y terminado.

Bloque de control de un proceso (PCB)

En el SO, cada proceso se representa mediante un **bloque de control de proceso** (**PCB**). Éste contiene mucha información relativa al proceso como el identificador de proceso (**pid**), el estado, el contador de programa, los registros de la CPU, información sobre la planificación, gestión de la memoria, E/S y la contabilidad como el tiempo uso de CPU.

Cambio de contexto entre procesos

El SO permite la **multiprogramación** (uso compartido de la CPUs entre distintos procesos).
El **planificador** es parte del núcleo del SO y es responsable de optimizar el uso de la(s) CPU(s).
Todos los procesos van avanzando con sensación de simultaneidad. Puede no haber paralelismo real, si solo hay una CPU.

Planificación de procesos *(scheduling)*

El **objetivo** de la **multiprogramación** es tener algún proceso en ejecución en todo momento para maximizar la utilización de la CPU; y, permitir a los usuarios interactuar con los procesos mientras se ejecutanpara cambiar la CPU con una frecuencia muy elevada.

Para que estos objetivos se cumplan, el **planificador de procesos** selecciona un proceso disponible para su ejecución de la **cola de procesos listos** que están en la memoria principal. La **cola de dispositivo** son los que están bloqueados en espera de un dispositivo de E/S particular.

Los procesos se colocan en la **cola de procesos listos *(ready queue)*** al crearse, en espera de ser despachados.

Una vez el proceso está ejecutándose en la CPU:

a) Si el proceso emite una petición de E/S → se le pone en cola de E/S del dispositivo correspondiente
b) Si el proceso crea un nuevo proceso hijo y espera a que termine → se pone en la cola de procesos bloqueados
c) Si el proceso es sacado a la fuerza de la CPU, como resultado de una interrupción → se vuelve a poner en la *ready queue*

En los casos *a)* y *b)* el proceso cambiará de estado → se le pondrá otra vez en la cola de procesos listos.

Creación de procesos. Árbol de procesos

Llamada al sistema: **fork()**.

Un proceso puede crear nuevos procesos donde el creador se llama, padre y los nuevos, hijos.

Llamada al sistema: **exec()**.

El proceso hijo puede optar por ejecutar un programa diferente al padre. El padre puede ejecutar en paralelo o esperar a que termine el proceso hijo.

El proceso **init()** sirve como proceso raíz para todos los procesos de usuario. Tiene **pid** 1 siempre. Una vez que se ha arrancado el sistema, puede crear otros procesos como *login, servidor ssh, etc.*

Los procesos hijos pueden crear otros procesos, formando un **árbol**.

3. Gestión de la memoria. Memoria virtual.

Gestión de la memoria

Cada proceso comparte la CPU y la memoria virtual con otros procesos.

Gestión de la CPU: conforme sube su demanda, los procesos se ralentizan. El SO se encarga de que los procesos usen la CPU por turnos.

Gestión de la memoria: si unos procesos demandan más memoria de la disponible, pueden hacer que otros no puedan ejecutarse. El SO hace lo posible para que cada proceso tenga toda la memoria que necesita.

Memoria virtual: es una abstracción ofrecida por el SO sobre la memoria principal para dar a cada proceso la ilusión de que tiene **toda la memoria** para sí mismo.

Memoria virtual: capacidades

1. **Facilita la utilización eficiente de la memoria principal**. Trata el RAM como una **caché** para un espacio de direcciones almacenado en el disco, transfiere datos entre disco y memoria principal y mantiene en esta última sólo las áreas activas del espacio de direcciones virtual de cada proceso.
2. **Simplifica la gestión de la memoria**. Es idéntico para todos los procesos, proporciona a cada proceso un espacio de direcciones uniforme y facilita la compartición de código/datos entre procesos.
3. **Protege el espacio de direcciones de cada proceso**. Aísla la memoria de cada proceso.

La VM es una solución elegante para la cooperación de hardware/software.

***Caching* mediante VM**

Conceptualmente, la **memoria virtual** se puede ver como un array de bytes contiguos almacenados en disco. Este contenido se cachea en memoria física donde la **DRAM** actúa como caché del disco.

La **página** es la unidad de transferencia entre los datos disco y la memoria física.

Gestión de memoria mediante VM

Cada proceso tiene su espacio virtual de direcciones privado. Esto simplifica la reserva de memoria y facilita la compartición de código y datos entre procesos.

Protección de memoria mediante VM

El SO mantiene una tabla de página (PT) que facilita la gestión del espacio de memoria de cada proceso. Las entradas de la tabla de páginas (PTEs) contienen bits con los permisos sobre la página y la MMU comprueba esos bits en cada acceso.

Direccionamiento virtual

Se usa en cualquier computador moderno y es una de las mejores ideas en la historia de los computadores ya que funciona de forma silenciosa y automática.

Espacio de direcciones virtual de un proceso

El SO proporciona a **cada proceso** una visión **uniforme** de la memoria. Esto se conoce como el **espacio de direcciones virtual**, el cual, está dividido en áreas.

a) **Código y datos** del programa. Se carga directamente del fichero ejecutable del programa en disco.
b) **Memoria montón (Heap)**. Espacio de memoria para reserva dinámica, en tiempo de ejecución.
c) **Librerías compartidas**. Librerías dinámicas cargadas necesarias para el funcionamiento del programa.
d) **Pila**. Espacio de memoria usado para soportar llamada a procedimientos.

La memoria virtual del kernel del SO tiene código y estructuras de datos del sistema operativo.

Memoria física vs Memoria virtual

Direcciones de **memoria física**: P bits. Puede variar entre CPUs aunque sean de la misma familia. La cantidad de memoria física instalada en una máquina puede ser menor que la cantidad máxima que puede ser accedida.
Direcciones de **memoria virtual**: V bits. Igual para todas las CPUs que implementan una misma arquitectura (ISA).

Paginación. Traducción de direcciones

El SO divide la memoria en trozos de igual tamaño llamados **páginas**. En la memoria virtual se llaman **páginas virtuales** (**VP**) y en la física, **páginas físicas** (**PP**) o **marcos**.
El SO mantiene la **correspondencia entre VP y PP**. No todas las VP de un proceso han de estar en memoria física a la vez, sólo se cargarán en ésta aquellas que cada proceso está usando.

Traducción de direcciones: tabla de páginas

La **tabla de páginas (PT)** mantiene la correspondencia entre páginas virtuales y físicas.
De la dirección virtual (VA) extraemos el número de página (VPN) → indexar PT. Cada entrada (PTE) indica si esa página virtual está en memoria física.

Tablas de páginas en la memoria del núcleo

El SO mantiene una tabla de páginas **por cada proceso**. Esta es una de las estructuras de datos clave en la gestión de la memoria donde parte de ella reside en memoria física, en la memoria del núcleo del SO.

Acierto de página

Si se hace referencia a una palabra en VM que está en memoria física ocurre un acierto en la *caché* DRAM.

Fallo de página

Si se hace referencia a una palabra en VM que **NO** está en memoria física ocurre un fallo en la *caché* DRAM.

Lanzamiento de un fallo de página

El programa de usuario escribe una dirección de memoria. Esa página de la memoria del usuario está en ese instante en disco. La MMU lanza una excepción por fallo de página que causa una llamada a un procedimiento del SO que maneja el fallo y eleva el nivel de privilegio de la CPU de modo usuario a modo núcleo.

Manejo de un fallo de página

Se produce un **fallo de página** cuando el acceso a la tabla de páginas no encuentra PP.
El manejador de fallo de página elige una VP en PP que será expulsada de la memoria física para hacer espacio y será escrita a disco.
Se produce un **acierto de página** cuando la instrucción que causa el fallo se reinicia. Se cono ce como **paginación bajo demanda** esperar a que se produzca un fallo para copiar la página de disco a DRAM.

Reanudación tras fallo de página

El manejador de fallo de página ejecuta una instrucción de retorno de interrupción *(iret en x86)*. Restaura el nivel de privilegio, regresa al código de usuario en el PC que causó el fallo de página y se re-ejecuta la instrucción sin causar fallo.

Manejo de excepciones por fallo de página

1. Referencia a una dirección virtual ilegal.
2. Excepción por protección.
3. Fallo de página convencional.

Memoria virtual y localidad

La VM parece muy ineficiente, pero funciona gracias a la **localidad**.

Los programas tienden a acceder a un conjunto de páginas virtuales activas llamado **conjunto de trabajo**.

- tamaño del conjunto trabajo < tamaño memoria principal: buen rendimiento
- tamaño memoria principal < tamaño conjunto trabajo: ocurre ***Thrashing*** donde el rendimiento cae en picado cuando las páginas se copian entre memoria y disco continuamente.

Hardware para acelerar la traducción V → P

El **TLB** es una pequeña caché hw asociativa que es parte de la MMU y mantiene correspondencias VPN ←→ PPN. Contiene entradas de la tabla de páginas para un pequeño número de páginas.

4. Gestión de la E/S. Sistema de ficheros.

Sistema de ficheros

Es el componente del SO encargado de gestionar los ficheros ofreciendo acceso eficiente a los discos al permitir que los datos puedan ser almacenados, localizados y recuperados fácilmente. Interactúa con el subsistema de E/S del kernel *(de más bajo nivel)*.

El **sistema de ficheros** está compuesto por **ficheros** y **directorios**. Ambos se identifican por su ruta.

El **SF** reside permanentemente en el almacenamiento secundario donde los datos almacenados en los ficheros pueden estar repartidos en varias áreas del disco.

Es la parte más visible del SO para la mayoría de los usuarios.

Tipos de ficheros. Atributos. Operaciones

Un **fichero** es una colección de información relacionada definida por su creador, identificada por un nombre, que se guarda en almacenamiento secundario.

Cada fichero tiene un **tipo** que indica su rol en el sistema. Los tipos en Linux:

- **Fichero regular**. Contiene datos arbitrarios
- **Directorio**. Índice para agrupar ficheros relacionados
- **Enlace simbólico**. Puntero indirecto a otro fichero/directorio
- **Otros**.

Cada fichero tiene una serie de **atributos**, entre los que se incluyen nombre, tipo, tamaño, etc.

Podemos realizar **operaciones** sobre ficheros como crear, leer, escribir, etc.

Ficheros regulares. Ficheros de texto

Un **fichero regular** contiene datos arbitrarios que no son relevantes al SO.

Las aplicaciones suelen distinguir entre **ficheros de texto**, regulares que contienen únicamente caracteres; y, **ficheros binarios**, contienen todo lo demás (JPEG,…). El núcleo del SO **no distingue** entre ficheros de texto y binarios.

Un **fichero de texto** es una secuencia de líneas de texto.

Fichero y dispositivos de E/S. Unix I/O

Otros tipos de **fichero** en Unix/Linux:

- **Sockets**. Comunicación entre procesos en hosts distintos.
- **Tuberías con nombre (FIFOs)**. Comunicación entre procesos en el mismo host.
- **Dispositivos de bloques/caracteres**.

Todos los dispositivos de E/S se presentan como ficheros gracias a la elegante correspondencia de ficheros a dispositivos E/S que hace que el SO ofrezca una interfaz muy simple **Unix I/O**.

Directorios. Jerarquía de directorios

Un **directorio** consiste en una lista de enlaces donde cada enlace mapea un nombre de fichero a un fichero. El tamaño del directorio viene dado por el número de entradas.

Cada directorio contiene al menos dos entradas de directorio: *.* y *..*

Todos los ficheros se organizan como una jerarquía anclada por el **directorio raíz**, llamado **/**.

Rutas a ficheros

La ubicación de los ficheros en la jerarquía de directorios se denota por su **ruta**. Hay dos tipos:

- **Absolutas**. Empiezan por / e indican la ruta desde el directorio raíz.
- **Relativas**. Denota la ruta desde el directorio de trabajo actual

Descriptores de fichero. Ficheros abiertos

Al operar con ficheros el SO debe buscar en el directorio la entrada correspondiente al fichero.
*La apertura vía **open()** devuelve un descriptor de fichero. El SO lo utiliza como índice en la **tabla de descriptores de un fichero**.*

Sistema de ficheros: Estructuras del kernel

Los procesos solicitan operaciones sobre ficheros indicando el descriptor del dichero devuelto por *open*.
El núcleo del SO mantiene una **tabla de descriptores** por cada proceso y una **tabla de ficheros abiertos**, la cual apunta al **bloque de control de fichero** (FCB).

Compartición de ficheros

Dos descriptores de fichero pueden compartir el mismo fichero en disco a través de dos entradas distintas en la tabla de ficheros abiertos.
Llamar a open dos veces con el mismo nombre de fichero como argumento

Gestión de la E/S: Ficheros abiertos

El SO mantiene cierta información por cada fichero abierto.
1. **Posición actual** en el fichero.
   Este puntero es diferente para cada proceso operando en el fichero y debe mantenerse separado de los atributos del fichero que se guardan en disco.
2. **Contador de aberturas**.
   Conforme se cierran los ficheros, el SO reutiliza las entradas de la tabla de ficheros abiertos para evitar quedarse sin espacio en la tabla.
3. **Ubicación en disco** del fichero.
   La mayoría de operaciones requieren la modificación de los datos en el fichero.
4. **Permisos de acceso**.
   Cada proceso abre un fichero en un determinado modo de acceso. Esta información se mantiene en la tabla de ficheros de cada proceso y se utiliza para permitir o denegar peticiones de E/S subsiguientes.

Estructura de E/S del núcleo. *Drivers*

El OS encapsula las particularidades de los distintos dispositivos de E/S: **manejadores de dispositivo**. Ocultan al subsistema de E/S del núcleo las diferencias entre las controladoras de dispositivo y presentan una interfaz de acceso a dispositivos uniforme: **independencia de dispositivo**.
Se identifican unas pocas clases generales de dispositivos de E/S donde a cada clase se accede a través de un conjunto de funciones estandarizado.
Los manejadores de dispositivo (**software**) son módulos del kernel, están hechos a medida del dispositivo específico y envían comandos para indicar las acciones que se deben emprender.
Las controladoras de dispositivo (**hardware**) son chips que controlan y operan la electrónica del periférico concreto, interfaz con el *driver*.

Acceso a la E/S

El programador usa los dispositivos de E/S solicitando el servicio al SO mediante una llamada al sistema donde se pasa a modo núcleo y se transfiere el control al SO. Éste organiza y coordina las peticiones de E/S de los distintos procesos, comprobando permisos. Usa un **manejador de dispositivo** adecuado para enviar comandos a la controladora del dispositivo donde el *driver* actúa como una especie de *traductor*. El **manejador de interrupción** asociado al dispositivo se encarga de transferir la información entre el dispositivo y los búferes del driver en memoria principal. Finalmente, si es una lectura, el kernel del SO copia los datos a la memoria del proceso y al terminar de servir la petición se regresa al código de usuario en el programa llamador.

Gestión de procesos y acceso a la E/S

Se realiza un esquema de tratamiento de una llamada al sistema:
1. Proceso A realiza llamada al SO *(syscall).*
2. Núcleo del SO lanza orden de E/S al dispositivo y pasa el control a otros procesos (B, C) mientras el dispositivo acaba su tarea.
3. Dichos procesos (B, C) pueden ir siendo a su vez interrumpidos por el planificador…
4. …hasta que el dispositivo de E/S termina su tarea y lanza interrupción (irq).
5. Núcleo entonces vuelve a tomar el control, y despierta el proceso bloqueado (A).

Tema 5. Lenguajes del computador: alto nivel, ensamblador y máquina

1. Introducción

Programas e instrucciones

Una **instrucción** es un conjunto de símbolos que representa una orden de operación para el computador. Se almacenan en memoria principal y se ejecutan en secuencia. Esta secuencia sólo se rompe por posibles instrucciones de salto.
Un **programa** es un conjunto *ordenado* de instrucciones que debe ejecutar el computador sobre los ***datos*** para procesarlos y obtener un resultado.

Codificación de las instrucciones

Cada instrucción indica una acción determinada a realizar por la CPU.
Leer un valor de memoria y copiarlo a un registro de la CPU
Las **instrucciones** en última instancia se codifican como ristras de bits de longitud fija o variable, dependiendo de la arquitectura.
La **unidad de control (UC)** de la **CPU** analizará e interpretará los distintos campos para saber qué operación se debe llevar a cabo, operandos de entrada y donde dejar el resultado.
Distintos tipos de instrucciones utilizarán distintos formatos.

Tratamiento de las instrucciones

La **unidad de control (UC)** de la **CPU** mantiene:

- **PC. Contador de programa**. Contiene la dirección de memoria de la instrucción a ejecutar.
- **RI. Registro de instrucción**. Contiene la instrucción a ejecutar.

Tipos de instrucciones

Cuantas más instrucciones haya la **UC** es más compleja y se requieren más bits en el campo código de operación.

- RISC: pocas instrucciones, sencillas y ejecutadas en pocos ciclos
- CISC: muchas instrucciones, complejas y muchos ciclos de reloj

Podemos distinguir 3 tipos de instrucciones:
1. **Instrucciones de movimiento de datos**. A/desde/entre registros CPU/direcciones de memoria
2. **Instrucciones aritmético–lógicas.** Suma resta, etc. Operaciones punto flotante
3. **Instrucciones de salto**. Condicionales, incondicionales, manejo de subrutinas

2. Jerarquía de traducción

Lenguajes de alto nivel

Las instrucciones que procesa la CPU están almacenadas en memoria principal en binario. Son instrucciones máquina.
Los **lenguajes de programación** son instrucciones representadas simbólicamente. Entre ellos se encuentra el lenguaje ensamblador (bajo nivel) y el de alto nivel.

Lenguajes de alto nivel

Permiten expresar los programas en un lenguaje formal, relativamente cercano a la forma de pensar. Existen multitud de *paradigmas* y de lenguajes concretos.

El sistema de compilación

Los programas escritos en un lenguaje son traducidos por ostros programas (**programas traductores**).
La **compilación** es el proceso general de traducción y tiene 4 fases:
1. Preprocesado: edición inicial con inserciones de código
2. Compilación: traducción de lenguaje de alto nivel a lenguaje de bajo nivel
3. Ensamblado: traducción de lenguaje de bajo nivel a código máquina
4. Enlazado: resolución de llamadas a funciones de librerías

Programas traductores

El **compilador** transforma el código de lenguaje de alto nivel (en texto ASCII) a **lenguaje ensamblador**.
El **ensamblador** transforma el programa en lenguaje ensamblador en un *fichero de código objeto* en **lenguaje máquina**.

4. Compilación e interpretación. El lenguaje Phyton

Compilación vs Interpretación

La traducción del programa fuente a código máquina se realiza con anterioridad a ser ejecutado o durante la propia ejecución de este.

De los **lenguajes compilados** *(C++)* sabemos que el programa será traducido a código máquina para poder ser ejecutado. La ejecución de los programas es más rápida que los códigos interpretados. Esto se debe a que el desarrollador tiene mayor control sobre los aspectos del hardware. Sin embargo, se necesita mucho tiempo de compilación en cada prueba y de recompilación cada vez que sufre un cambio. También tienen mucha dependencia de la plataforma que genera el código binario.

De los **lenguajes interpretados** *(Python)* sabemos que el código fuente no es traducido directamente al código máquina de la CPU destino. Un programa **intérprete** lee y ejecuta el código. Este código es portable y la compilación a bytecode suele ser muy rápida. El problema es que un programa de este estilo es mucho más lento a la hora de ejecutarlo y los usuarios tendrán que instalar una máquina virtual en su ordenador que consume recursos.

Podemos tener un **enfoque híbrido** *(compilación + interpretación)* donde el compilador traduce a código intermedio y el intérprete lo ejecuta.

Compilación *ahead-of-time* AOT

Es la traducción del código fuente de un programa con anterioridad a su ejecución.

El código fuente se convierte al lenguaje máquina de la CPU obteniendo un fichero que contiene el programa ejecutable. Optimiza los programas **para una máquina destino concreta**. Se puede dar la compilación cruzada (compilación para una plataforma distinta a la usada en la compilación).

Las ventajas son que el compilador puede optimizar el código máquina en función de la CPU haciendo que se ejecute más rápido y que no se necesita una máquina virtual para poder ejecutar el programa.

Los inconvenientes son que los programas ejecutables dependen de la plataforma y que compilar a código máquina lleva más tiempo que compilar a *bytecode*.

Interpretación

El programa se ejecuta a partir de un código que no es el lenguaje de la máquina. Podemos interpretar el código a distintos niveles: alto y bytecode.

La interpretación necesita un **intérprete del lenguaje** donde parte del **entorno de ejecución** de dicho lenguaje. Éste puede generar representaciones intermedias del código.

Compilación *just-in-time* JIT

Conocida como traducción dinámica. Combinación de dos enfoques: AOT + interpretación.

No genera código ejecutable en código máquina con anterioridad a la ejecución y la traducción se realiza en el momento en el que se ejecuta.

Puede partir del código fuente aunque para mejorar el rendimiento se utiliza la pre-compilación al bytecode. La máquina virtual analiza lo que se está ejecutando.

La interpretación de bytecode y la compilación JIT son muy similares.

Bytecode

Representación intermedia del código fuente, usada internamente por el compilador o máquina virtual. La idea es traducir el código fuente a una representación intermedia genérica y usar un intérprete o máquina virtual para cada plataforma.

Las instrucciones son independientes del ISA de la CPU donde se ejecutará el programa.

El objetivo principal es la **portabilidad** (independencia de la plataforma).

El origen del nombre viene de usar 1 byte para codificar cada instrucción.

- Una ISA virtual abstrae los detalles de un procesador concreto donde la mayoría de VMs que interpretan bytecode son máquinas de pila.
- Está diseñado para ejecutar eficientemente mediante un intérprete de software gracias a su sintaxis de fácil interpretación.
- Las instrucciones son simples y estructuradas.

Como el lenguaje intermedio es sencillo, se basa en componentes adicionales para funcionar:

- Tablas de valores que forman parte de la memoria del programa
- Una máquina de pila que mantiene el estado del programa conforme se ejecuta. Esta pila almacena valores de entrada de los comandos y los resultados de las operaciones.

El lenguaje Python

Fue creado a principio de los 90. Es un lenguaje interpretado con tipado dinámico, multiplataforma y orientado a objetos.

Se traduce a código intermedio bytecode en la primera ejecución o cuando se indica que se quiere realizar la compilación. Es un lenguaje muy flexible.

Tiene una sintaxis clara y sencilla, tipado dinámico, gestor de memoria eficiente y transpatente y gran cantidad de librerías disponibles. Pero, es inadecuado para programación de bajo nivel.

Ejecución de programas en Python

Podemos trabajar de dos modos:

- Sesión interactiva. Se ejecuta línea a línea como si fuesen comandos del Shell
- Intérprete de programa. Se ejecuta un programa en un archivo.

Fases en la ejecución de un programa en Python:

1. Inicialización.
   Reserva y copia de datos en las diferentes estructuras de datos necesarias para el proceso Python
2. Compilación.
   Análisis léxico y sintáctico, y generación de código en bytecode.
3. Interpretación.
   Ejecución de código bytecode.

Interpretación en Python

La máquina virtual de Python PVM ejecuta las instrucciones en bytecode. Ésta se ejecuta sobre el hardware. Cuando queremos interpretar en la PVM, primero se inicializa el intérprete y los hilos de ejecución. Después, las zonas de memoria del programa a interpretar. Finalmente, se crea un **bucle de evaluación** de instrucciones en bytecode.